IJCAI
International Joint Conferences on Artificial Intelligence

# Proceedings of the IJCAI 2005 Workshop on Planning and Learning in A Priori Unknown or Dynamic Domains

Vadim Bulitko, Sven Koenig (editors)

Edinburgh, United Kingdom
August 1, 2005

### Preface

Many testbeds in Artificial Intelligence are a priori known and static. However, as Artificial Intelligence systems are applied to more realistic problems, they need to be able to cope with a priori unknown and dynamic domains. The state of an environment, for instance, can change over time due to the presence of other (cooperative or competitive) agents or exogenous events. In such domains, an agent cannot predict the effects of its actions with certainty and thus needs to be able to plan under uncertainty with its current domain model as well as learn improved domain models, often in real time. Example domains include real-time games, mobile robotics, space applications, supply-chain management, and decision support for crisis management. The following research areas are important for planning and learning in a priori unknown or dynamic domains:

- finding robust plans and enhancing the robustness of existing plans,
- time-dependent planning,
- trading off planning time and plan quality,
- interleaving planning and plan execution,
- fast replanning and plan adaptation,
- planning when and what to sense,
- multi-agent planning and learning,
- life-long learning of domain models,
- acting robustly in the presence of unknown but important state features or discovering them,
- reinforcement learning, and
- learning with very few training data.

These issues are being explored by researchers from different communities in Artificial Intelligence. They are, for instance, being studied in search, generative and reactive planning, scheduling, agents, robotics, reasoning and meta reasoning, and machine learning. This workshop brought together researchers from these communities to learn about each other's approaches, form long-term collaborations, and cross-fertilize the different areas to accelerate progress towards scaling up to larger and more realistic applications. If you would like to keep informed on the developments in the area, you can check the webpage of the workshop periodically or subscribe to the mailing list as follows.

### Mailing list

To sign up, please send an email to ijcai05-pludd-request@cs.ualberta.ca with the line: `subscribe' in the body of the message. This will add you to the mailing list for the workshop. The mailing list is moderated against spam abuse.

### Web page

http://www-rcf.usc.edu/~skoenig/workshop.html

**Program Committee**

Douglas Aberdeen, The Australian National University (Australia)
Blai Bonet, Universidad Simon Bolivar (Venezuela)
Adi Botea, University of Alberta (Canada)
James Bruce, Carnegie Mellon University (USC)
Greg Calbert, DSTO (Australia)
Alessandro Cimatti, IRST (Italy)
Alan Fern, Oregon State University (USA)
Natalia Gardiol, MIT (USA)
Russ Gayle, University of North Carolina (USA)
Enrico Giunchiglia, Universita di Genova (Italy)
Russ Greiner, University of Alberta (Canada)
Charles Gretton, National ICT Australia (Australia)
Richard Korf, University of California at Los Angeles (USA)
Lihong Li, Rutgers University (USA)
Michael Littman, Rutgers University (USA)
Yaxin Liu, University of Texas (USA)
Frederic Maire, Queensland University of Technology (Australia)
Mausam, University of Washington (USA)
Martin Mueller, University of Alberta (Canada)
Doina Precup, McGill University (Canada)
Ioannis Refanidis, University of Macedonia (Greece)
Matt Rudary, University of Michigan (USA)
Jonathan Schaeffer, University of Alberta (Canada)
Masashi Shimbo, Nara Institute of Science and Technology (Japan)
Bill Smart, Washinton University in St. Louis (USA)
Trey Smith, Carnegie Mellon University (USA)
Finnegan Southey, University of Alberta (Canada)
Peter Stone, University of Texas at Austin (USA)
Nathan Sturtevant, University of Alberta (Canada)
Rich Sutton, University of Alberta (Canada)
Sylvie Thiebaux, The Australian National University (Australia)
Vincent Vidal, CRIL (France)
Eric Wiewiora, University of California in San Diego (USA)
Martin Zinkevich, Brown University (USA)

**Chairs**

Vadim Bulitko (Canada)
Sven Koenig (USA)

**Additional credits**

Carlos Guestrin (coordinating IJCAI'05 workshops)
Alex Morrice (cover photograph)

# Papers

# Learning Planning Rules in Noisy Stochastic Worlds

**Luke S. Zettlemoyer**
MIT CSAIL
lsz@csail.mit.edu

**Hanna M. Pasula**
MIT CSAIL
pasula@csail.mit.edu

**Leslie Pack Kaelbling**
MIT CSAIL
lpk@csail.mit.edu

## Abstract

We present an algorithm for learning a model of the effects of actions in noisy stochastic worlds. We consider learning in a 3D simulated blocks world with realistic physics. To model this world, we develop a planning representation with explicit mechanisms for expressing object reference and noise. We then present a learning algorithm that can create rules while also learning derived predicates, and evaluate this algorithm in the blocks world simulator, demonstrating that we can learn rules that effectively model the world dynamics.

## 1 Introduction

One of the goals of artificial intelligence is to build systems that can act in complex environments as effectively as humans do: to perform everyday human tasks, like making breakfast or unpacking and putting away the contents of an office. Any robot that hopes to solve these tasks must be an integrated system that perceives the world, understands it in an, at least naively, human manner, and commands motors to effect changes to it. Unfortunately, the current state of the art in reasoning, planning, learning, perception, locomotion, and manipulation is so far removed from human-level abilities that we cannot even contemplate working in an actual domain of interest. Instead, we choose to work in domains that are its almost ridiculously simplified proxies.

One popular such proxy, used since the beginning of work in AI planning [Fikes & Nilsson, 1971] is a world of stacking blocks. This *blocks world* is typically formalized in some version of logic, using predicates such as *on*$(a, b)$ and *clear*$(a)$ to describe the relationships of the blocks to one another. Blocks are always very neatly stacked; they don't fall into jumbles. In this paper, we will work in a slightly less ridiculous version of the blocks world, one constructed using a three-dimensional rigid-body dynamics simulator [ODE, 2004]. An example domain configuration is shown in Figure 1. In this simulated blocks world, blocks are not always in tidy piles; blocks sometimes slip out of the gripper; and piles sometimes fall over. We would like to learn models that enable effective action in this world.

Unfortunately, previous approaches to action model learning cannot solve this problem. The algorithms that learn deterministic rule descriptions [Shen & Simon, 1989; Gil, 1994; Wang, 1995] have limited applicability in a stochastic world. One approach [Pasula, Zettlemoyer, & Kaelbling, 2004] has extended those methods to learn probabilistic STRIPS rules, but this representation cannot cope with the complexity of the simulated blocks world. The work of Benson (1996), which extends a deterministic ILP [Lavrač & Džeroski, 1994] learning algorithm that is robust to noise in the training set, would, perhaps, come the closest, but it lacks the ability to handle complex action effects such as piles of blocks falling over. We address this challenge by developing a more flexible algorithm that creates models that include mechanisms for referring to objects and abstracting away rare or highly complex action outcomes, and also invents new concepts that help determine when actions will have different effects.

When learning these models, we assume that the learner has access to training examples that show how the world changes when an action is executed. The learning problem is then one of density estimation. The learner must estimate the distribution over next states of the world that executing an action will cause.

In the rest of this paper, we first present our representation, showing how these extensions are added to probabilistic STRIPS rules. Then, we develop a learning algorithm for these rules. Finally, we evaluate these learned rules in the simulated blocks world.

## 2 Representation

This section describes representations for the set $\mathcal{S}$ of possible states of the world, the set $\mathcal{A}$ of possible actions the agent can take, and the probabilistic transition dynamics $\Pr(s'|s, a)$, where $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$. In each case, we use a subset of a relatively standard first-order logic with equality. States and actions are ground; the rules used to express the transition dynamics quantify over variables.

We begin by defining a language that includes a set of predicates $\Phi$ and a set of functions $\Omega$. There are three types of functions in $\Omega$: traditional functions, which range over

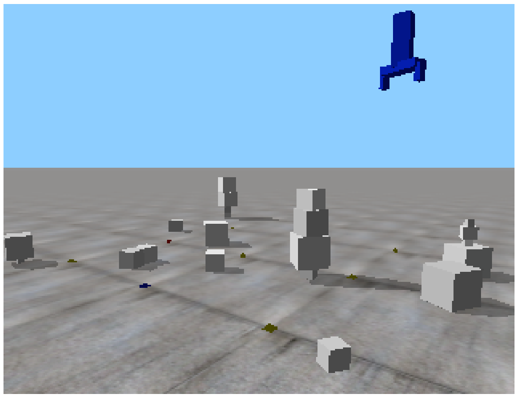


Figure 1: A screen capture of the simulated blocks world. The blocks come in various sizes, visible here, and various colors. The gripper can perform two macro actions: *pickup*, which centers the gripper above a block, lowers it until it hits something, closes it, and raises the gripper; and *puton*, which centers the gripper above a block, lowers until it encounters pressure, opens it, and raises it.

objects; discrete-valued functions, which range over a predefined discrete set of values; and integer-valued functions, which range over a finite subset of the integers.

## 2.1 State Representation

In this work, we assume that the environment is completely observable; that is, that the agent is able to perceive an unambiguous and correct description of the current state.[1] Each state consists of a particular configuration of the properties of and relations between objects for all of the objects in the world, where those individual objects are denoted using constants. State descriptions are conjunctive sentences that list the truth values for all of the possible groundings of the predicates and functions with the constants. When writing them down, we will make the closed world assumption and omit the negative literals.

As an example, let us consider representing the state of a simple blocks world, using a language that contains the predicates *on*, *table*, *clear*, *inhand*, and *inhand-nil*. The objects in this world include two blocks, $c_1$ and $c_2$, a table $t$, and a gripper. The sentence

$$\textit{on}(c_1, c_2) \wedge \textit{on}(c_2, t) \wedge \textit{inhand-nil} \wedge \textit{clear}(c_1) \wedge \textit{table}(t) \qquad (1)$$

represents a blocks world where the gripper holds nothing and the two blocks are in a single stack on the table.

## 2.2 Action Representation

Actions are represented as positive literals whose predicates are drawn from a special set, $\alpha$, and whose terms are drawn from the set of constants C associated with the world $s$ where the action is to be executed.

For example, in the simulated blocks world, $\alpha$ contains *pickup*$/1$, an action for picking up blocks, and *puton*$/1$, an action for putting down blocks. The action literal *pickup*$(c_1)$ could represent the action where the gripper attempts to pickup the block $c_1$ in the state represented in Sentence 1.

## 2.3 World Dynamics Representation

We begin by defining probabilistic STRIPS rules [Blum & Langford, 1999]. Next, we describe the changes we have made to the rules to enable them to model more complex worlds. Then, we explain how the representation language is extended to allow for the construction of additional predicates and functions. Finally, we show how to use a set of rules to provide a model of world dynamics.

**Probabilistic STRIPS rules**

Each probabilistic STRIPS rule specifies the conditions under which it applies, as well as a small number of simple action *outcomes*—sets of changes that might occur in tandem. More formally, a rule for action $z$ has the form

$$\forall \bar{x}.\Psi(\bar{x}) \wedge z(\bar{x}) \rightarrow \bullet \begin{cases} p_1 & \Psi'_1(\bar{x}) \\ \ldots & \ldots \\ p_n & \Psi'_n(\bar{x}) \end{cases},$$

where $\bar{x}$ is a vector of variables, $\Psi$ is the *context*, a formula that might hold of them at the current time step, $\Psi'_1 \ldots \Psi'_n$ are *outcomes*, formulas that might hold in the next step, and $p_1 \ldots p_n$ are positive numbers summing to 1, representing a probability distribution over the outcomes. Traditionally, the action $z(\bar{x})$ must contain every $x_i \in \bar{x}$. We constrain $\Psi$ and $\Psi'$ to be conjunctions of literals constructed from the predicates in $\Phi$ and the variables $\bar{x}$ as well as equality statements comparing a function (taken from $\Omega$) of these variables to a value in its range. In addition, $\Psi$ is allowed to contain greater-than and less-than statements.

We say that a rule *covers* a state $\Gamma(\mathrm{C})$ and action $a(\mathrm{C})$ if there exists an action substitution $\sigma$ mapping the variables in $\bar{x}$ to C (note that there may be fewer variables in $\bar{x}$ than constants in C) such that $\Gamma(\mathrm{C}) \models \Psi(\sigma(\bar{x}))$ and $a(\mathrm{C}) = z(\sigma(\bar{x}))$. That is, if there exists a substitution of constants for variables that, when applied to antecedent, grounds it so that it is entailed by the state and, when applied to the rule action, makes it equal the action the rule covers.

Here is an example using the language of Sentence 1:

$$\begin{array}{l} \textit{pickup}(X, Y): \\ \textit{on}(X, Y), \textit{inhand-nil} \\ \quad \rightarrow \begin{cases} .80: & \neg\textit{on}(X, Y), \textit{inhand}(X), \neg\textit{inhand-nil}, \\ & \textit{clear}(Y) \\ .10: & \neg\textit{on}(X, Y), \textit{on}(X, t), \textit{clear}(Y) \\ .10: & \text{no change} \end{cases} \end{array}$$

The context of this rule states that $X$ is on $Y$, and there is nothing in the gripper. The rule covers the world of Sentence 1 and action *pickup*$(c_1, c_2)$ under the action substitution $\{X \rightarrow c_1, Y \rightarrow c_2\}$. The first outcome describes the situation where the gripper successfully picks up the block $X$, and the second indicates that $X$ falls onto the table.

Let us now consider what a rule that covers the state and action can tell us about the possible subsequent states. Each outcome directly specifies that $\Psi'(\sigma(\bar{x}))$ holds at the next step, but this may be only an incomplete specification of the state. We use the frame assumption to fill in the rest; every literal

[1] This is a very strong, and ultimately indefensible assumption; one of our highest priorities for future work is to extend this to the case when the environment is partially observable.

that would be needed to make a complete description of the state that is not included in $\Psi'(\sigma(\bar{x}))$ is retrieved, with its associated truth value or equality assignment, from $\Gamma(\mathrm{C})$.

Thus, each outcome $\Psi'_i$ can be used to construct a new state $s'_i$, which will occur with probability $p_i$. The probability that a rule $r$ assigns to moving from state $s$ to state $s'$ when action $a$ is taken, $P(s'|s,a,r)$, can be calculated as:

$$\begin{aligned} P(s'|s,a,r) &= \sum_{i=1}^{n} P(s',\Psi'_i|s,a,r) \\ &= \sum_{i=1}^{n} P(s'|\Psi'_i,s,a,r)P(\Psi'_i|s,a,r) \qquad (2) \end{aligned}$$

where $P(\Psi'_i|s,a,r)$ is $p_i$, and the outcome distribution $P(s'|\Psi'_i,s,a,r)$ is a deterministic distribution that assigns all of its mass to the relevant $s'$. If $P(s'|\Psi'_i,s,a,r) = 1.0$, that is, if $s'$ is the state that would be constructed given that rule and outcome, we say that the outcome $\Psi'_i$ *covers* $s'$.

### Noisy Deictic Rules

We extend probabilistic STRIPS rules in two ways: by permitting them to refer to objects not mentioned in the action description, and by adding a noise outcome.

### Deictic References

Relational planning representations use a list of action variables to abstract over the objects in the world. For example, $\mathit{pickup}(X,Y)$ abstracts the identity of the block *X* to be picked up and the block *Y* that *X* will be picked up from. This abstraction allows the rules to compactly encode actions that affect many different objects. Part of the challenge of creating effective rules is to determine what to abstract over. Traditionally, this is done when defining the set of actions, since abstraction can occur only in the action argument list.

We have developed *deictic references*, an extension of a mechanism originally introduced by Benson (1996), as a way of introducing additional variables to the rules. Our rule learning algorithm uses them to learn useful abstractions that were not initially included in the action arguments.

We extend probabilistic STRIPS rules as follows. Each rule is augmented with a list, $D$, of deictic references. A reference consists of a variable $v_i$ and a restriction $\rho_i$, which is a set of literals that define $v_i$ with respect to the variables $\bar{x}$ in the action and the other $v_j$ such that $j < i$.

For example, the $\mathit{pickup}(X,Y)$ rule we saw earlier can be rewritten to use deictic references as follows:

$$\begin{array}{l} \mathit{pickup}(X) : \{\ Y : \mathit{on}(X,Y),\ \ Z : \mathit{table}(Z)\ \} \\ \mathit{inhand\text{-}nil} \\ \qquad \rightarrow \left\{ \begin{array}{ll} .80: & \neg\mathit{on}(X,Y), \mathit{inhand}(X), \neg\mathit{inhand\text{-}nil}, \\ & \mathit{clear}(Y) \\ .10: & \neg\mathit{on}(X,Y), \mathit{on}(X,Z), \mathit{clear}(Y) \\ .10: & \text{no change} \end{array} \right. \end{array}$$

where *Y* is now defined as a deictic reference that names that unique thing that *X* is *on*. In many ways, this is a more natural encoding because it makes explicit the fact that the only block that *Y* should ever name is the one that *X* is on. This reduces the number of arguments to the action, which can greatly increase planning efficiency [Gardiol & Kaelbling, 2003]. Note also that, in this representation, different rules for the same action can abstract over different sets of objects.

To use rules with deictic references, we must extend our procedure for computing rule coverage to ensure that all of the deictic references can be resolved. The deictic variables are bound by starting with bindings for $\bar{x}$ and working sequentially through the deictic references $D$, using their restrictions to determine their unique bindings. If a deictic variable does not have a unique binding—if it has either no possible bindings, or several—it fails to refer, and the rule fails to cover the state and action.

### The Noise Outcome

Probability models of the type we have seen thus far, ones with a small set of possible outcomes, are not sufficiently flexible to handle noisy domains where there may be a large number of possible action effects that are highly unlikely and yet hard to model—such as all the configurations that may result when a tall stack of blocks topples. It would be inappropriate to model such effects as impossible, and yet we don't have the space or inclination to model each of them as an individual outcome.

We handle this issue by augmenting each rule with an additional *noise outcome*. This outcome has the probability $p_{noise} = 1 - \sum_1^n p_i$, but no associated $\Psi'$; we are declining to model in detail what happens to the world in such cases.

As an example, consider the rule

$$\begin{array}{l} \mathit{pickup}(X) : \{\ Y : \mathit{on}(X,Y),\ \ Z : \mathit{table}(Z)\ \} \\ \mathit{inhand\text{-}nil} \\ \qquad \rightarrow \left\{ \begin{array}{ll} .80: & \neg\mathit{on}(X,Y), \mathit{inhand}(X), \neg\mathit{inhand\text{-}nil}, \\ & \mathit{clear}(Y) \\ .10: & \neg\mathit{on}(X,Y), \mathit{on}(X,Z), \mathit{clear}(Y) \\ .05: & \text{no change} \\ .05: & \text{noise} \end{array} \right. \end{array}$$

where noise can happen with a probability of $0.05$. Here, the noise outcome might model the fact that towers sometimes fall over when you are picking up a block.

Since we are not explicitly modeling the effects of noise, we can no longer calculate the transition probability $\Pr(s'|s,a,r)$ using Equation 2: we lack the distribution over next states given the noise outcome, $P(s'|noise,s,a,r)$. Instead, we substitute a worst case constant bound $p_{min} \leq P(s'|noise,s,a,r)$ everywhere this distribution would be required, and bound the transition probability as

$$\begin{aligned} \hat{P}(s'|s,a,r) &= p_{noise}p_{min} + \sum_{i=1}^{n} P(s'|\Psi'_i,s,a,r)p_i \\ &\leq P(s'|s,a,r). \end{aligned}$$

In this way, we create a partial model that allows us to ignore unlikely or overly complex state transitions while still learning and acting effectively. [2]

[2] $P(s'|noise,s,a,r)$ could alternately be any well-defined probability distribution that models the noise of the world. However, we would have to ensure that this distribution does not assign probability to worlds that are impossible (for example, blocks worlds where blocks are floating in midair), because this would complicate planning. We will leave the exploration of this alternative approach to future work.

## 2.4 Background knowledge

In the rule semantics as described so far, the same set of primitive predicates has been used to construct all the elements of the rule. However, it is often useful to divide the predicates and functions of the language into two sets: a set of primitives whose values are observed directly, and represented within a state, and a set of additional predicates and functions that can be derived from these primitives, and so do not need to be represented directly. The derived predicates and functions can then be used in the antecedents, but not in the outcomes—a good thing, since it can be difficult to describe how the values of the derived predicates change directly. (The predicate *above*, the transitive closure of *on*, is an example of a hard-to-update predicate.) This has been found to be essential for representing certain advanced planning domains [Edelkamp & Hoffman, 2004].

We define such background knowledge using a *concept language* that includes existential quantification, universal quantification, transitive closure, and counting. Consider the situation where the only primitive predicates are *on* and *table*. Quantification is used for defining predicates such as *inhand*. Transitive closure is included in the language via the Kleene star and plus and defines predicates such as *above*. Finally, counting is included using a special quantifier # which returns the number of objects for which a formula is true. It is useful for defining integer-valued functions such as *height*. The derived predicates can be used in the context and deictic reference restrictions.

As an example, here is a deictic noisy rule for attempting to pick up block *X* together with the background knowledge used by this rule:

$$\begin{array}{l} \mathit{pickup}(X) : \left\{ \begin{array}{l} Y : \mathit{topstack}(Y,X), \\ Z : \mathit{on}(Y,Z), \\ T : \mathit{table}(T) \end{array} \right\} \\ \mathit{inhand\text{-}nil}, \mathit{height}(Y) < 9 \\ \quad \rightarrow \left\{ \begin{array}{ll} .80 : & \neg \mathit{on}(Y,Z) \\ .10 : & \neg \mathit{on}(Y,Z), \mathit{on}(Y,T) \\ .05 : & \text{no change} \\ .05 : & \text{noise} \end{array} \right. \end{array} \tag{3}$$

$$\begin{array}{rcl} \mathit{clear}(V_1) & := & \neg\exists V_2.\mathit{on}(V_2,V_1) \\ \mathit{inhand}(V_1) & := & \neg\exists V_2.\mathit{on}(V_1,V_2) \\ \mathit{inhand\text{-}nil} & := & \neg\exists V_2 \mathit{inhand}(V_2) \\ \mathit{above}(V_1,V_2) & := & \mathit{on}^*(V_1,V_2) \\ \mathit{topstack}(V_1,V_2) & := & \mathit{clear}(V_1) \wedge \mathit{above}(V_1,V_2) \\ \mathit{height}(V_1) & := & \#V_2.\mathit{above}(V_1,V_2)) \end{array}$$

The rule is far more complicated than our running example: it deals with the situation when the block to be picked up, *X*, is in the middle of a stack. It is now useful to abstract over even more objects: the deictic variable *Y* identifies the (unique) block on top of the stack, and the deictic variable *Z*—the block under $Y$. As might be expected, the gripper succeeds in lifting *Y* with a high probability.

```
LearnRuleSet(E)
Inputs:
  Training examples E
Computation:
  Initialize rule set $R$ to contain only the default rule
  While better rules sets are found
    For each search operator $O$
      Create new rule sets with $O$, $R_O = O(R, \mathbf{E})$
      For each rule set $R' \in R_O$
        If the score improves ($S(R') > S(R)$)
          Update the new best rule set, $R = R'$
Output:
  The final rule set $R$
```

Figure 2: *LearnRuleSet* Pseudocode. This algorithm performs greedy search through the space of rule sets. At each step a set of search operators each propose a set of new rule sets. The highest scoring rule set is selected and used in the next iteration.

## 2.5 Action Models

Individual rules define the world dynamics only in specific situations; a general description is provided by an *action model*, which consists of some background knowledge and a set of rules $R$ that, together, define the action dynamics of a world. Given an action $a$ and state $s$, the rule $r \in R$ that covers $s$ and $a$ is used to predict the effects of $a$ in $s$. When no such rule exists, we use the *default rule*. This rule has an empty context and two outcomes: a no-change outcome (which, in combination with the frame assumption, models the situations where nothing changes), and, again, a noise outcome (modeling all other situations). This rule allows noise to occur in situations where no single non-default rule applies; the probability assigned to the noise outcome in the default rule specifies a kind of "background noise" level. The default rule is also used when more than one rule covers $s$ and $a$. However, in general, we hope to learn rule sets where the rules are mutually exclusive.

# 3 Learning

In this section, we describe an algorithm for learning action models from training examples that describe action effects. More formally, each training example $E \in \mathbf{E}$ is a state, action, next state triple $(s, a, s')$ where states are described in terms of primitive functions and predicates.

We divide the problem of learning action models into two parts: learning background knowledge, and learning a rule set $R$. First, we describe how to learn a rule set given some background knowledge. Then, we show how to derive new useful concepts.

## 3.1 Learning Rule Sets

The *LearnRuleSet* algorithm takes a set of examples $\mathbf{E}$ and a fixed language of primitive and derived predicates. It then performs a greedy search through the space of possible rule sets as described in the pseudocode in Figure 2.

The search starts with a rule set that contains only the noisy default rule. At every step, we take the current rule set and apply all our search operators to it to obtain a set of new rule sets. We then select the rule set $R$ that maximizes the scoring metric

$$S(R) = \sum_{(s,a,s')\in\mathbf{E}} \log(\hat{P}(s'|s,a,r_{(s,a)})) - \alpha \sum_{r\in\mathbf{R}} PEN(r)$$

where $r_{(s,a)}$ is the rule that covers $(s,a)$, $\alpha$ is a scaling parameter, and the penalty $PEN(r)$ is the number of literals in the rule $r$. Ties in $S(R)$ are broken randomly.

As a greedy search through the space of rule sets, *LearnRuleSet* is similar in spirit to previous work [Pasula, Zettlemoyer, & Kaelbling, 2004]. However, adapting that work to handle our representation extensions involved substantial redesign of the algorithm, including changing the initial rule set, the scoring metric, and the search operators.

**Search Operators**

Each search operator $O$ takes as input a rule set $R$ and a set of training examples $\mathbf{E}$, and creates a set of new rule sets $R_O$ to be evaluated by the greedy search loop. There are eight search operators. We first describe the most complex operator, *ExplainExamples*, and then the most simple one, *DropRules*. Finally, we present the remaining six operators which all share a common computational framework, outlined in Figure 4.

- *ExplainExamples* takes as input a training set $\mathbf{E}$ and a rule set $R$ and creates new rule sets that contain additional rules modeling the training examples that were covered by the default rule in $R$. Figure 3 shows the pseudocode for this algorithm, which considers each training example $E$ that was covered by the default rule in $R$, and executes a three-step procedure. The first step builds a large and specific rule $r'$ that describes this example; the second step attempts to trim this rule, and so generalize it so as to maximize its score, while still ensuring that it covers $E$; and the third step creates a new rule set $R'$ by copying $R$ and integrating the new rule $r'$ into this new rule set.

  As an illustration, let us consider how steps 1 and 2 of *ExplainExamples* might be applied to the training example $(s,a,s') = (\{\mathit{on}(a,t),\mathit{on}(b,a)\},\mathit{pickup}(b),\{\mathit{on}(a,t)\})$, when the background knowledge is as defined for Rule 3. Step 1 builds a rule $r$. It creates a new variable $X$ to represent the object $b$ in the action; then, the action substitution becomes $\sigma = \{X \to b\}$, and the action of $r$ is set to $\mathit{pickup}(X)$. The context of $r$ is set to the conjunction $\mathit{inhand\text{-}nil}, \neg\mathit{inhand}(X), \mathit{clear}(X), \mathit{height}(X) = 2, \neg\mathit{on}(X,X), \neg\mathit{above}(X,X), \neg\mathit{topstack}(X,X)$ Then, in Step 1.2, *ExplainExamples* attempts to create deictic references that name the constants whose properties changed in the example. In this case, the only changed literal is $\mathit{on}(b,a)$, so $C = \{a\}$; a new deictic variable $Y$ is created and restricted, and $\sigma$ is extended to be $\{X \to b, Y \to a\}$. The resulting rule $r'$ looks as follows:

$$\mathit{pickup}(X) : \left\{ Y : \begin{array}{l} \neg\mathit{inhand}(Y), \neg\mathit{clear}(Y), \mathit{on}(X,Y), \\ \mathit{above}(X,Y), \mathit{topstack}(X,Y), \\ \neg\mathit{above}(Y,Y), \neg\mathit{topstack}(Y,Y), \\ \neg\mathit{on}(Y,Y), \mathit{height}(Y) = 1 \end{array} \right\}$$
$$\mathit{inhand\text{-}nil}, \neg\mathit{inhand}(X), \mathit{clear}(X), \mathit{height}(X) = 2, \neg\mathit{on}(X,X), \neg\mathit{above}(X,X), \neg\mathit{topstack}(X,X)$$
$$\to \{ \;\; 1.0 : \neg\mathit{on}(X,Y)$$

```
ExplainExamples$(R, \mathbf{E})$
Inputs:
  A rule set $R$
  A training set $\mathbf{E}$
Computation:
  For each example $(s,a,s') \in \mathbf{E}$ covered by the default
    rule in $R$
    Step 1: Create a new rule $r$
      Step 1.1: Create an action and context for $r$
        Create new variables to represent the arguments of $a$
        Use them to create a new action substitution $\sigma$
        Set $r$'s action to be $\sigma^{-1}(a)$
        Set $r$'s context to be the conjunction of boolean
          and equality literals that can be formed using the
          variables and the available functions and predicates
          (primitive and derived) and that are entailed by $s$
      Step 1.2: Create deictic references for $r$
        Collect the set of constants $C$ whose properties changed
          from $s$ to $s'$, but which are not in $a$
        For each $c \in C$
          Create a new variable $v$ and extend $\sigma$ to map $v$ to $c$
          Create $\rho$, the conjunction of literals containing $v$
            that can be formed using the available variables,
            functions, and predicates, and that are entailed by $s$
          Create deictic reference $d$ with variable $v$ and
            restriction $\sigma^{-1}(\rho)$
          If $d$ uniquely refers to $c$ in $s$, add it to $r$
    Step 2: Trim literals from $r$
      Create a rule set $R'$ containing $r$ and the default rule
      Greedily trim literals from $r$ while $r$ still covers $(s,a,s')$
        and $R'$'s score improves
    Step 3: Create a new rule set containing $r$
      Create a new rule set $R' = R$
      Add $r$ to $R'$ and remove any rules in $R'$ that
        cover any examples $r$ covers
      Recompute the set of examples that the default rule in $R'$
        covers and the parameters of this default rule
    Add $R'$ to the return rule sets $R_O$
Output:
  A set of rule sets, $R_O$
```

Figure 3: *ExplainExamples* Pseudocode. This algorithm attempts to augment the rule set with new rules covering examples currently handled by the default rule.

  In Step 2, *ExplainExamples* trims this rule to remove the invariably true literals, like $\neg\mathit{on}(X,X)$, and the redundant ones, like $\neg\mathit{inhand}()$ and $\neg\mathit{clear}(Y)$, to give

$$\mathit{pickup}(X) : \{ Y : \;\; \mathit{on}(X,Y), \mathit{height}(Y) = 0 \;\; \}$$
$$\mathit{inhand\text{-}nil}, \mathit{clear}(X), \mathit{height}(X) = 1$$
$$\to \{ \;\; 1.0 : \neg\mathit{on}(X,Y)$$

  which is then integrated into the rule set.

- *DropRules* cycles through all the rules in the current rule set, and removes each one in turn from the set. It returns a set of rule sets, each one missing a different rule.

The remaining six operators create new rule sets from the input rule set $R$ by repeatedly choosing a rule $r \in R$ and making changes to it to create one or more new rules. These new rules are then integrated into $R$, just as in *ExplainExamples*, to create a new rule set $R'$. Figure 4 shows the the general pseudocode for how this is done. The operators vary in

```
**OperatorTemplate**$(R, \mathbf{E})$
**Inputs:**
  Rule set $R$
  Training examples $\mathbf{E}$
**Computation:**
  Repeatedly select a rule $r \in R$
    Create a copy of the input rule set $R' = R$
    Create a new set of rules, $N$, by making changes to $r$
    For each new rule $r' \in N$
      Estimate new outcomes for $r'$ with the *InduceOutcomes*
        algorithm described by Pasula et al (2004)
      Add $r'$ to $R'$ and remove and rules in $R'$ that
        cover any examples $r'$ covers
    Recompute the set of examples that the default rule in $R'$
      covers and the parameters of this default rule
    Add $R'$ to the return rule sets $R_O$
**Output:**
  The set of rules sets, $R_O$
```

Figure 4: *OperatorTemplate* Pseudocode. This algorithm is the basic framework that is used by six different search operators. Each operator repeatedly selects a rule, uses it to make $n$ new rules, and integrates those rules into the original rule set to create a new rule set.

the way they select rules and the changes they make to them. These variations are described for each operator below:

- *DropLits* selects every rule $r \in R$ $n$ times, where $n$ is the number of literals in the context of $r$; in other words, it selects each $r$ once for each literal in its context. It then creates a new rule $r'$ by removing that literal from $r$'s context; $N$ of Figure 4 is simply the set containing $r'$.
- *DropRefs* selects each rule $r \in R$ once for each deictic reference in $r$. It then creates a new rule $r'$ by removing that deictic reference from $r$.
- *ChangeRanges* selects each rule $r \in R$ $n$ times for each equality or inequality literal in the context, where $n$ is the total number of values in the range of each literal. Each time it selects $r$ it creates a new rule $r'$ by replacing the numeric value of the chosen (in)equality with another other possible value from the range. Thus, if $f()$ ranges over $[1 \ldots n]$, *ChangeRange* would, when applied to a rule containing the inequality $f() < i$, construct rule sets in which $i$ is replaced by all other integers in $[1 \ldots n]$.
- *SplitOnLits* selects each rule $r \in R$ $n$ times, where $n$ is the number of literals that are absent from the rule's context. (The set of absent literals is obtained by applying the available predicates and functions—both primitive and derived—to the variables defined in the rule, and removing those already present.) It then constructs a set of new rules. In the case of predicate and inequality literals, it creates one rule in which the positive version of the literal is inserted into the context, and one in which it is the negative version. In the case of equality literals, it constructs a rule for every possible value the equality could take. This time, $N$ contains all these rules.
- *AddLits* selects each rule $r \in R$ $n$ times, where $n$ is the number of predicate-based literals that are absent from the rule's antecedent. It constructs a new rule by inserting that literal into the earliest place in which the its variables are all well-defined. If the literal contains no deictic variables, this will be the context, otherwise this will be the restriction of the last deictic variable mentioned in the literal. (If $V_1$ and $V_2$ are deictic variables and $V_1$ appears first, $p(V_1, V_2)$ would be inserted into the restriction of $V_2$.)
- *AddRefs* selects each rule $r \in R$ $n$ times, where $n$ is the number of literals that can be constructed from variables in $r$ and a new variable $v$. It then creates a new rule by adding a deictic reference with the variable $v$ and a restriction defined by one of the literals.

We have found that all of these types of operators are consistently used during learning. While this set of operators is heuristic, it is complete in the sense that every rule set can be constructed from the initial rule set—although, of course, there is no guarantee that the scoring metric will lead the greedy search to the global maximum.

### 3.2 Learning Background Knowledge

We learn background knowledge using an algorithm which iteratively constructs increasingly complex concepts, then tests their usefulness by running *LearnRuleSet* and checking whether they appear in the learned rules. The first set is created by applying the operators in Figure 5 to literals built with the original language. Subsequent sets of concepts are constructed using the literals that proved useful on the latest run; concepts that have been tried before, or that are always true or always false across all examples, are discarded. The search ends when none of the new concepts prove useful.

Since our concept language is quite rich, overfitting (e.g., by learning concepts that can be used to identify individual examples) can be a serious problem. We handle this in the expected way: by introducing a penalty term, $\alpha' c(R)$, to create a new scoring metric

$$S'(R) = S(R) - \alpha' c(R)$$

where $c(R)$ is the number of distinct concepts used in the rule set $R$ and $\alpha'$ is a scaling parameter. This new metric $S'$ is now used by *LearnRuleSet*; it avoids overfitting by favoring rule sets that use fewer derived predicates.

## 4 Evaluation

In this section, we demonstrate that noise outcomes and derived predicates are necessary to learn good action models for the physics-based blocks world simulator of Figure 1, and also that our algorithm is capable of discovering the relevant background knowledge. We accomplish this by learning a variety of action models and then comparing their performance on a simple planning task.

All the experiments are set in a world containing twenty blocks. The observed, primitive predicates include $on(X, Y)$ (which is true if block $X$ exerts a downward force on $Y$), $size(X)$, $color(X)$, and the typing predicate *table*$(X)$. There

$$\begin{array}{rcl}
p(X) & \rightarrow & n := QY.p(Y) \\
p(X_1, X_2) & \rightarrow & n(Y_2) := QY_1.p(Y_1, Y_2) \\
p(X_1, X_2) & \rightarrow & n(Y_1) := QY_2.p(Y_1, Y_2) \\
p(X_1, X_2) & \rightarrow & n(Y_1, Y_2) := p^*(Y_1, Y_2) \\
p(X_1, X_2) & \rightarrow & n(Y_1, Y_2) := p^+(Y_1, Y_2) \\
p_1(X_1), p_2(X_2) & \rightarrow & n(Y_1) := p_1(Y_1) \wedge p_2(Y_1) \\
p_1(X_1), p_2(X_2, X_3) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1) \wedge p_2(Y_1, Y_2) \\
p_1(X_1), p_2(X_2, X_3) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1) \wedge p_2(Y_2, Y_1) \\
p_1(X_1, X_2), p_2(X_3, X_4) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1, Y_2) \wedge p_2(Y_1, Y_2) \\
p_1(X_1, X_2), p_2(X_3, X_4) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1, Y_2) \wedge p_2(Y_2, Y_1) \\
p_1(X_1, X_2), p_2(X_3, X_4) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1, Y_2) \wedge p_2(Y_1, Y_1) \\
p_1(X_1, X_2), p_2(X_3, X_4) & \rightarrow & n(Y_1, Y_2) := p_1(Y_1, Y_2) \wedge p_2(Y_2, Y_2) \\
f(X) = c & \rightarrow & n() := \#Y.f(Y) = c \\
f(X) \leq c & \rightarrow & n() := \#Y.f(Y) \leq c \\
f(X) \geq c & \rightarrow & n() := \#Y.f(Y) \geq c
\end{array}$$

Figure 5: Operators used to invent a new predicate $n$. Each operator takes as input one or more literals, listed on the left. The $p$s represent old predicates; $f$ represents an old function; $Q$ can refer to $\forall$ or $\exists$; and $c$ is a numerical constant. Each operator takes a literal and returns a concept definition. These operators are applied to all of the literals used in rules in a rule set to create new predicates.

were five sizes and five colors, both uniformly distributed. The color attribute is a distractor. The sizes complicate the action dynamics, both because they influence stack stability, and because the gripper does best with blocks of average size, and is unable to grasp giant blocks at all. The training data were generated by repeatedly attempting to perform random actions in random simulator states and noting the result. The random starting states were generated by randomly placing blocks on each other, or on the table. The last block was sometimes placed in the gripper.

## 4.1 Planning

Since we have no true model to compare the rule sets to, we evaluate them by using them to plan. We implemented a simple planner based on the sparse sampling algorithm [Kearns, Mansour, & Ng, 2002], which treats the domain as a Markov Decision Problem (MDP) [Puterman, 1999]. Given a state $s$, it creates a tree of states (of predefined depth and branching factor) by sampling forward using a transition model, computes the value of each node using the Bellman equation, and selects the action that has the highest value. In our implementation, the transition function is defined using an action model and the reward function is defined by hand.

We adapt the algorithm to handle noisy outcomes, which do not predict the next state, by estimating the value of the unknown next state as a fraction of the value of staying in the same state: i.e., we sample forward as if we had stayed in the same state and then scale down the value we obtain. Our scaling factor was $0.75$, our depth was three, and our branching factor was five.

This scaling method is only a guess at what the value of the unknown next state might be; because noisy rules are partial models, there is no way to compute the value explicitly. In the future, we would like to explore methods that learn to associate values with noise outcomes. For example, the value of the outcome where a tower of blocks falls over is different if the goal is to build a tall stack of blocks than if the goal is to put all of the blocks on the table.

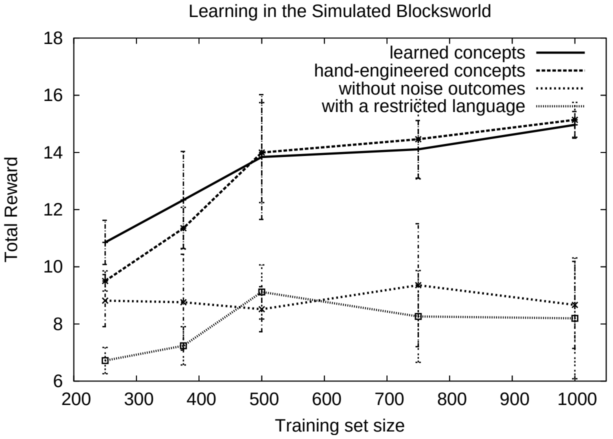


Figure 6: The performance of various action model variants as a function of the number of training examples. All data points were averaged over five runs each of three rule sets learned on different training data sets. For comparison, the average reward for performing no actions is 9.2, and the reward obtained when a human directed the gripper averaged 16.2.

## 4.2 Experiments

We set our planner the task of building tall stacks: our reward function was the average height of the blocks in the world. The plans were executed for ten time steps. The scaling parameters $\alpha$ and $\alpha'$ (associated respectively with the rule complexity penalty term, and the background knowledge complexity penalty term) were set to $1.0$ and $5.0$. The noise probability bound $p_{min}$ was set to $0.00001$.

To evaluate the overall quality of the learned rules, we did an informal experiment to measure the reward achieved when a human domain expert directed the robot arm. (Note that humans have an advantage over the planner, since they can view the entire 3D world while the planner only has access to the information encoded in the *on*, *height*, and *size* relations.)

### Results

We tested four action model variants, varying the training set size; the results are shown in Figure 6. The curve labeled ‘learned concepts’ represents the full algorithm as presented in this paper. Its performance approaches that obtained by a human expert, and is comparable to that of the algorithm labeled ‘hand-engineered concepts’ that did not do concept learning, but was, instead, provided with hand-coded versions of the concepts *clear*, *inhand*, *inhand-nil*, *above*, *topstack*, and *height*. The concept learner discovered all of these, as well as other useful predicates, e.g., $p(X, Y) := \mathit{clear}(Y) \wedge on(Y, X)$, which we will call *onclear*. This could be why its action models outperformed the hand-engineered ones slightly on small training sets. In domains less well-studied than the blocks world, it might be less obvious what the useful concepts are; the concept-discovery technique presented here should prove helpful.

The remaining two model variants obtained rewards comparable to the reward for doing nothing at all. (The planner did attempt to act during these experiments, it just did a poor job.) In one variant, we used the same full set of predefined concepts but the rules could not have noise outcomes. The requirement that they explain every action effect led to significant overfitting and a decrease in performance. The other rule set was given the traditional blocks world language, which does not include *above*, *topstack*, or *height*, and allowed to learn rules with noise outcomes. We also tried a full-language variant where noise outcomes were allowed, but deictic references were not: the resulting rule sets contained only a few very noisy rules, and the planner did not attempt to act at all. The poor performance of these ablated versions of our representation shows that all three of our extensions are essential for modeling the simulated blocks world domain.

**Example Learned Rules**

To get a better feel for the types of rules learned, here are two interesting rules learned by the full algorithm.

$$\mathit{pickup}(X) : \left\{ \begin{array}{l} Y : \mathit{onclear}(X,Y),\ Z : \mathit{on}(Y,Z), \\ T : \mathit{table}(T) \end{array} \right\}$$

$$\mathit{inhand\text{-}nil}, \mathit{size}(X) < 2$$

$$\rightarrow \left\{ \begin{array}{ll} .80 : & \neg \mathit{on}(Y,Z) \\ .10 : & \neg \mathit{on}(X,Y) \\ .10 : & \neg \mathit{on}(X,Y), \mathit{on}(Y,T), \neg \mathit{on}(Y,Z) \end{array} \right.$$

This rule applies when the empty gripper is asked to pick up a small block *X* that sits on top of another block *Y*. The gripper grabs both with a high probability.

$$\mathit{puton}(X) : \left\{ \begin{array}{l} Y : \mathit{topstack}(Y,X),\ Z : \mathit{inhand}(Z), \\ T : \mathit{table}(T) \end{array} \right\}$$

$$\mathit{size}(Y) < 2$$

$$\rightarrow \left\{ \begin{array}{ll} .62 : & \mathit{on}(Z,Y) \\ .12 : & \mathit{on}(Z,T) \\ .04 : & \mathit{on}(Z,T), \mathit{on}(Y,T), \neg \mathit{on}(Y,X) \\ .22 : & \text{noise} \end{array} \right.$$

This rule applies when the gripper is asked to put its contents, *Z*, on a block *X* which is inside a stack topped by a small block *Y*. Because placing things on a small block is chancy, there is a reasonable probability that *Z* will fall to the table, and a small probability that *Y* will follow.

## 5 Discussion and Future Work

In this paper, we developed a probabilistic action model representation that is rich enough to be used to learn models for planning in the simulated blocks world. This is a first step towards defining representations and algorithms that will enable learning in more complex worlds.

There remains much work to be done in the context of learning probabilistic planning rules. We plan to expand our approach to handle partial observability, possibly incorporating some of the techniques from work on deterministic learning [Amir, 2005]. We also plan to learn probabilistic operators in an incremental, online manner, similar to the learning setup in the deterministic case [Shen & Simon, 1989; Gil, 1994; Wang, 1995], which has the potential to help scale this approach to larger domains. Finally, we plan to explore the learning of parallel planning rules.

## 6 Acknowledgments

This material is based upon work supported in part by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010; and in part by DARPA Grant No. HR0011-04-1-0012 .

## References

[Amir, 2005] Amir, E. 2005. Learning partially observable deterministic action models. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*.

[Benson, 1996] Benson, S. 1996. *Learning Action Models for Reactive Autonomous Agents*. Ph.D. Dissertation, Stanford University.

[Blum & Langford, 1999] Blum, A., and Langford, J. 1999. Probabilistic planning in the graphplan framework. In *Proceedings of the Fifth European Conference on Planning*.

[Edelkamp & Hoffman, 2004] Edelkamp, S., and Hoffman, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. *Technical Report 195, Albert-Ludwigs-Universität, Freiburg, Germany*.

[Fikes & Nilsson, 1971] Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(2).

[Gardiol & Kaelbling, 2003] Gardiol, N., and Kaelbling, L. 2003. Envelope-based planning in relational MDPs. In *Advances in Neural Information Processing Systems 16*.

[Gil, 1994] Gil, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning*.

[Kearns, Mansour, & Ng, 2002] Kearns, M.; Mansour, Y.; and Ng, A. 2002. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning* 49(2).

[Lavrač & Džeroski, 1994] Lavrač, N., and Džeroski, S. 1994. *Inductive Logic Programming Techniques and Applications*. Ellis Horwood.

[ODE, 2004] ODE. 2004. Open dynamics engine toolkit. http://opende.sourceforge.net.

[Pasula, Zettlemoyer, & Kaelbling, 2004] Pasula, H.; Zettlemoyer, L.; and Kaelbling, L. 2004. Learning probabilistic relational planning rules. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*.

[Puterman, 1999] Puterman, M. L. 1999. *Markov Decision Processes*. John Wiley and Sons, New York.

[Shen & Simon, 1989] Shen, W.-M., and Simon, H. A. 1989. Rule creation and rule learning through environmental exploration. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*.

[Wang, 1995] Wang, X. 1995. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on Machine Learning*.

# Adaptive Partitioning of State Spaces using Decision Graphs for Real-Time Modeling and Planning

**Mykel J. Kochenderfer** and **Gillian Hayes**
Institute of Perception, Action and Behaviour
School of Informatics, University of Edinburgh
Edinburgh, United Kingdom EH9 3JZ
m.kochenderfer@ed.ac.uk, gmh@inf.ed.ac.uk

## Abstract

This paper introduces a framework that integrates incremental model learning and reactive plan construction for real-time control of an agent situated in a stochastic world. This system is designed to achieve competent performance in continuous-time problems involving large or infinite state spaces. In order to learn and plan in such domains effectively, experience must be generalized over time and space. This system achieves generalization by incrementally adapting a discretization of the state space using a decision graph as the agent gains experience and refines its plan. In contrast with other approaches, the framework presented in this paper makes no assumption about the representation of the state space, making it broadly applicable across relational and continuous-valued domains. The mechanism of this framework is illustrated on an artificial problem.

## 1 Introduction

This paper presents a new system that integrates incremental model learning and reactive plan construction for real-time control of an agent situated in a stochastic world. This system is a specialized instance of our more general "Adaptive Modeling and Planning System" (AMPS). Under consideration in this paper are problems with large or infinite state spaces and durative actions that operate in continuous time. The agent has no prior knowledge of the dynamics of the world or its task. The objective of the agent is to use its experience in the world to competently maximize its expected discounted reward.

In order to make problems with large or infinite state spaces tractable, a *decision graph* partitions the state space into regions and treats all states in the same region collectively. A decision graph is a generalization of a decision tree where nodes may have multiple parents. Associated with every internal node is a condition, which is a binary test that maps states to truth values. The leaf nodes represent regions of the state space. A state is mapped to a region through a series of tests starting at the root and continuing through the graph according to the results of these tests until arriving at a leaf.

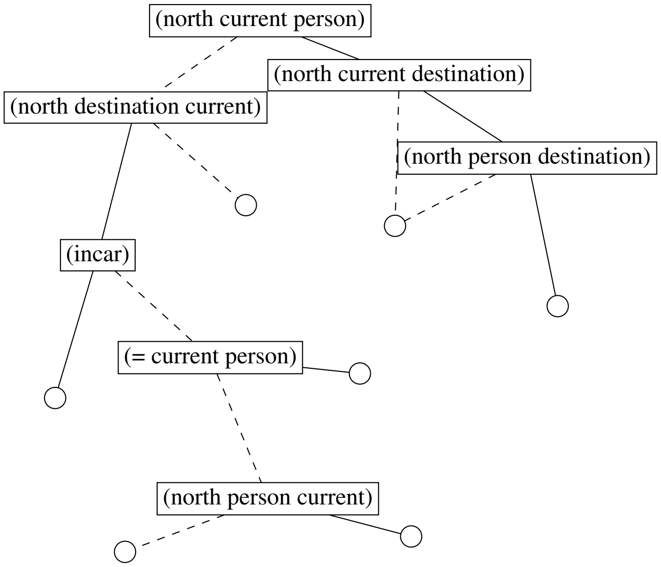

(north current person)
(north current destination)
(north destination current)
(north person destination)
(incar)
(= current person)
(north person current)

Figure 1: A decision graph constructed by AMPS during early exploration of the Taxi World domain. Decision nodes are represented by rectangles and leaf nodes, corresponding to regions of the state space, are represented by circles. When the test at a decision node evaluates to true, the solid edge is followed. Otherwise, the dashed edge is followed.

Figure 1 is an example of a decision graph that was incrementally constructed by AMPS.

AMPS begins with a decision graph consisting of a single node representing the entire state space. As the agent accumulates experience, AMPS incrementally refines its partition of the state space by splitting and merging regions, thereby growing the decision graph, aiming to arrive at a model that is consistent with its experience and conducive to building a successful reactive plan.

The primary contribution of this paper is a method for incrementally adapting the partition of the state space by splitting and merging regions. One of the methods for splitting regions in AMPS is inspired in part by the work of Uther and Veloso [1998] where regions of the state space are split when there are observed differences in expected discounted reward

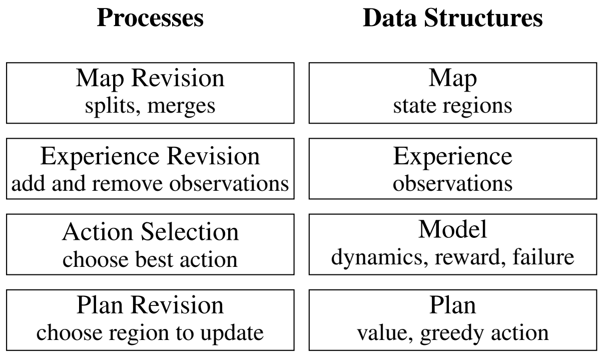


Figure 2: A high-level view of AMPS showing the four processes and the four data structures.

for different states within the same region. Another method for splitting regions in AMPS is inspired by the PARTI-GAME algorithm [Moore and Atkeson, 1995]. In PARTI-GAME and AMPS, the agent detects when it has become "stuck," meaning that the repeated application of the same action is not likely to achieve success in transitioning out of the current region. Both PARTI-GAME and AMPS use this information to refine the discretization of the state space. In AMPS this is known as failure revision. In addition to splitting regions, AMPS merges regions when possible, which is not typically done in other discretization algorithms.

AMPS is composed of four processes and four data structures as illustrated in Figure 2. The Map Revision process splits and merges regions of the state space. The Experience Revision process maintains past experience. The Action Selection process makes control decisions. The Plan Revision process incrementally improves the plan. These processes read from and modify the Map, Experience, Model, and Plan data structures.

The next section defines the problem under investigation, followed by a description of the data structures and processes involved in AMPS. We then present our experiments and results. We conclude with comparisons to related work and a discussion of further research.

## 2 Problem Statement

The class of problems considered in this research involves an agent situated in a world. The agent continuously (or at high frequency) observes and acts in this world. At any point in time, the agent is in exactly one of potentially infinitely many states and the agent may execute exactly one of a small finite number of possible actions. Actions may be *durative* meaning that they may be interrupted at any time (e.g. "rotate left at 1 rad/s") or *ballistic* meaning that they return control to the agent after some amount of time (e.g. "rotate left 1 rad").

The experience of the agent may be recorded as a sequence

$$s_1, a_1, t_1, f_1, r_1, s_2, a_2, t_2, f_2, r_2, \ldots$$

where $s_k$ is the state at the $k$th sample, $a_k$ is the action taken after the $k$th sample, $t_k$ is the time spent between sample $k$ and $k+1$, $f_k$ indicates whether a failure was encountered, and $r_k$ is the reward received after the $k$th sample. The objective of the agent after sample $k$ is to maximize its expected discounted reward,

$$E\left[\sum_{i=0}^{\infty} e^{-\beta\sigma_{k+i}} r_{k+i}\right]$$

where $\sigma_k = \sum_{i=1}^{k} t_k$ and $\beta \in (0, \infty)$ is the continuous compound discount rate. Discounting the reward pressures the agent to aggressively pursue reward.

AMPS is expected to perform well on problems that are modeled sufficiently well by a semi-Markov decision process [Puterman, 1994] over a state space that has been partitioned into a finite set of regions $N$. Transitions, reward, and failures are determined by fixed probability distributions:

- $P(\cdot|n, a)$ is the probability distribution over the regions transitioned to after starting in $n$ and continuously applying $a$.
- $P_t(\cdot|n, a, n')$ is the c.d.f. for the time required to transition from $n$ to $n'$ by continuously applying $a$.
- $P_r(\cdot|n, a, n')$ is the c.d.f. for the lump sum reward received after transitioning from $n$ to $n'$ by continuously applying $a$.[1]

If $N$ and these probability distributions are known, then the optimal value function, $V^*$, and optimal reactive plan, $\pi^*$, satisfy:

$$V^*(n) = \max_a Q(n, a, n') \qquad (1)$$

$$\pi^*(n) = \arg\max_a Q(n, a, n') \qquad (2)$$

where

$$\gamma(n, a, n') \triangleq \int_0^{\infty} e^{-\beta t} dP_t(t|n, a, n')$$

$$R(n, a, n') \triangleq P(n'|n, a)\gamma(n, a, n') \int_{-\infty}^{\infty} r dP_r(r|n, a, n')$$

$$Q(n, a, n') \triangleq R(n, a, n') + \sum_{n'} P(n'|n, a)\gamma(n, a, n')V^*(n)$$

The value function $V^*(n)$ is the expected discounted reward given that the agent starts in region $n$ and follows an optimal policy. The expected discounted reward given that the agent starts with a transition from $n$ to $n'$ by action $a$ and then follows an optimal policy is given by $Q(n, a, n')/P(n'|n, a)$, a value that will be used in the map revision process described in Section 4.1.

The agent has no prior knowledge of $N$, $P$, $P_t$, or $P_r$. AMPS incrementally adapts its partitioning of the state space and revises its estimates of $P$, $\gamma$, and $R$ accordingly. As these estimates are revised, AMPS uses prioritized value iteration to improve its reactive plan.

[1] Other SMDP models considered in the literature [Bradtke and Duff, 1995] involve reward that is accumulated at some constant rate as opposed to lump sum rewards following transitions. In this paper we will only consider lump sum rewards, although AMPS can easily be extended to handle reward rate models.

# 3 Data Structures

The data structures in AMPS store information about state regions, observations, world models, and plans. This section will describe each data structure in turn.

## 3.1 Map

The *Map* data structure maps states to regions and allows these regions to be efficiently split and merged to adapt with the experience of the agent. The instance of AMPS considered in this paper performs these operations using a decision graph.[2]

Merging regions is straightforward in a decision graph, but splitting regions requires a mechanism called a *separator*. The separator mechanism creates binary tests based on sample states belonging to different categories, much like a supervised learning classifier [Duda and Hart, 1973]. The tests that are generated by the separator should be simple. A test should not be, for example, an entire decision tree, a neural network, or a lengthy sentence in first order logic. Instead, simple tests such as $x_2 < 2.3$ or $\exists x\ \textsc{on}(\textsc{a}, x)$ should be used. Simple tests are to be combined using the decision graph.

Clearly, the separator mechanism must be engineered according to how the state space is represented. It is important to note that the separator is the only component in AMPS that interacts with the state space representation directly. This is intentional. AMPS is designed to be representation independent, unlike most other partitioning algorithms,[3] enabling it to be applied across both relational and continuous-valued domains with only the separator requiring changing.

We implemented a program that generates separators for any logical domain given the names of the constants, functions, and relations and their associated types. We integrated the JTP theorem prover [Fikes *et al.*, 2003] to prune unnecessary logical sentences from consideration based on a set of axioms about the domain. The separator selects the sentence that provides the greatest information gain [Shannon, 1948].

## 3.2 Experience

The *Experience* data structure keeps a record of the states and transitions experienced by the agent and associates them with the regions, i.e. the leaf nodes of the decision graph, managed by the Map. When regions are split and merged, the association of past observations to regions is updated appropriately.

## 3.3 Model

The *Model* data structure models state transition dynamics, failure, and reward. Let $P_f(n, a)$ be the probability of encountering failure when executing action $a$ from region $n$. This may be estimated directly from experience as can $P(n'|n, a)$. Instead of estimating the c.d.f.s $P_t$ and $P_r$ directly, AMPS estimates $\gamma$ and $R$ instead. The integrals in $\gamma$ and $R$ are approximated as follows (see Kalos and Whitlock 1986, pp. 89–116):

- $\int_0^\infty e^{-\beta t} dP_t(t|n, a, n')$ is approximated by averaging $e^{-\beta \tau_i}$ where $\tau_i$ is the time required to make the $i$th transition from $n$ to $n'$ by action $a$.
- $\int_{-\infty}^\infty r dP_r(r|n, a, n')$ is approximated by averaging the reward received while transitioning from $n$ to $n'$ by action $a$.

These approximations are used by the Plan data structure, which is described next.

## 3.4 Plan

The *Plan* data structure maintains the value function and the greedy policy. The estimated value function, $V(n)$, is an estimate of the expected discounted reward starting in region $n$ and proceeding with an estimate of optimal behavior. The estimated greedy policy, $\pi(n)$, associates with each state region an estimate of the greedy action that maximizes the expected discounted reward.

As can be seen in the equations given in Section 2, the value and the greedy action associated with each state region depend upon the value of other state regions. The Plan data structure supports a function called $\textsc{Update}(n)$ that updates the value and greedy action of region $n$ using Equation 1 assuming that the values of all other regions are correct. The Plan Revision process is responsible for calling UPDATE on state regions in response to changes in the Model and Plan.

# 4 Processes

The four processes in AMPS are responsible for splitting and merging regions of the state space, recording experience, selecting actions, and constructing plans. All processes are designed to be incremental and perform (relatively) simple operations on the data structures at a frequency dependent on available computational resources.

## 4.1 Map Revision

The Map Revision process is responsible for dynamically splitting and merging state regions in response to changes in the Model and Plan. If the agent has no prior knowledge of how the state and action space should be partitioned when it commences its interaction with the world, the entire state space is contained within a single region. Over time, this region is incrementally refined and simplified as experience is acquired, growing the decision graph. The objective is to find a partition of the state space that has the following properties:

1. All transitions resulting from a greedy action have approximately the same estimated value.
2. The expected failure of greedy actions is minimal.
3. The partition is as simple as possible.

Different types of revision can be done to bring the Map closer to the criteria listed above. Each type of revision is given a priority at each region that indicates (heuristically) the

[2] Alternatively, nearest-neighbor [Cover and Hart, 1967] may be used provided that a distance metric is defined over the state space.

[3] Other partitioning algorithms in the literature make strong assumptions about how states and actions are represented. For example, Chapman and Kaelbling [1991] and Munos and Patinel [1994] assume that states are represented as fixed-length binary strings, McCallum [1995] assumes an attribute-value representation, Dean *et al.* [1998] assume a factored state and action representation, and Mahadevan and Connell [1992] assume that states are represented as real-valued vectors.

likelihood that performing that particular type of revision will improve the Map. When the Map Revision process runs, it will perform the highest priority revision. The Map Revision process ignores revisions below a certain threshold so as to not over-fit potentially noisy experience.

The three types of revision used in this system correspond to the three criteria above, and they are as follows:

**Value Revision**

This type of revision attempts to separate state observations that have resulted in transitions with differing estimated value. This separation is done with the separator mechanism for the decision graph. The priority of performing this type of revision is proportional to a measure of variation of estimated value. The experimental implementation separates states involved in greedy transitions based on whether $Q(n, \pi(n), n')/P(n'|n, \pi(n))$ is above or below the mean. The priority is proportional to the variance of $Q(n, \pi(n), n')/P(n'|n, \pi(n))$.

**Failure Revision**

This type of revision uses the separator mechanism to separate states that have led to success from those that have led to failure. The priority of this type of revision for a state region is related to the estimated probability of failure in the Model when taking a greedy action, which is given by $P_f(n, \pi(n))$.

**Simplification Revision**

This type of revision merges state regions when their distinction seems to be irrelevant to the task. Non-greedy action regions are merged with low priority. State regions are merged in two situations. The first situation occurs when there exists an approximately deterministic greedy transition[4] from $n$ to $n'$ and both nodes have the same greedy action, in which case $n$ and $n'$ are merged. The second situation occurs when there exists approximately deterministic greedy transitions from $n$ and $n'$ leading to the same node and $\pi(n) = \pi(n')$, in which case $n$ and $n'$ may be merged. These conditions for merging are related to the SQUISH algorithm described by Nilsson [2000] for deterministic teleo-reactive trees.

## 4.2 Experience Revision

The *Experience Revision* process adds state and transition observations to the Experience data structure. If the state space is continuous and it is being sampled at a high frequency, it is impractical to add each sample. Instead, the agent should filter out samples that are not likely to be useful. The exact mechanism for determining significance is domain dependent, but one approach to filtering out states is to ignore all states until the agent has transitioned to a new state that is outside some threshold distance.

In addition to filtering out insignificant samples, the Experience Revision process is also responsible for removing old samples from the Experience data structure. The schedule for removing old samples depends upon memory and processor constraints. The removal of old samples also allows the agent to adapt to slowly changing environments.

## 4.3 Action Selection

The *Action Selection* process is responsible for continuously selecting a single action to execute. This process must be extremely efficient because it is typically executed as frequently as the agent samples the state of the world.

Usually, the agent should execute an action from the greedy region of the current state region as computed by the policy revision process. However, as with traditional reinforcement learning there are issues with balancing exploration of the world and exploitation of the optimal policy. There have been many techniques proposed for balancing exploration with exploitation [Thrun, 1992], any of which may be used in this system.

As mentioned earlier, the agent is able to sense failures when they occur. What exactly counts as a failure depends upon the domain, but failures are generally situations where executing the same action repeatedly will not lead to success. In the Taxi World domain described later, a failure might be trying to drop off a person who is not in the taxi. The Action Selection mechanism uses information about past failures to better direct the agent toward a goal.

In our current implementation of AMPS, the agent will take the greedy action $\pi(n)$ with probability $1 - P_f(n, \pi(n)) - \epsilon$, where $\epsilon$ is a small positive value allowing random exploration even when no failure has been observed. If $\pi(n)$ is not taken, then the last action $a$ is followed with probability $1 - P_f(n, a) - \epsilon$. Otherwise, a random action is selected.

## 4.4 Plan Revision

The *Plan Revision* process incrementally builds an optimal plan with the assumption that the current Model is correct. This process calls the UPDATE($n$) function supplied by the Plan data structure, which updates the value and greedy action for $n$.

One way to arrive at an optimal policy is to repeatedly iterate through the state regions calling UPDATE until convergence [cf. Bellman 1957]. However, doing so is not likely to be feasible in real time since the Model is continually changing. Instead, it is better to use a method related to prioritized sweeping [Moore and Atkeson, 1993], which was designed for solving Markov decision processes (MDPs) but can be adapted for solving SMDPs. The idea of the algorithm is to prioritize updates of regions based on observed changes in the value function from earlier updates. Pseudo-code is given in Figure 3.

# 5 Experiments and Results

This section demonstrates AMPS in the Taxi World domain. In this problem the agent must control a car in a grid world to pick up people and transport them to their desired destination. The agent may move up, down, left, and right and pick up and put down passengers. When the agent moves around in the world, it will move in a direction orthogonal to the direction it intended with 5% probability.

The state space is represented by a seven-dimensional vector containing the $x$ and $y$ coordinates of the car, person, and desired destination and a Boolean value indicating whether

[4] An approximately deterministic greedy transition from $n$ to $n'$ is one where $P(n'|n, \pi(n)) \approx 1$.

```
PRIORITIZED-SWEEPING
while priority queue is not empty
    do remove region $n$ from the front of the queue
        $v \leftarrow V(n)$
        UPDATE$(n)$
        $\Delta \leftarrow |V(n) - v|$
        for each pair $\langle n', a' \rangle$
              s.t. $n' \neq n$ and $P(n|n', a') > 0$
            do $p \leftarrow P(n|n', a')\gamma(n', a', n)\Delta$
                if $p > \epsilon$ and $n'$ is either not in the queue
                      or $p$ is greater than its current priority
                    then promote the priority of $n'$ to $p$
```

Figure 3: The adapted version of prioritized sweeping used by the Plan Revision process.

the person is in or out of the car. The separator partitions the state space using the IN-CAR relation and various directional relations (e.g. NORTH and DIRECTLY-WEST) that take two objects as arguments, where the objects may be CURRENT, PERSON, and DESTINATION, representing the current position of the taxi, the person, and the destination respectively. Failures occur when the agent attempts to pick up or put down a person when not occupying the same square or when it attempts to move past the border of the world. Goal states are states where IN-CAR is true and the person is at its destination. The agent receives zero reward except at goal states where it receives unit reward. The continuous compound discount rate, $\beta$, was set to 0.01.

In our experiments we used a $40 \times 40$ grid, which allows for over 4 billion states. Since the agent has no prior knowledge about the dynamics of the world or its task, it considers all states to be equivalent. Hence, the agent must rely upon random exploration until it reaches a goal. Random exploration is not practical in the Taxi World or most other interesting domains because the state space is so large, and therefore the agent needs some mechanism to bias it toward the relevant goal states [Whitehead, 1991]. In our experiments, we used a teacher to guide the agent to a goal for two training episodes. These two training episodes allowed AMPS to partition the state space and distribute the observed reward through the model. The teacher used in the experiments returned random actions 5% of the time.

For comparison purposes and to control for the information gained from the noisy teacher, we trained a decision tree based on the same teacher. The decision tree is induced using the separator to partition the state space according to the actions taken by the teacher. This sort of supervised learning of control is known as *behavioral cloning* and has been successfully applied to a variety of domains including aircraft control [Sammut *et al.*, 1992]. The primary disadvantages of a behavioral clone is that it is entirely dependent on a good teacher and it is not able to adapt its behavior based on its own experience in contrast to AMPS.

In our experiments, we tested seven different agents: AMPS with all forms of map revision (A), AMPS without value revision (V), AMPS without failure revision (F), AMPS without simplification revision (S), behavioral clone (C), the noisy teacher that was used to train the other agents (T), and random (R). We ran 100 runs of 20 episodes (not including the two training episodes) each using different random seeds. For the AMPS agents, no more than one merge or split of the state space was allowed per time step. Each episode lasts 300 steps or when the agent successfully picks up and drops off its passenger at the desired destination. The results of 100 repeated experiments are summarized in Table 1.

| | Successes | | Disc. Reward | | Regions | |
|---|---|---|---|---|---|---|
| | mean | s.d. | mean | s.d. | mean | s.d. |
| A | 14.49 | 4.27 | 6.05 | 2.20 | 66.73 | 11.78 |
| V | 0.97 | 2.01 | 0.31 | 0.99 | 18.48 | 8.28 |
| F | 0.02 | 0.14 | 0.02 | 0.14 | 1.00 | 0.00 |
| S | 14.33 | 3.81 | 6.11 | 2.02 | 84.66 | 13.12 |
| C | 3.49 | 3.24 | 1.48 | 1.82 | 12.39 | 2.11 |
| T | 20.00 | 0.00 | 10.29 | 0.69 | N/A | N/A |
| R | 0.02 | 0.14 | 0.02 | 0.14 | N/A | N/A |

Table 1: Experimental results of various agents in the Taxi World domain. Shown are the means and standard deviations of the number of successes and discounted reward over 100 runs of 20 episodes each. Shown for each AMPS-based agent is the mean and standard deviation of the number of regions at the end of the run.

## 6 Discussion

As can be seen in the table of results, AMPS quickly learned how to navigate from a state selected randomly from the space of over 4 billion possible states to one of the 160 goal states. The success of AMPS is due to its ability to take advantage of the structure inherent in the task.

AMPS began with a simple model of the world where all states are equivalent. After two teaching episodes, AMPS proceeded on its own with the reactive plan that it developed. As AMPS encountered evidence indicating that its model and plan was incorrect, the Map Revision process revised the state space partition appropriately. The complexity of the agent's model increased rapidly as the agent initially explored the world but then this complexity leveled off. The number of regions commonly used in effective solutions to the Taxi World problem was typically in the range of 60–80. Figure 6 shows the connectivity between regions after five episodes.

The results demonstrate that AMPS without value revision performs extremely poorly and AMPS without failure revision performs like a random controller. As can be expected, AMPS without simplification revision is capable of quickly learning effective behavior. However, without simplification revision the model consists of about 27% more regions, and consequently requires more memory and processing power.

Behavioral clones did quite poorly, solving on average only 3.49 of the 20 random problems. This poor performance is due to the fact that behavioral clones do not adapt their policies in response to their experience. Because the two training instances provided such a limited exposure to the state space, the clones could not consistently generalize the training instances to new situations.

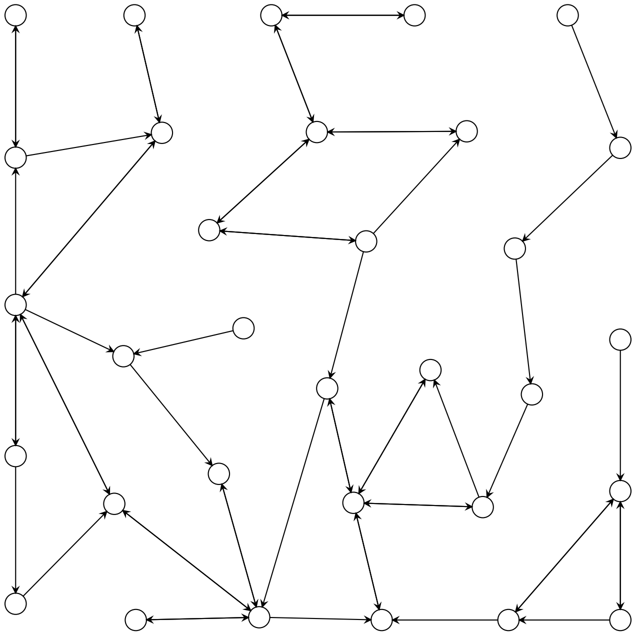


Figure 4: A graph illustrating the connectivity between regions after running AMPS for five episodes.

Although AMPS is unique in the way it decides how and when to split and merge regions of the state space, there have been other algorithms proposed, both model-free and model-based, that use decision trees to partition the state space [Chapman and Kaelbling, 1991; McCallum, 1995; Uther and Veloso, 2003; 1998]. Other techniques have been proposed for solving problems with large or infinite state spaces that do not involve partitioning the state space into regions. For example, parameterized function approximators, such as neural networks, may be used with traditional reinforcement learning techniques to estimate the value function [Bertsekas and Tsitsiklis, 1996]. Other work has been done on problems with durative actions [Benson and Nilsson, 1995]. The way AMPS uses SMDPs to model durative actions relates to work done in hierarchical reinforcement learning [surveyed in Barto and Mahadevan 2003].

## 7 Conclusions and Further Work

This paper has introduced a real-time system that integrates adaptive partitioning of the state space and incremental planning over the estimated model to produce goal-directed behavior. The system was tested on the Taxi World domain and was shown to quickly learn successful behavior in the presence of noise in the environment and the teacher.

AMPS is currently being tested in other more complex domains and compared against other methods such as traditional reinforcement learning with function approximation. Further work will investigate the use of different separators that take into account values of previous observations, making AMPS potentially applicable to domains where the current state is only partially observable.

## Acknowledgments

The authors would like to thank Nils Nilsson, George Konidaris, Marc Toussaint, and the anonymous reviewers for their helpful suggestions on earlier drafts of this paper.

## References

[Barto and Mahadevan, 2003] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, October 2003.

[Bellman, 1957] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Benson and Nilsson, 1995] Scott Benson and Nils J. Nilsson. Reacting, planning and learning in an autonomous agent. In K. Furukawa, D. Michie, and S. Muggleton, editors, *Machine Intelligence*, volume 14, pages 29–64. Oxford University Press, 1995.

[Bertsekas and Tsitsiklis, 1996] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[Bradtke and Duff, 1995] Steven J. Bradtke and Michael O. Duff. Reinforcement learning methods for continuous-time Markov decision problems. In Gerald Tesauro, David S. Touretzky, and Todd K. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 393–400. MIT Press, 1995.

[Chapman and Kaelbling, 1991] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 726–731. Morgan Kaufmann, 1991.

[Cover and Hart, 1967] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, January 1967.

[Dean *et al.*, 1998] Thomas Dean, Robert Givan, and Kee-Eung Kim. Solving planning problems with large state and action spaces. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 102–110. AAAI Press, 1998.

[Duda and Hart, 1973] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973.

[Fikes *et al.*, 2003] Richard Fikes, Jessica Jenkins, and Gleb Frank. JTP: A system architecture and component library for hybrid reasoning. Technical Report KSL-03-01, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, 2003.

[Kalos and Whitlock, 1986] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo Methods*, volume 1. John Wiley and Sons, 1986.

[Mahadevan and Connell, 1992] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55(2–3):311–365, June 1992.

[McCallum, 1995] Andrew Kachites McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, Department of Computer Science, University of Rochester, 1995.

[Moore and Atkeson, 1993] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, October 1993.

[Moore and Atkeson, 1995] Andrew W. Moore and Christopher G. Atkeson. The Parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21(3):199–233, December 1995.

[Munos and Patinel, 1994] Rémi Munos and Jocelyn Patinel. Reinforcement learning with dynamic covering of state-action space: Partitioning Q-learning. In *Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 354–363. MIT Press, 1994.

[Nilsson, 2000] Nils J. Nilsson. Learning strategies for mid-level robot control: Some preliminary considerations and results. www.robotics.stanford.edu/users/nilsson/trweb, 2000. Computer Science Department, Stanford University.

[Puterman, 1994] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[Sammut *et al.*, 1992] Claude Sammut, Scott Hurst, Dana Kedzier, and Donald Michie. Learning to fly. In *Proceedings of the Ninth International Workshop on Machine Learning*, pages 385–393. Morgan Kaufmann, 1992.

[Shannon, 1948] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[Thrun, 1992] Sebastian B. Thrun. The role of exploration in learning control. In D. White and D. Sofge, editors, *Handbook for Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pages 527–559. Van Nostrand Reinhold, 1992.

[Uther and Veloso, 1998] William Uther and Manuela Veloso. Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 769–775. AAAI Press, 1998.

[Uther and Veloso, 2003] William Uther and Manuela Veloso. TTree: Tree-based state generalization with temporally abstract actions. In *Adaptive Agents and Multi-Agent Systems: Adaptation and Multi-Agent Learning*, volume 2636 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2003.

[Whitehead, 1991] Steven D. Whitehead. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 607–613. AAAI Press, 1991.

# Modeling and Planning in Large State and Action Spaces

**Mykel J. Kochenderfer** and **Gillian Hayes**
Institute of Perception, Action and Behaviour
School of Informatics, University of Edinburgh
Edinburgh, United Kingdom EH9 3JZ
m.kochenderfer@ed.ac.uk, gmh@inf.ed.ac.uk

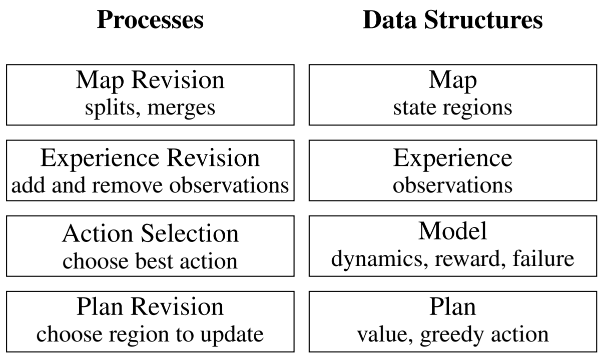
## Abstract

This paper introduces a general framework that integrates incremental model learning and reactive plan construction for real-time control of an agent situated in a stochastic world. This system is designed to achieve competent performance in continuous-time problems involving large or infinite state and action spaces. In order to learn and plan in such domains effectively, experience must be generalized over time, state, and action. This system achieves generalization by incrementally adapting a discretization of the state and action spaces as the agent gains experience and refines its plan. In contrast with other approaches, the framework presented in this paper makes no assumption about the representation of the state space, making it broadly applicable across relational and continuous-valued domains. The mechanism of this framework is illustrated on an artificial problem.

## 1 Introduction

This paper presents the "Adaptive Modeling and Planning System" (AMPS), a general framework for integrating incremental model learning and reactive plan construction for real-time control of an agent situated in a stochastic world. Under consideration are problems with large or infinite state and action spaces where individual actions operate in continuous time. The agent has no prior knowledge of the dynamics of the world or its task. The objective of the agent is to use the experience it gains through interacting with the world to maximize its expected discounted reward.

In order to make problems with large or infinite state spaces tractable, AMPS partitions the state space into regions and treats all states in the same region collectively. Since the action space is also large or infinite, AMPS also partitions the action space into regions and treats all actions from the same region identically. AMPS incrementally refines its partition of the state and action spaces by splitting and merging regions of the state and action spaces to arrive at a model that is consistent with its experience and conducive to building a successful reactive plan.



| Processes | Data Structures |
|---|---|
| Map Revision<br>splits, merges | Map<br>state regions |
| Experience Revision<br>add and remove observations | Experience<br>observations |
| Action Selection<br>choose best action | Model<br>dynamics, reward, failure |
| Plan Revision<br>choose region to update | Plan<br>value, greedy action |

Figure 1: A high-level view of AMPS showing the four processes and the four data structures.

The primary contribution of this paper is a method for incrementally adapting the partitions of the state and action spaces by splitting and merging regions. One of the methods for splitting regions in AMPS is inspired by the work of Uther and Veloso [1998] where regions of the state space are split when there are observed differences in utility for different states within the same region. Another method for splitting regions in AMPS is inspired by the PARTI-GAME algorithm [Moore and Atkeson, 1995]. In PARTI-GAME and AMPS, the agent detects when it has become "stuck," meaning that the repeated application of the same action is not likely to achieve success in transitioning out of the current region. Both PARTI-GAME and AMPS use this information to refine their discretization. In AMPS this is known as failure revision. In addition to splitting regions, AMPS merges regions when possible, which is not typically done in other discretization algorithms.

AMPS is composed of four processes and four data structures as illustrated in Figure 1. The Map Revision process splits and merges regions of the state and action spaces. The Experience Revision process maintains past experience. The Action Selection process makes control decisions. The Plan Revision process incrementally improves the plan. These processes read from and modify the Map, Experience, Model, and Plan data structures.

The next section defines the problem under investigation, followed by a description of the data structures and processes involved in AMPS. We then present our experiments and results. We conclude with comparisons to related work and a discussion of further research.

## 2 Problem Statement

The class of problems considered in this research involves an agent situated in a world. The agent continuously (or at high frequency) observes and acts in this world. At any point in time, the agent is in exactly one state belonging to a potentially infinite set $\mathcal{S}$ and the agent may execute exactly one action from a potentially infinite set $\mathcal{A}$. Actions may be *durative* meaning that they may be interrupted at any time (e.g. "rotate left at 1 rad/s") or *ballistic* meaning that they return control to the agent after some amount of time (e.g. "rotate left 1 rad").

The experience of the agent may be recorded as a sequence

$$s_1, a_1, t_1, f_1, r_1, s_2, a_2, t_2, f_2, r_2, \ldots$$

where $s_k$ is the state at the $k$th sample, $a_k$ is the action taken after the $k$th sample, $t_k$ is the time spent between sample $k$ and $k+1$, $f_k$ indicates whether a failure was encountered after the $k$th sample, and $r_k$ is the reward received after the $k$th sample. The objective of the agent after sample $k$ is to maximize its expected discounted reward,

$$E\left[\sum_{k=0}^{\infty} e^{-\beta\sigma_{k+i}} r_{k+i}\right]$$

where $\sigma_k = \sum_{i=1}^{k} t_k$ and $\beta \in (0, \infty)$ is the continuous compound discount rate. Discounting the reward pressures the agent to aggressively pursue reward.

AMPS is expected to perform well on problems that are modeled sufficiently well by a semi-Markov decision process [Puterman, 1994]. It is assumed that $\mathcal{S}$ may be partitioned into a finite set of regions $N$ and for each region $n \in N$ there exists a partition of $\mathcal{A}$ into regions $U(n)$ such that the transitions, reward, and failures are determined by fixed probability distributions:

- $P(\cdot|n, u)$ is the probability distribution over the regions transitioned to after starting in $n$ and continuously applying actions in $u$.
- $P_t(\cdot|n, u, n')$ is the c.d.f. for the time required to transition from $n$ to $n'$ by continuously applying actions in $u$.
- $P_r(\cdot|n, u, n')$ is the c.d.f. for the lump sum reward received after transitioning from $n$ to $n'$ by continuously applying actions in $u$.[1]

If $N$, $U$, and these probability distributions are known, then the optimal value function, $V^*$, and optimal reactive plan, $\pi^*$, satisfy:

$$V^*(n) = \max_u Q(n, u, n') \tag{1}$$

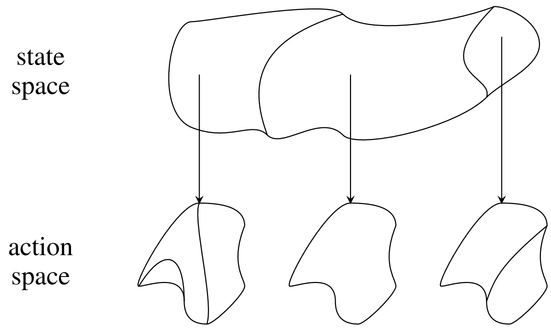


Figure 2: Associated with each region of the state space is a different partition of the action space.

$$\pi^*(n) = \arg\max_a Q(n, u, n') \tag{2}$$

where

$$\gamma(n, u, n') \triangleq \int_0^{\infty} e^{-\beta t} dP_t(t|n, u, n')$$

$$R(n, u, n') \triangleq P(n'|n, u)\gamma(n, u, n') \int_{-\infty}^{\infty} r dP_r(r|n, u, n')$$

$$Q(n, u, n') \triangleq R(n, u, n') + \sum_{n'} P(n'|n, u)\gamma(n, u, n')V^*(n)$$

The value function $V^*(n)$ is the expected discounted reward given that the agent starts in region $n$ and follows an optimal policy. The expected discounted reward given that the agent starts with a transition from $n$ to $n'$ by actions in $u$ and then follows an optimal policy is given by $Q(n, u, n')/P(n'|n, u)$, a value that will be used in the map revision process described in Section 4.1.

The agent has no prior knowledge of $N$, $U$, $P$, $P_t$, or $P_r$. AMPS incrementally adapts its partitioning of the state and action spaces and revises its estimates of $P$, $\gamma$, and $R$ accordingly. As these estimates are revised, AMPS uses prioritized value iteration to improve its reactive plan.

## 3 Data Structures

The four data structures in AMPS store information about state and action regions, observations, world models, and plans. These data structures are manipulated by the processes described in the next section, and changes in one data structure can result in changes in another data structure. Of the four data structures, the Map is the most significant and so most of the following discussion will focus on it.

### 3.1 Map

The *Map* data structure maps states and actions to regions and adapts this mapping by splitting and merging regions. The map partitions the state space into a finite set of state regions, $N$. Associated with every state region is a finite partition of the action space, as illustrated in Figure 2. Let $U$ be the set of all action regions defined across all state regions. The computational mechanism implementing the Map should be such that mapping from $\mathcal{S} \rightarrow N$ and $N \times \mathcal{A} \rightarrow U$ is efficient.

[1] Other SMDP models considered in the literature [Bradtke and Duff, 1995] involve reward that is accumulated at some constant rate as opposed to lump sum rewards following transitions. In this paper we will only consider lump sum rewards, although AMPS can easily be extended to handle reward rate models.

The representation used by the Map must allow the Map Revision process to split and merge regions. The Map implements the following functions to support revision:

1. $\text{SPLIT}(n, S_1, \ldots, S_m)$. This function splits the state region $n$ into some number of new regions such that each set of states $S_1, \ldots, S_m \subset \mathcal{S}$ that were all mapped to $n$ become mapped as well as possible to separate state regions.
2. $\text{SPLIT}(u, A_1, \ldots, A_m)$. This function splits the action region $u$ into some number of new regions such that each set of actions $A_1, \ldots, A_m \subset \mathcal{A}$ that were all mapped to $u$ become mapped as well as possible to separate action regions.
3. $\text{MERGE}(n_1, \ldots, n_m)$. This function merges the state regions $n_1, \ldots, n_m$.
4. $\text{MERGE}(u_1, \ldots, u_m)$. This function merges the action regions $u_1, \ldots, u_m$ belonging to the same state region.

The Map can be implemented, in principle, using any representation that supports these operations. For example, if a distance metric may be defined between any two states and any two actions, then a mapping based on nearest-neighbor [Cover and Hart, 1967] may be used. There have been computationally efficient algorithms and data structures, such as vantage point trees [Yianilos, 1993], suggested for computing nearest neighbors without making any additional assumptions about the representation of the space (e.g. that the space is Euclidean).

Another representation that can be used is a decision graph. Decision graphs are extremely well suited because they allow regions to be split and merged very quickly, and this is the representation used in the experiments here.

Merging regions is straightforward in a decision graph, but splitting regions requires a mechanism called a *separator* that produces the (typically binary) tests in the decision graph. The separator produces a test based on sample states belonging to multiple categories, much like a supervised learning classifier [Duda *et al.*, 2000]. The test returned by the separator should be simple. The condition should not be, for example, an entire decision tree, a neural network, or a lengthy sentence in first order logic. Instead, simple tests such as $x_2 < 2.3$ or $\exists x\, \text{ON}(\text{A}, x)$ should be used. Simple tests are to be combined using the decision graph to create more complex boundaries between regions.

The separator function must be engineered according to how the state and action spaces are represented. It is interesting to note that the separator function is the only component in the system that interacts with the state space representation directly, enabling the system to be applied across domains with only the separator requiring changing.[2]

In the experimental domain described later, the state and action space requires a representation in the form of a vector of real values. The separator used in the experiments was designed to run in time linear with respect to the number of samples. The function returns the condition that best separates the points along some axis-parallel hyperplane according to information gain [Shannon, 1948].

It should be noted that the mechanism for partitioning the state space does not have to be the same mechanism for partitioning the action space. For example, the state space may be partitioned using nearest-neighbor, but the action space associated with each state region may be partitioned using a decision graph.

### 3.2 Experience

The *Experience* data structure keeps a record of the states and transitions experienced by the agent and associates them with the state and action regions managed by the Map. When state and action regions are split and merged, the association of past observations to regions is updated appropriately.

### 3.3 Model

The *Model* data structure models state transition dynamics, failure, and reward. Let $P_f(n, u)$ be the probability of encountering failure when executing an action in $u$ from region $n$. This may be estimated directly from experience as can $P(n'|n, u)$. Instead of estimating the c.d.f.s $P_t$ and $P_r$ directly, AMPS estimates $\gamma$ and $R$ instead. The integrals in $\gamma$ and $R$ are approximated as follows [see Kalos and Whitlock 1986, pp. 89–116]:

- $\int_0^\infty e^{-\beta t} dP_t(t|n, u, n')$ is approximated by averaging $e^{-\beta \tau_i}$ where $\tau_i$ is the time required to make the $i$th transition from $n$ to $n'$ by actions in $u$.
- $\int_{-\infty}^\infty r dP_r(r|n, u, n')$ is approximated by averaging the reward received while transitioning from $n$ to $n'$ by actions in $u$.

These approximations are used by the Plan data structure, which is described next.

### 3.4 Plan

The *Plan* data structure maintains the value function and the greedy policy. The estimated value function, $V(n)$, is an estimate of the expected discounted reward starting in region $n$ and proceeding with an estimate of optimal behavior. The estimated greedy policy, $\pi(n)$, associates with each state region an estimate of the greedy action that maximizes the expected discounted reward.

As can be seen in the equations given in Section 2, the value and the greedy action associated with each state region depend upon the value of other state regions. The Plan data structure supports a function called $\text{UPDATE}(n)$ that updates the value and greedy action of state region $n$ using Equation 1 assuming that the values of all other regions are correct. The Plan Revision process is responsible for calling UPDATE on state regions in response to changes in the Model and Plan.

[2] Other partitioning algorithms in the literature make strong assumptions about how states and actions are represented. For example, Chapman and Kaelbling [1991] and Munos and Patinel [1994] assume that states are represented as fixed-length binary strings, McCallum [1995] assumes an attribute-value representation, Dean *et al.* [1998] assume a factored state and action representation, and Mahadevan and Connell [1992] assume that states are represented as real-valued vectors.

# 4 Processes

The four processes in AMPS are responsible for splitting and merging regions of the state and action space, recording experience, selecting actions, and constructing plans. All processes are designed to be incremental and perform (relatively) simple operations on the data structures at a frequency dependent on available computational resources.

## 4.1 Map Revision

The Map Revision process is responsible for dynamically splitting and merging state and action regions in response to changes in the Model and Plan. If the agent has no prior knowledge of how the state and action space should be partitioned when it commences its interaction with the world, the entire state space is contained within a single region. Over time, this region is incrementally refined and simplified as experience is acquired. The objective is to find a partition of the state and action space that has the following properties:

1. All transitions resulting from a greedy action have approximately the same estimated value.
2. The expected failure of greedy actions is minimal.
3. The partition is as simple as possible.

Different types of revision can be done to bring the Map closer to the criteria listed above. Each type of revision is given a priority at each region that indicates (heuristically) the likelihood that performing that particular type of revision will improve the Map. When the Map Revision process runs, it will perform the highest priority revision. The Map Revision process ignores revisions below a certain threshold so as to not over-fit potentially noisy experience.

The three types of revision used in this system correspond to the three criteria above, and they are are as follows:

**Value Revision**

This type of revision attempts to separate state-action observations that have resulted in transitions with differing estimated value. This separation can be done by splitting action regions within a state region or by splitting the state region. Deciding whether to split by state or by action can be done using information gain.[3] The priority of performing this type of revision is proportional to a measure of variation of estimated value. The experimental implementation separates samples involved in greedy transitions based on whether $Q(n, \pi(n), n')/P(n'|n, u)$ is above or below the mean. The priority is proportional to the variance of $Q(n, \pi(n), n')/P(n'|n, u)$.

**Failure Revision**

This type of revision separates states that have led to success from those that have led to failure. The priority of this type of revision for a state region is related to the estimated probability of failure in the Model.

**Simplification Revision**

This type of revision merges state and action regions when their distinction seems to be irrelevant to the task. Non-greedy action regions are merged with low priority. State regions are merged in two situations. The first situation occurs when $P(n'|n, \pi(n)) \approx 1$ and $\pi(n) \approx \pi(n')$, in which case $n$ and $n'$ are merged. Since action regions are partitioned differently in different state regions, it is necessary to define the equivalence relation $u \approx u'$, which means that all the actions experienced in $u$ can be mapped to action region $u'$ in the state region of $u'$ and vice versa. The second situation when two state regions are merged is when $P(n|n', \pi(n')) \approx 1$ and $P(n|n'', \pi(n'')) \approx 1$ and $\pi(n') \approx \pi(n'')$. In this case, $n'$ and $n''$ can be merged. These conditions for merging are related to the SQUISH algorithm described by Nilsson [2000] for deterministic teleo-reactive trees.

## 4.2 Experience Revision

The *Experience Revision* process adds state and transition observations to the Experience data structure. If the state-space is continuous and it is being sampled at a high frequency, it is impractical to add each sample. Instead, the agent should filter out samples that are not likely to be useful. The exact mechanism for determining significance is domain dependent, but one approach to filtering out states is to ignore all states until the agent has transitioned to a new state that is outside some threshold distance.

In addition to filtering out insignificant samples, the Experience Revision process is also responsible for removing old samples from the Experience data structure. The schedule for removing old samples depends upon memory and processor constraints. The removal of old samples also allows the agent to adapt to slowly changing environments.

## 4.3 Action Selection

The *Action Selection* process is responsible for continuously selecting a single action to execute. This process must be extremely efficient because it is typically executed as frequently as the agent samples the state of the world.

Usually, the agent should execute an action from the greedy region of the current state region as computed by the policy revision process. However, as with traditional reinforcement learning, there are issues with balancing exploration of the world and exploitation of the optimal policy. There have been many techniques proposed for balancing exploration with exploitation [Thrun, 1992], any of which may be used in this system.

As mentioned earlier, the agent is able to sense failures when they occur. What exactly counts as a failure depends upon the domain, but failures are generally situations where executing the same action repeatedly will not lead to success. In the Corner World domain described later, a failure might be trying to push through a wall. The Action Selection mechanism uses information about past failures to better direct the agent toward a goal.

## 4.4 Plan Revision

The *Plan Revision* process incrementally builds an optimal plan with the assumption that the current Model is correct.

[3] If the Map is implemented using a nearest-neighbor classifier, it does not make sense to make splits based on information gain since nearest neighbor can always separate two different sets (assuming that the same exact sample does not appear in both sets). Instead, an approach based on cross-validation or bootstrapping would be appropriate [Weiss, 1991].

This process calls the UPDATE$(n)$ function supplied by the Plan data structure, which updates the value and greedy action region for $n$.

One way to arrive at an optimal policy is to repeatedly iterate through the state regions calling UPDATE until convergence [cf. Bellman 1957]. However, doing so is not likely to be feasible in real time since the Model changes in response to the Map Revision and Experience Update processes. Instead, it is better to use a method related to prioritized sweeping [Moore and Atkeson, 1993]. Prioritized sweeping was designed for solving Markov decision processes (MDPs) but it can be easily adapted for solving the class of problems under consideration here. The idea of the algorithm is to prioritize updates of regions based on observed changes in the value function from earlier updates.

## 5 Experiments and Results

In the Corner World domain the agent starts at a random location on the starting line and maneuvers along an L-shaped track to the finish line. The track is contained within a $10 \times 10$ meter square, and the width of the track is 1 m. Each experimental run consists of 20 episodes of this task. The continuous compound discount rate $\beta$ was set to $0.01$.

The state space is represented by the tuple $(x, y)$ corresponding to the agent's location. The agent may translate in any direction at 5 m/s. The agent samples the environment and makes control decisions at 5 Hz. Also at 5 Hz, the position of the agent is perturbed by an amount selected from a normal distribution with 5 mm standard deviation.

Since AMPS has no prior knowledge about the dynamics of the world or its task, it considers all states to be equivalent. Hence, the agent must rely upon random exploration until it reaches a goal. Random exploration is not practical in the Corner World or most other interesting domains because the state space is so large, and therefore the agent needs some mechanism to bias it toward the relevant goal states [Whitehead, 1991]. In our experiments, we used a teacher to guide the agent to a goal for two training episodes. These two training episodes allowed AMPS to partition the state space and distribute the observed reward through the model. The teacher used in the experiments returned random actions 5% of the time.

For comparison purposes and to control for the information gained from the noisy teacher, we trained a decision tree based on the same teacher. The induction of the decision tree is done by using the separator function to partition the state space according to the actions taken by the teacher. This sort of supervised learning of control is known as *behavioral cloning* and has been successfully applied to a variety of domains including aircraft control [Sammut *et al.*, 1992]. The primary disadvantages of a behavioral clone is that it is entirely dependent on a good teacher and it is not able to adapt its behavior based on its own experience in contrast to AMPS.

In our experiments, AMPS was allowed to perform a merge or split of the state or action space at 5 Hz. Episodes terminated when the agent reached the finish line or after 100 s without success. The results of 100 repeated experiments are summarized in Table 1.

| | Successes | | Disc. Reward | |
|---|---|---|---|---|
| | mean | s.d. | mean | s.d. |
| AMPS | 16.02 | 6.04 | 9.54 | 4.13 |
| Clone | 12.75 | 5.74 | 7.87 | 3.85 |
| Teacher | 20.00 | 0.00 | 13.00 | 0.18 |
| Random | 0.00 | 0.00 | 0.00 | 0.00 |

Table 1: Experimental results of various agents in the Corner World domain. Each experiment was repeated 100 times. Shown above are the means and standard deviations of the number of successes and discounted reward in each run of 20 episodes.

## 6 Related Work

There have been algorithms proposed that assume a discrete action space but partition a large state space using a decision graph such as the G-ALGORITHM [Chapman and Kaelbling, 1991] and U-TREE [McCallum, 1995]. Uther and Veloso extended the work by McCallum to continuous state spaces [1998] and temporally abstract actions [2003]. None of these algorithms consider continuous action spaces.

Other techniques have been proposed for solving problems with large or infinite state spaces that do not involve partitioning the state space into regions. For example, parameterized function approximators, such as neural networks, may be used with traditional reinforcement learning techniques to estimate the value function [Bertsekas and Tsitsiklis, 1996].

Doya [2000] developed a reinforcement learning framework for continuous state, action, and time that uses function approximation. However, it assumes that the dynamics of the system are deterministic and the state and action spaces are represented by real-valued vectors. Smith [2002] used self-organizing maps to handle continuous state and action spaces represented as vectors of real-values.

Other work has been done on problems with durative actions in relational domains [Benson and Nilsson, 1995; Ryan, 2004]. The way AMPS uses SMDPs to model durative actions connects to work done in hierarchical reinforcement learning [surveyed in Barto and Mahadevan 2003].

## 7 Conclusions and Further Work

This paper has introduced a real-time system that integrates the adaptive partitioning of the state and action space and the incremental planning over the estimated model to produce goal-directed behavior. The system was tested on the Corner World domain and was shown to quickly learn successful behavior in the presence of noise in the environment.

AMPS has been designed to be broadly applicable across domains and representations. The state and action space may be infinite, and actions may have a continuous effect on the world and may be of variable duration. The Map data structure has been defined generically so that states and action spaces may have any representation. The Map may be implemented using a decision graph, in which case a separator must be defined. Alternatively, the Map may be implemented using a nearest-neighbor classifier, in which case only a distance metric must be defined. In either case, the separator or the distance metric can be built to exploit the structure inherent in the state and action representation.

AMPS is currently being tested in other more complex domains and compared against other methods such as traditional reinforcement learning with function approximation. Further work will investigate the use of different separators and distance metrics that take into account values of previous observations, making AMPS potentially applicable to domains where the current state is only partially observable.

## Acknowledgments

The authors would like to thank Nils Nilsson and the anonymous reviewers for their helpful suggestions on an earlier draft of this paper.

## References

[Barto and Mahadevan, 2003] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, October 2003.

[Bellman, 1957] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Benson and Nilsson, 1995] Scott Benson and Nils J. Nilsson. Reacting, planning and learning in an autonomous agent. In K. Furukawa, D. Michie, and S. Muggleton, editors, *Machine Intelligence*, volume 14, pages 29–64. Oxford University Press, 1995.

[Bertsekas and Tsitsiklis, 1996] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[Bradtke and Duff, 1995] Steven J. Bradtke and Michael O. Duff. Reinforcement learning methods for continuous-time Markov decision problems. In Gerald Tesauro, David S. Touretzky, and Todd K. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 393–400. MIT Press, 1995.

[Chapman and Kaelbling, 1991] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 726–731. Morgan Kaufmann, 1991.

[Cover and Hart, 1967] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, January 1967.

[Dean *et al.*, 1998] Thomas Dean, Robert Givan, and Kee-Eung Kim. Solving planning problems with large state and action spaces. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 102–110. AAAI Press, 1998.

[Doya, 2000] Kenji Doya. Reinforcement learning in continuous time and space. *Neural Computation*, 12(1):219–245, January 2000.

[Duda *et al.*, 2000] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2000.

[Kalos and Whitlock, 1986] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo Methods*, volume 1. John Wiley and Sons, 1986.

[Mahadevan and Connell, 1992] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55(2–3):311–365, June 1992.

[McCallum, 1995] Andrew Kachites McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, Department of Computer Science, University of Rochester, 1995.

[Moore and Atkeson, 1993] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, October 1993.

[Moore and Atkeson, 1995] Andrew W. Moore and Christopher G. Atkeson. The Parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21(3):199–233, December 1995.

[Munos and Patinel, 1994] Rémi Munos and Jocelyn Patinel. Reinforcement learning with dynamic covering of state-action space: Partitioning Q-learning. In *Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 354–363. MIT Press, 1994.

[Nilsson, 2000] Nils J. Nilsson. Learning strategies for mid-level robot control: Some preliminary considerations and results. www.robotics.stanford.edu/users/nilsson/trweb, 2000. Computer Science Department, Stanford University.

[Puterman, 1994] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[Ryan, 2004] Malcolm R. K. Ryan. *Hierarchical Reinforcement Learning: A Hybrid Approach*. PhD thesis, University of New South Wales, 2004.

[Sammut *et al.*, 1992] Claude Sammut, Scott Hurst, Dana Kedzier, and Donald Michie. Learning to fly. In *Proceedings of the Ninth International Workshop on Machine Learning*, pages 385–393. Morgan Kaufmann, 1992.

[Shannon, 1948] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[Smith, 2002] Andrew James Smith. Applications of the self-organising map to reinforcement learning. *Neural Networks*, 15(8–9):1107–1124, October–November 2002.

[Thrun, 1992] Sebastian B. Thrun. The role of exploration in learning control. In D. White and D. Sofge, editors, *Handbook for Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pages 527–559. Van Nostrand Reinhold, 1992.

[Uther and Veloso, 1998] William Uther and Manuela Veloso. Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 769–775. AAAI Press, 1998.

[Uther and Veloso, 2003] William Uther and Manuela Veloso. TTree: Tree-based state generalization with temporally abstract actions. In *Adaptive Agents and Multi-Agent Systems: Adaptation and Multi-Agent Learning*, volume 2636 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2003.

[Weiss, 1991] Sholom M. Weiss. Small sample error rate estimation for k-NN classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(3):285–289, March 1991.

[Whitehead, 1991] Steven D. Whitehead. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 607–613. AAAI Press, 1991.

[Yianilos, 1993] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321. Society for Industrial and Applied Mathematics, 1993.

# Apprenticeship Learning for Initial Value Functions in Reinforcement Learning

**Frederic Maire**
Faculty of Information Technology
Queensland University of Technology
Box 2434, Brisbane Q 4001
Australia
f.maire@qut.edu.au

**Vadim Bulitko**
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Canada
bulitko@ualberta.ca

## Abstract

Reinforcement Learning has had spectacular successes over the last several decades. While meant to require less human input than supervised learning, reinforcement learning can be substantially accelerated with *a priori* available domain expertise. The ways of providing human knowledge to a reinforcement learning agent vary from crafting state features to initial policy design to initial value function design. We chose the latter and propose a novel approach for acquiring a high-quality initial value function via apprenticeship learning. This approach works well in domain when a body of expert data are available. Our apprentice reinforcement learning (ARL) agent uses dynamic programming to compute values for the states visited by the expert. A Laplacian regularizer is then engaged to extrapolate these onto the entire state space. The result of this process is a high-quality initial value function to be further refined by any value-function based reinforcement learning method. In a grid world domain, ARL was able to speed up TD($\lambda$) learning method by a factor of two from a single observed expert's trace.

## 1 Introduction and Related Research

For some application domains of reinforcement learning, demonstrations by an expert are readily available. For instance, extensive databases of games between strong players can be easily found for popular board games like chess, go or hex. The task of learning by observing an expert is called *apprenticeship learning*. Most apprenticeship learning approaches try to mimic the observed experts by applying a supervised learning algorithm to learn a direct mapping from the states to the actions as pioneered by [Sammut *et al.*, 1992]. A recent alternative to this is to assume a reward-maximizing nature of the expert and then estimate the unknown reward function employed by the expert as a linear combination over given state features [Abbeel and Ng, 2004]. The premise of this approach is that the reward function, rather than the policy or the value function, is the most succinct, robust, and transferable definition of the task.

Work on *reward shaping* demonstrates that a well-chosen reward function can, indeed, significantly speed reinforcement learning [Laud and DeJong, 2003]. Furthermore, *safe* shaping rewards are differentials in a potential function [Ng *et al.*, 1999]. Therefore, by supplying a non-trivial potential function, the user can speed the learning process. This is not surprising since learning with shaping rewards derived from potential function is equivalent to learning without the shaping rewards but with the initial value function being equal to potential function [Wiewiora, 2003]. For instance, learning with the optimal value function $V^*$ as the potential function is equivalent to learning with $V^*$ as the initial value function – in both cases no further learning is needed.

Over the years, a number of methods for acquiring a non-trivial initial value function have been proposed. They vary from hand-engineering [Korf and Taylor, 1996; Ng *et al.*, 1999] and solving abstracted problems [Holte *et al.*, 1994; Korf and Felner, 2002] to artificial evolution [Ackley and Littman, 1991].

In this paper, we propose a novel method for deriving high-quality initial value functions from existing demonstrations by an expert. Our apprentice learning agent first builds an auxiliary graph whose vertices are points in the state space and whose arcs are transitions used by the expert. After estimating the state-values of the vertices visited by the expert, we extrapolate these values onto all other states with a graph Laplacian operator. This step requires a similarity measure over the state space. The resulting value function constitutes a starting point for any value function based reinforcement learning method. As the initial value function is substantially more informative than a random or optimistic value function frequently used with RL methods, the remaining on-line learning process is substantially accelerated. We evaluate the effectiveness of deriving an initial value function in a standard grid-world domain and find that a *single* observed demonstration by an expert speed up learning as much as 300 additional TD($\lambda$) episodes. This constitutes a two-fold speed-up.

The rest of the paper is organized as follows. In section 2 we detail our method. Section 3 sets up the experimental environment and presents results of the empirical evaluation. A discussion of future research follows.

## 2 ARL: The Proposed Approach

The presentation will be tailored to non-discounted episodic reinforcement learning tasks with a single absorbing state (called the *goal state* henceforth). While this setting covers a number of planning tasks [Bonet *et al.*, 1997; Aberdeen *et al.*, 2004], combinatorial puzzles [Korf, 1997; Korf and Felner, 2002], games [Tesauro, 1995; Schaeffer *et al.*, 2001], path-finding [Ng *et al.*, 1999; Shimbo and Ishida, 2003; Koenig, 2004] and other applications [Kohl and Stone, 2004], our approach can be extended to handle multiple goal states as well as non-episodic tasks.

The agent operates on a directed weighted graph whose vertices are agent's states. A single state is designated as the goal state. Edges correspond to actions available to the agent. By traversing an edge, the agent incurs a negative reward equal to the edge's weight. Arriving at the goal state ends the episode and does not deliver any additional reward. The learning task is to learn a state value function such that following this function greedily from any state will maximize the collected reward. This is equivalent to following a shortest path from any start state to the goal.

Notationally, the sum of negative rewards (i.e., the costs) incurred along a path $P$ between states $a$ and $b$ is denoted as $\mathrm{dist}_P(a, b)$. As usual, the value $V_\pi(s)$ of a state $s$ with respect to the policy $\pi$ is the expectation of the cumulative cost incurred by the agent when following the policy $\pi$ from the state $s$.

### 2.1 Step 1: Collecting Observations

The observed demonstrations by an expert are encoded in the form of trajectories in the state space. The premise of our approach is that an expert agent will follow trajectories that are close to optimal. On this premise, we induce a partial order over the states visited by the expert. The training set (i.e., the set of state trajectories) therefore induces a directed subgraph on the set of visited states. If a sequence of states from an expert agent has a state repeated, we prune out all states between the two occurrences. That is, the sequence $(\dots, s_{t-1}, s_t, s_{t+1}, \dots, s_{t+m}, s_{t+m+1}, \dots)$ with state $s_t = s_{t+m}$ will become $(\dots, s_{t-1}, s_t, s_{t+m+1}, \dots)$.

Once all cycles have been eliminated, we label each state in each sequence with its distance from the goal state. We then merge multiple sequences labeled with their distances into one directed labeled graph. If several sequences pass through the same state, we use the minimal distance as the final label of this state. In other words, we compute the shortest weighted paths to the goal state in this auxiliary graph using Disjktra algorithm. At this stage, the value of a state that has been visited by an expert is approximated by $\widehat{h}(s)$, the length of a shortest path from the goal state to $s$.

### 2.2 Step 2: Generalization with a Graph Laplacian

Several approaches in the past generalized high-quality values of sample states onto the entire space. In [Levner and Bulitko, 2004], the researchers applied reinforcement learning methods to acquire a computer vision policy in the domain of tree canopy recognition. A training set of aerial images was used as the starting states of an RL-agent. Since the "ground truth" was provided for each training image, a roll-out (i.e., full subtree expansion) technique allowed to compute the exact values of each input and subsequent states. The resulting very sparse samples were generalized onto the entire abstracted state space using Artificial Neural Networks and hand-crafted state features.

In [Bulitko and Wilkins, 2003], Artificial Neural Networks were used to generalize the values of damage control status (i.e., state) of a naval vessel computed from past scenarios onto the entire state space. Again, hand-built state features were used to reduce the dimensionality of the state space.

Our setting is different from the first approach mentioned above as we assume having access to traces of expert behavior as opposed to the final state that the expert arrived at. An additional difference from this and the second approach is that we take an advantage of the similarity between states by using a Graph Laplacian. This allows us to induce high-quality value functions from a single expert trajectory whereas both aforementioned systems required large bodies of labeled images or damage control scenarios.

We will now present the details of Step 2 of the ARL approach. The training set $\{[s, \widehat{h}(s)]\}$ computed in Step 1 is now fed into a Graph Laplacian regularizer [Chung-Graham, 1997] to generalize the data onto all other states. To illustrate the role of the graph Laplacian, consider the toy-sized graph of Figure 1 where vertex $5$ represents the goal state. Given the values of the vertices on the path $(7, 4, 2, 1, 5)$, traversed by the expert, we would like to generalize the values of these vertices to non-visited vertices. That is, we are looking for reasonable values for vertex $3$ and vertex $6$. The value $v_3$ should be related to the values $v_2$ and $v_7$ of its neighbours. Similarly, the value $v_6$ should be related to $v_1$, $v_4$ and $v_7$. We determine the values $v_3$ and $v_6$ by minimizing the Laplacian cost function:

$$(v_3 - 2)^2 + (v_3 - 4)^2 + (v_6 - 1)^2 + (v_6 - 3)^2 + (v_6 - 4)^2.$$

From here the optimal values are computed as $v_3 = 3$ (which happens to be its correct value) and $v_6 = 2.667$ (which exceeds its correct value of 2). If there were an edge between vertex 3 and vertex 6, we would add the term $(v_3 - v_6)^2$ to this cost function. In general, the cost function to be minimized is of the form:

$$\frac{1}{2}\tilde{v}^T H \tilde{v} - f^T \tilde{v}$$

where $\tilde{v}$ represents the vector of the values of the states not visited by the experts. The solution for $\tilde{v}$ is $H^+ f$, where $H^+$ is the pseudo-inverse of $H$. Alternatively, we can formulate the optimization problem as a quadratic program on $v$ (the full vector) with equality constraints for the visited states. The values of the states visited by the experts are clamped to the values found in Step 1. The pseudo-code of Step 2 is found in Algorithm 1.

The use of a set of non-visited states is motivated by the fact we may have access to valuable information on the entire state space. For instance, in game playing, such information is in the form of a database of games where some games were played by experts and other games by non-expert players. The board positions reached by non-expert players are still relevant to learning a strong program for this game as

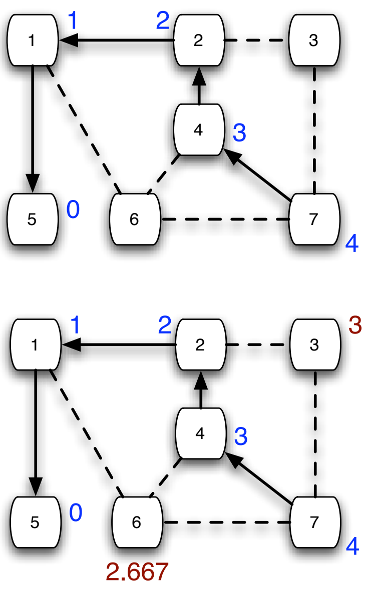


Figure 1: **Top:** a hand-traceable micro-graph used to illustrate the first two steps of our ARL algorithm. The vertex labels are inside the nodes (1 through 6). The expert path from vertex 7 to vertex 5 (the goal) is shown by the solid arrows. The numbers next to vertices of the path are the values of the vertices $\widehat{h}$ computed in Step 1. **Bottom:** the values estimated for vertices 3 and 6 in Step 2 are shown.

they correspond to states likely to be visited. If we have a reasonable quality similarity measure between board positions, then we can derive a weighted graph similar to the graph of Figure 1. The weights of the egdes are the similarity measures between states and the Laplacian will be defined for the weighted graph. Then we can induced some initial values of the board positions for the games of non-expert players. Despite not having seen expert behavior in these states, the initial values can still be expected to be of a higher quality than a random initial values.

As usually done within the semi-supervised learning framework [Zhu *et al.*, 2005], the machine learning module is not only given a labeled data set consisting of input-output pairs $\{(x_1, v_1), \ldots, (x_l, v_l)\}$, but also a (typically much larger) unlabeled data set $\{x_{l+1}, ..., x_m\}$ where the $x_i$'s are in some general input space and the $v_i$'s are real values. In addition to this training set $T$, we are given a graph of similarity on the input state space. That is, two vertices are connected by an edge if they are deemed similar. From the adjacency relation of the graph, the Laplacian operator $L$ is defined as the matrix whose entries satisfy:

$$L_{i,j} = \begin{cases} 1 & \text{if } i \sim j \\ -d_i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

where $d_i$ represents the degree of vertex $i$, and $i \sim j$ means that vertex $i$ is adjacent to vertex $j$. It is easy to check that:

$$v^T L v = -\frac{1}{2}\sum_{i \sim j}(v_i - v_j)^2.$$

The scalar $v^T L v$ quantifies how much $v$ varies locally, or how smooth $v$ is over the vertex set.

```
Algorithm 1 $V$ = InitializeValueFunction($T$)
Require: $T$, a collection of experts' trajectories in a discrete
    state space.
Ensure: $V$ is an initial value function.
    - Build auxiliary digraph by inserting an arc between state
    $s_i$ and $s_j$ whenever $s_j$ is a successor state of $s_i$ in a expert
    trajectory of $T$.
    - Compute the distance to a goal of each visited state vertex
    in the auxiliary digraph.
    - Clamping the computing value of visited states, determine the vector $V$ of values of the other states by minimizing the Laplacian subject to the clamping constraint.
```

In [Smola and Kondor, 2003], graph Laplacians are put into the principled framework of Regularization Theory and a family of regularization operators (equivalently, kernels) on graphs that include Diffusion Kernels is proposed. It is worth mentioning that the approach we propose herein can be extended to any of these regularization operators. For instance, consider the diffusion kernel $K$ and the training set $T$ as above. Then we would be maximizing $v^T L v$ subject to $v_i = \widehat{h}(x_i)$ for $i \in [1 : l]$.

### 2.3 Step 3: On-line Refinement via RL

The result of Step 2 is the initial value function $V_0(s) = -h(s)$. This function is then refined during the on-line reinforcement learning with a value-function based algorithm such as the simple value iteration or TD($\lambda$) [Sutton and Barto, 1998]. The initial value function computed during Steps 1 and 2 captures certain expert knowledge induced from the recorded experiences. We believe that our approach is superior to the learning by value iteration by replaying the observed state trajectories of the experts. Indeed, assume we have two experts $A$ and $B$ such that $A$ makes a good decision in the state $a$ and a lower quality decision in the state $b$. On the other hand, the expert $B$ is strong in the state $b$ and less optimal in the state $a$. The value of the state $a$ obtained by value iteration on the trajectories of $A$ and $B$ will depend on which expert trajectory was presented last. If $B$ was used last, then the result will not be as accurate as if $A$ was used last. Our approach does not suffer from this ordering problem because of the way we determine the values of the states in the auxiliary graph.

## 3 Empirical Evaluation

To assess the benefits of the proposed approach, we have tested the performance of the ARL in a classical maze-like grid world such as the one in Figure 2. The primary objective of the empirical studies was to measure the acceleration in reinforcement learning with respect to the amount and quality of observed expert trajectories.

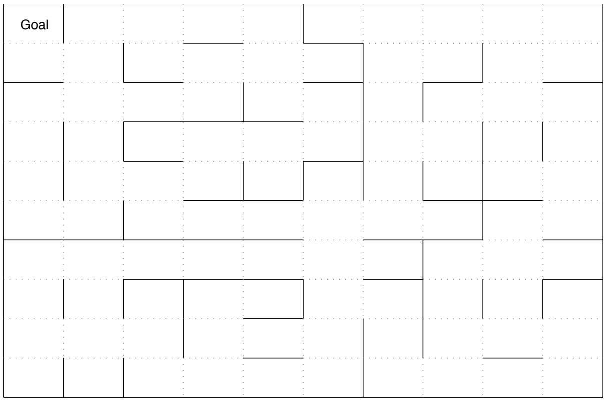


Figure 2: A sample grid-world used for empirical evaluation.

We used a heuristically guided full-width fixed-lookahead search with the user-specified lookahead $d$ as the domain expert and recorded the trajectories it traversed. In the state $s$, the expert agent with the lookahead $d$ computes the frontier $S(s,d)$ of all states reachable from $s$ in exactly $d$ moves. For each of the state $s'$ on the frontier $S(s,d)$, its heuristic value $h(s')$ is computed. As commonly done, we used the Manhattan distance between the state and the goal state as the heuristic. The agent then takes $d$ moves from state $s$ to the frontier state with the most promising heuristic:

$$\arg \min_{s' \in S(s,d)} h(s').$$

Note that in this example, the distance from $s$ to $s'$ is ignored in the selection as all moves have uniform cost. Upon reaching a dead end, the agent backtracks. The backtracking causes re-visits of states but such loops are removed in Step 1 of the ARL method. As our expert agent remembers its trajectory during each trial, we do not need to adjust the heuristic $h$ as done in real-time search algorithms such as RTA* [Korf, 1990] to avoid infinite cycles.

As we real-time search agents, deeper lookahead leads (on average) to more optimal paths. Thus, by varying $d$, we were able to adjust the degree of expertise from being a perfect agent (e.g., when the goal state is within the lookahead radius) to the uninformed search (e.g., $d = 0$). In Step 2 of ARL, the heuristic estimates derived from the recorded trajectories were extrapolated onto the entire maze using the Graph Laplacian approach. In Step 3, the initial value function produced was further refined by a temporal difference (TD) algorithm with $\lambda = 0.7, \epsilon = 0, \gamma = 1.0$ and a learing rate $\alpha$ of $0.1$ [Sutton and Barto, 1998, Figure 7.7, p. 174]. The initial state for the TD agents was chosen randomly while the goal state and the maze were fixed.

In order to evaluate the quality of a value function $V$, we computed the average expected return of a policy greedy with respect to $V$:

$$V^{\pi} = \frac{1}{|S|} \sum_{s \in S} E_{\pi} \left\{ \sum_{i} r_i | s_{\text{start}} = s \right\}.$$

A policy greedy with respect to $V$ always selects the action that is expected to lead to the state with the highest $V$-value. While it is computationally less expensive to assess the quality of a value function as the mean squared deviation from the optimal value function $V^*$, [Li *et al.*, 2004b; 2004a] demonstrates that such measure does not lead to maximizing reward collected by the agent. The value of the greedy policy with the zero value function is $-27.69$. The value of the optimal policy is $-12.83$.

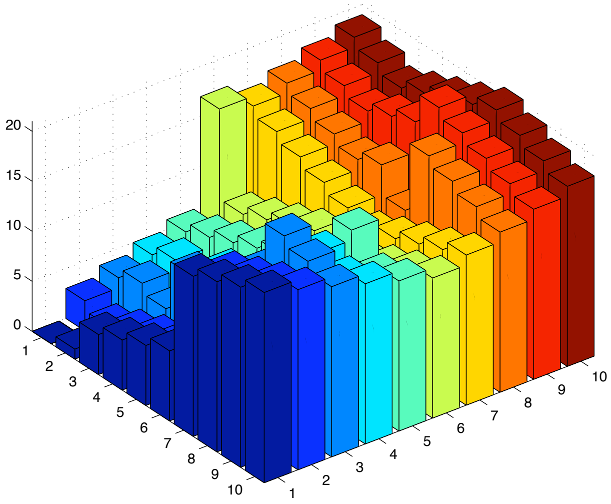


Figure 3: The optimal value function of the maze in Figure 2. The value $V^{\pi^*}$ of the optimal policy $\pi^*$ is $-12.82$. To make the plot easier to read, we are showing the negative policy value as positive numbers. Hence, visually lower values are better.

To evaluate the benefits of the apprenticeship-based value function initialization, we compared the policy values from the ARL-initialized policies with TD-learned policy with the initial value function of zero. Initializing the value function with the Manhattan distance immediately yielded a greedy policy whose performance is $-18.20$. This initial heuristic is equivalent to the one obtained by approximately $100$ episodes of zero-initialized TD-learning as seen in Figure 4.

Figure 6 shows the induced value function from the single trajectory in Figure 5. It is remarkable that the Laplacian operator allows the capture of a rough landscape of the optimal value function from merely a single trajectory: as seen by comparing Figure 3 and Figure 6.

Table 1: **Top:** value of the greedy policy over the value function derived by ARL. A single trajectory of the expert with the shown lookahead was used as the input to ARL. **Bottom:** the number of TD-learning episodes required to learn a value function of the quality produced by ARL from a single expert's trajectory.

| Expert's lookahead | 1 | 5 | 10 | 30 |
|---|---|---|---|---|
| Policy value $V^{\pi}$ | -19.31 | -19.28 | -16.39 | -13.47 |

| Expert's lookahead | 1 | 5 | 10 | 30 |
|---|---|---|---|---|
| Equivalent number of TD episodes | 95 | 95 | 140 | 300 |

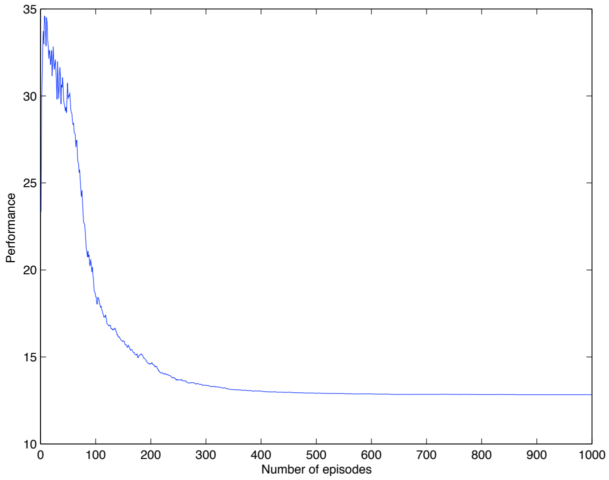


Figure 4: Learning curve of TD($\lambda = 0.7$, $\alpha = 0.1$, $\gamma = 1.0$) initialized with a zero value function. Each point is averaged over 30 independent TD-learning runs starting from random initial states. To make the plot easier to read, we are showing the negative policy value as positive numbers. Hence, visually lower values are better.

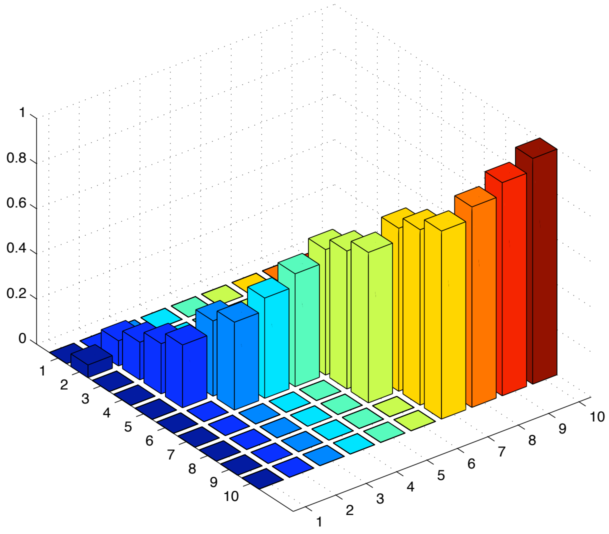


Figure 5: Values of states computed in Step 1 from a single a trajectory of our expert search agent with the lookahead of 5.

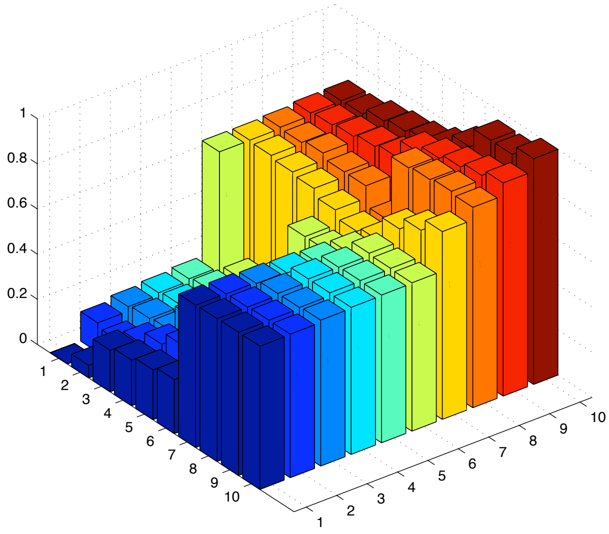


Figure 6: The results of Step 1 (Figure 5) are generalized onto the entire space with the Laplacian in Step 2.

The top portion of Table 1 shows how the quality of the initial value functions produced by ARL from a single expert's trajectory depends on the expert's lookahead depth. The bottom portion demonstrates the number of TD learning episodes required to produce a value function of the same quality as those produced by ARL.

We have experimented with the number of trajectories used as the input to ARL. In this micro-maze, the gains in performance are negligible. We expect that in larger and more complex state spaces leading to sparser sampling with each trajectory, ARL will be able to benefit substantially from a collection of recorded expert problem-solving traces.

## 4 Future Work and Conclusions

The promising initial results encourage several follow-up research directions. First, we would like to extend the approach to non-episodic tasks. Second, it is of interest to investigate the extent to which this novel method applies to approximated (as opposed to tabular) value function. Third, we are presently working on scaling up this approach to challenging real-world tasks such as the game of Hex wherein human players are presently far superior to computer players and massive amounts of past games are available for apprenticeship learning.

In summary, we have proposed a novel effective approach for generalizing problem-solving traces from expert agents into high-quality initial value functions. Practical evaluation in a standard grid world micro-domain demonstrated that even as few as a single recorded expert trajectory speeds up temporal differences with eligibility traces TD($\lambda$) by as many as 300 learning episodes. That is equivalent to halving the convergence process.

## References

[Abbeel and Ng, 2004] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, 2004.

[Aberdeen *et al.*, 2004] Douglas Aberdeen, Sylvie Thiebaux, and Lin Zhang. Decision-theoretic military operations planning. In *Proceedings of International Conference on Automated Planning and Scheduling*, 3–13, Whistler, Canada, 2004.

[Ackley and Littman, 1991] David H. Ackley and Michael L. Littman. Interactions between learning and evolution. In *Artificial Life II*, volume 10, 487–509. AddisonWesley, Redwood City, CA, Santa Fe Institute Studies in the Sciences of Complexity, 1991.

[Bonet *et al.*, 1997] Blai Bonet, Gabor Loerincs, and Hector Geffner. A fast and robust action selection mechanism for planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 714–719, 1997. AAAI Press / The MIT Press.

[Bulitko and Wilkins, 2003] V. Bulitko and D.C. Wilkins. Qualitative simulation of temporal concurrent processes using Time Interval Petri Nets. *Artificial Intelligence*, 144(1-2):95 – 124, 2003.

[Chung-Graham, 1997] F. Chung-Graham. *Spectral Graph Theory*. Spectral Graph Theory. Number 92 in CBMS Regional Conference Series in Mathematics. AMS, 1997.

[Holte *et al.*, 1994] R.C. Holte, C. Drummond, M.B. Perez, R.M. Zimmer, and A.J. MacDonald. Searching with abstractions: A unifying framework and new high-performance algorithm. In *Proceedings of the Canadian Artificial Intelligence Conference*, 263–270, 1994.

[Koenig, 2004] Sven Koenig. A comparison of fast search methods for real-time situated agents. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2 (AAMAS'04)*, 864 – 871, 2004.

[Kohl and Stone, 2004] Nate Kohl and Peter Stone. Machine learning for fast quadrupedal locomotion. In *The Nineteenth National Conference on Artificial Intelligence*, 611–616, July 2004.

[Korf and Felner, 2002] R. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

[Korf and Taylor, 1996] R. E. Korf and L. A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, 1202–1207, Portland, Oregon, USA, 1996. AAAI Press / The MIT Press.

[Korf, 1990] R.E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[Korf, 1997] R. Korf. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of the Workshop on Computer Games (W31) at IJCAI-97*, 21–26, Nagoya, Japan, 1997.

[Laud and DeJong, 2003] Adam Laud and Gerald DeJong. The influence of reward on the speed of reinforcement learning:an analysis of shaping. In Tom Fawcett and Nina Mishra, editors, *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*. AAAI Press, 2003.

[Levner and Bulitko, 2004] Ilya Levner and Vadim Bulitko. Machine learning for adaptive image interpretation. In *Proceedings of the National Conference on Artificial Intelligence (AAAI) and Sixteenth Innovative Applications of Artificial Intelligence Conference (IAAI)*, 870 – 876, San Jose, California, 2004.

[Li *et al.*, 2004a] Lihong Li, Vadim Bulitko, and Russell Greiner. Batch reinforcement learning with state importance. In *Proceedings of European Conference on Machine Learning, Poster Section*, 566–568, Pisa, Italy, 2004.

[Li *et al.*, 2004b] Lihong Li, Vadim Bulitko, and Russell Greiner. Focus of attention in sequential decision making. In *Proceedings of National Conference on Artificial Intelligence (AAAI), Workshop on Learning and Planning in Markov Processes - Advances and Challenges*, San Jose, California, 2004.

[Ng *et al.*, 1999] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of of ICML*, 1999.

[Sammut *et al.*, 1992] C. Sammut, S. Hurst, D. Kedzier, and D. Michie. Learning to fly. In *Proceedings of ICML*, Aberdeen, 1992. Morgan Kaufmann.

[Schaeffer *et al.*, 2001] Jonathan Schaeffer, Markian Hlynka, and Vili Jussila. Temporal difference learning applied to a high-performance game-playing program. In *Proceedings of IJCAI*, 529–534, 2001.

[Shimbo and Ishida, 2003] Masashi Shimbo and Toru Ishida. Controlling the learning process of real-time heuristic search. *Artificial Intelligence*, 146(1):1–41, 2003.

[Smola and Kondor, 2003] A. Smola and R. Kondor. Kernels and regularization on graphs, 2003.

[Sutton and Barto, 1998] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[Tesauro, 1995] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), March 1995.

[Wiewiora, 2003] Eric Wiewiora. Potential-based shaping and Q-value initialization are equivalent. *Journal of Artificial Intelligence Research*, 19:205–208, 2003.

[Zhu *et al.*, 2005] Xiaojin Zhu, Jaz Kandola, Zoubin Ghahramani, and John Lafferty. Semi-supervised kernels via convex optimization with order constraints. In *Advances in Neural Information Processing Systems 17*. MIT Press, Cambridge, MA, 2005.

# Fast Reachability Analysis for Uncertain SSPs

**Olivier Buffet**
National ICT Australia &
The Australian National University
olivier.buffet@nicta.com.au

## Abstract

Stochastic Shortest Path problems (SSPs) can be efficiently dealt with by the *Real-Time Dynamic Programming* algorithm (RTDP). Yet, RTDP requires that a goal state is always reachable, what can be checked easily for a certain SSP, and with a more complex algorithm for an *uncertain* SSP, i.e. where only a possible interval is known for each transition probability. This paper makes a simplified description of these two processes, and demonstrates how the time consuming uncertain analysis can be dramatically speeded up. The main improvement still needed is to turn to a symbolic analysis in order to avoid a complete state-space enumeration.

## 1 Introduction

In decision-theoretic planning, Markov Decision Problems [Bertsekas and Tsitsiklis, 1996] are of major interest when a probabilistic model of the domain is available. A range of algorithms make it possible to find plans (policies) optimizing the expected long-term utility. Yet, optimal policy convergence results all depend on the assumption that the probabilistic model of the domain is accurate.

Unfortunately, a large number of MDP models are based on uncertain probabilities (and rewards). Many rely on statistical models of physical or natural systems, may they be toy problems such as the mountain-car or the inverted-pendulum, or real problems such as plant control or animal behavior analysis. These statistical models are based on simulations (themselves being mathematical models), observations of a real system or human expertise.

Working with uncertain models first requires answering two closely related questions: 1- how to model this uncertainty, and 2- how to use the resulting model. Existing work shows that uncertainty is sometimes represented as a set of possible models, each assigned a model probability [Munos, 2001]. The simplest example is sets of possible models that are assumed equally probable [Bagnell *et al.*, 2001; Nilim and Ghaoui, 2004]. Rather than construct a possibly infinite set of models we represent model uncertainty by allowing each probability in a single model to lie in an interval [Givan *et al.*, 2000; Hosaka *et al.*, 2001].

Uncertain probabilities have been investigated in resource allocation problems [Munos, 2001] — investigating efficient exploration [Strehl and Littman, 2004] and state aggregation [Givan *et al.*, 2000] — and policy robustness [Bagnell *et al.*, 2001; Hosaka *et al.*, 2001; Nilim and Ghaoui, 2004]. We focus on the later, considering a two-player game where the opponent chooses from the possible models to reduce the long-term utility.

Our principal aim is to develop an efficient planner for a common sub-class of MDPs for which optimal policies are guaranteed to eventually terminate in a goal state: Stochastic Shortest Path (SSP) problems. A greedy version of *Real-Time Dynamic Programming algorithm* (RTDP) [Barto *et al.*, 1995] is particularly suitable for SSPs, as it finds good policies quickly and does not require complete exploration of the state space. Yet, if it can be made robust [Buffet and Aberdeen, 2004; 2005], it also requires that a goal state is reachable from any visited state, which can be checked through a reachability analysis.

This paper makes a simple description of the reachability analysis for certain and uncertain SSPs [Buffet, 2004], and shows how the time consuming uncertain analysis can be dramatically speeded up. In Section 2 we present SSPs, RTDP and robustness. We then explain the algorithms for reachability analysis in the certain and uncertain case. Finally, the fast uncertain reachability analysis is depicted and practical experiments are presented before a conclusion.

## 2 Background

### 2.1 Stochastic Shortest Path Problems

A Stochastic Shortest Path Markov Decision Problem [Bertsekas and Tsitsiklis, 1996] is defined here as a tuple $\langle S, s_0, G, A, T, c\rangle$. It describes a control problem where $S$ is the finite set of **states** of the system considered, $s_0 \in S$ is a starting state, and $G \subseteq S$ is a set of goal states. $A$ is the finite set of possible **actions** $a$. Actions control transitions from one state $s$ to another state $s'$ according to the system's probabilistic dynamics, described by the **transition function** $T$ defined as $T(s,a,s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$. The aim is to optimize a performance measure based on the **cost function** $c : S \times A \times S \to \mathbb{R}^+$.[1]

[1] As the model is not sufficiently known, we do not make the usual assumption $c(s,a) = \mathbb{E}_{s'}[c(s,a,s')]$.

SSPs assume a goal state is reachable from any state in $S$, at least for the optimal policy, so that one cannot get stuck in a looping subset of states. An algorithm solving an SSP has to find a **policy** that maps states to probability distributions over actions $\pi : S \to \Pi(A)$ which optimizes the chosen performance measure, here the **value** $V$ defined as the expected sum of *costs* to a goal state.

In this paper, we only consider SSPs for planning purposes, with only inaccurate knowledge of the transition function $T$. In this framework, well-known stochastic dynamic programming algorithms such as *value iteration* (VI) make it possible to find a deterministic policy that corresponds to the minimal expected long-term cost $V$. *Value iteration* works by computing the value function $V^*(s)$ that gives the expected reward of the optimal policies. It is the unique solution of the fixed point equation [Bellman, 1957]:

$$V(s) \quad = \quad \min_{a \in A} \sum_{s' \in S} T(s,a,s') \left[ c(s,a,s') + V(s') \right]. \quad (1)$$

Updating $V$ with this formula leads to the optimal value function. For convenience, we also introduce the $Q$-value:$Q(s,a) = \sum_{s' \in S} T(s,a,s')[c(s,a,s') + V(s')]$.

This kind of problem can easily be viewed as a shortest path problem where choosing a path only probabilistically leads you to the expected destination. SSPs can represent a useful subset of MDPs. They are essentially a finite-horizon MDP with no discount factor.

## 2.2 RTDP

A first algorithm making use of the structure of SSPs is a version of the *Real-Time Dynamic Programming* algorithm (RTDP) [Barto *et al.*, 1995]. It uses the fact that the SSP cost function is positive and the additional assumption that every trial will reach a goal state with probability 1. Thus, with a zero initialization of the $J$, both the $J$ and $Q$-values monotonically increase during their iterative computation.

The idea behind RTDP (Algorithm 1) is to follow paths from the start state $s_0$, always greedily choosing actions of low value and updating $Q(s,a)$ as states $s$ are encountered. In other words, the action chosen is the one expected to lead to the lowest future costs, until the iterative computations show that another action may do better.

**Algorithm 1** RTDP algorithm for SSPs

```
RTDP($s$:state) // $s = s_0$
repeat
    RTDPTRIAL($s$)
until // no termination condition
.......................................................
RTDPTRIAL($s$:state)
while ¬GOAL($s$) do
    $a$ =GREEDYACTION($s$)
    $J(s)$ =QVALUE($s, a$)
    $s$ =PICKNEXTSTATE($s, a$)
end while
```

RTDP has the advantage of quickly avoiding plans that lead to high costs. Thus, the exploration looks mainly at a promising subset of the state space. Because it follows paths by simulating the system's dynamics, common transitions are favored, so that good policies are obtained early. Yet, the bad update frequency of rare transitions slows the convergence.

## 2.3 Robust Value Iteration

We now turn to the problem of taking the model's uncertainty into account when looking for a "best" policy. The (possibly infinite) set of alternative models is denoted $\mathcal{M}$.

We follow the approach described in [Bagnell *et al.*, 2001], that consists of finding a policy that behaves well under the worst possible model. This amounts to considering a two-player zero-sum game where a player's gain is its opponent's loss. The player chooses a policy while its "disturber" opponent simultaneously chooses a model. A simple process may be used to compute the value function while looking simultaneously for the worst model. It requires the hypothesis that state-distributions $T(s,a,\cdot)$ are independent from one state-action pair $(s,a)$ to another. Under this assumption, the worst model can be chosen locally when $Q$ is updated for a given state-action pair. If this assumption does not always actually hold, it induces a larger set of possible models, what results in a worst-case assumption in the pessimistic approach.

**Problem —** We are particularly interested in handling *uncertain SSPs* (USSP), where only intervals of possible transition probabilities are known: $T(s,a,s') \in [Pr^{\min}(s'|s,a), Pr^{\max}(s'|s,a)]$. Yet, to use (robust) RTDP, this theorem is of major interest:

**Theorem 1. [Bertsekas and Tsitsiklis, 1996]** *If the goal is reachable with positive probability from every state, RTDP unlike the greedy policy cannot be trapped into loops forever and must eventually reach the goal in every trial. That is, every RTDP trial terminates in a finite number of steps.*

The purpose of this paper is to determine from which states a goal state is still reachable in SSPs. The uncertain case could be brought back to the certain case by finding an appropriate pessimistic model. To that end, our policy should be fixed to one that chooses all actions with equal probability and the opponent could then learn a model to prevent goal states from being reached. Yet, the opponent's problem is no SSP, what would imply coming back from RTDP to *Value Iteration*. Moreover, we prefer performing a graph analysis, as it gives more practical information and would be a first step toward a symbolic analysis avoiding the enumeration of the complete state-space.

# 3 Reachability Analyses

When applying algorithms such as RTDP on an SSP having no proper policy, the main problem is to detect if current state $s$ still has a positive probability of reaching the goal set, in which case $s$ is said to be "`reaching`". If $s$ is non-`reaching`, RTDP should stop and a specific process be applied, such as associating this state to an infinite cost.

Non-`reaching` states constitute looping sub-sets of states which we will refer to as "dead-ends". The process just described results in dead-ends avoidance. Yet some states may be `reaching` but also have a positive probability to lead to a dead-end whatever the policy. If non-`reaching` states incur infinite costs, these "`dangerous`" states will necessarily have an infinite long-term cost to the goal. It would thus be of interest to also identify these `dangerous` states.

Note that what to do when in a non-`reaching` state may depend on the user's preferences. But in all cases the first step is to perform a "reachability analysis" through a graph traversal beginning with goal states. Then, if required, a "danger analysis" can be performed through another (simpler) graph traversal beginning with non-`reaching` states. This paper mainly focuses on the "reachability analysis", as this process is necessary and somewhat subtle in the case of USSPs.

### 3.1 Certain SSP

In a certain SSP, if $s'$ is `reaching`, any state $s$ such that $T(s,a,s') > 0$ for some action $a$ is also reaching. This results in a straightforward analysis by making a graph traversal starting with goal states.

Let $Parents(s)$ be the set of states $s'$ for which there exists a transition $(s',a) \rightarrow s$: $Parents(s) = \{s' \in S \text{ s.t. } \exists a \in A Pr(s|s',a) > 0\}$. Alg. 2 uses this information to perform the reachability analysis. Then states which have not been marked as `reaching` are dead-ends, and a second graph traversal starting with these states will identify `dangerous` states (see Alg. 3).[2]

**Algorithm 2** PROPAGATEREACHABILITYSSP ($Parents$)

```
PUSHALL($G, st$) {$st$: stack of goal states}
while $st \neq \emptyset$ do
    POP($s, st$)
    if $\neg reaching(s)$ then
        MARK($s$, reaching)
        PUSHALL($Parents(s), st$)
    end if
end while
```

**Algorithm 3** PROPAGATEDANGER($Parents$)

```
for all $s \in S$ s.t. $\neg reaching(s)$ do
    PUSH($s, st$)
end for
while $st \neq \emptyset$ do
    POP($s, st$)
    if $\neg dangerous(s)$ and $\forall\, a \in A$ :
    $\exists\, s' \in S$ s.t. $Pr(s'|s,a) > 0 \;\&\; dangerous(s')$ then
        MARK($s$, dangerous)
        PUSHALL($Parents(s), st$)
    end if
end while
```

### 3.2 Uncertain SSP

In an uncertain SSP, the reachability analysis depends on the fact that the opponent can forbid a transition $(s,a) \rightarrow s'$ if $Pr^{\min}(s'|s,a) = 0$. A difficulty is that $Pr^{\min}(s'_1|s,a) = 0$ and $Pr^{\min}(s'_2|s,a) = 0$ are not sufficient to tell if $s'_1$ and $s'_2$ may be forbidden simultaneously in some possible model. Fig. 1 shows an example where the 3 potentially reachable states cannot be forbidden simultaneously (there is no possible model s.t. $\forall j \in \{1,2,3\}\; T(s_o, a_0, s'_j) = 0$). With upper probabilities of 1, any 2 states could be forbidden.

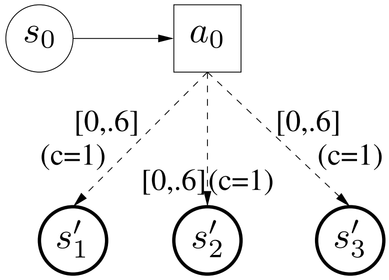


Figure 1: USSP where only 1 of the 3 reachable states can be forbidden (goal states in bold circles).

Let us define the set of all lists of states which cannot be forbidden simultaneously (from $(s,a)$):[3]

$$L^{\oslash}_{(s,a)} = \left\{ \begin{array}{c} l \subseteq S \text{ s.t. } s' \in l \Rightarrow Pr^{\max}_{(s'|s,a)} > 0, \\ \text{and} \begin{array}{c} \exists s' \in l \text{ s.t. } Pr^{\min}_{(s'|s,a)} > 0 \\ \text{or } \sum_{s' \in S \setminus l} Pr^{\max}_{(s'|s,a)} < 1 \end{array} \end{array} \right\}.$$

To know if a given action $a$ can lead to a goal state from current state $s$, one has to find at least one such list where all states are `reaching`. In this case, the opponent cannot prevent the planner having some chance of terminating. The reachability analysis only needs to work with the subset of minimal lists:

$$L^{\min\oslash}_{(s,a)} = \left\{ \begin{array}{c} l \in L^{\oslash}_{(s,a)} \text{ s.t. } \forall l' \in L^{\oslash}_{(s,a)} : \\ l \cap l' = l \text{ or } (l \cap l') \notin L^{\oslash}_{(s,a)} \end{array} \right\}.$$

In other words, removing any state of such a list makes it possible for the opponent to forbid all states in the list. On Fig. 1: $L^{\min\oslash}_{(s_o,a_o)} = \{\{s'_1, s'_2\}, \{s'_1, s'_3\}, \{s'_2, s'_3\}\}$.

From this basic idea, two problems arise:

- How to perform the reachability analysis ?
- How to obtain these lists ?

We now just give a brief idea of the answers to these two questions (details in Sections 3.2 and 3.3 of [Buffet, 2004]).

**Performing the Reachability Analysis –** The minimal lists we have just described are defined with respect to a given state-action pair $(s,a)$. They are used to obtain a new set $L^{\min\oslash}_{(s)}$ of minimal lists relative to the state $s$, since the precise action chosen is of no interest when just checking whether a state could reach the goal or not.

From there, determining which states can reach a goal state is again done through a propagation starting from these goal

[2] Alg. 3 can be implemented efficiently by remembering which state-action pairs are known to be dangerous.

[3] $\oslash \sim$ "states **cannot** be forbidden simultaneously"

states. This "back"-propagation takes place in an `AND-OR` graph where nodes are states and their minimal lists, as illustrated by Fig. 2. This is an `AND-OR` graph because a list is "`reaching`" if *all* its children states are `reaching` (`AND`), and a state is `reaching` if *one* of its children lists is `reaching` (`OR`).

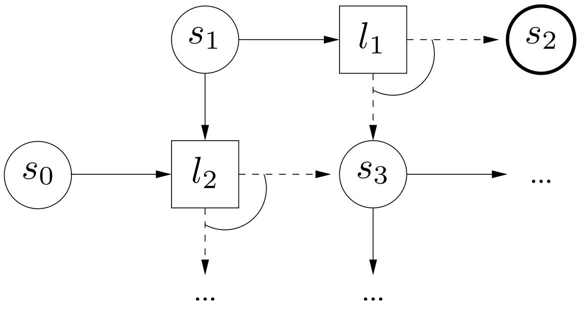


Figure 2: Example of `AND-OR` graph in which the reachability analysis is done (starting with goal states as $s_2$ here). If $s_3$ is `reaching`, then so is $l_1$ (the opponent cannot forbid $s_2$ and $s_3$), and therefore $s_1$.

After this reachability analysis for uncertain SSPs, the danger analysis from Alg. 3 can be performed with no modification, using the most probable model for example. Indeed in this second phase the opponent has no need to prevent some transitions from happening (by assigning them a zero probability mass). On the contrary, its aim should be to allow all possible transitions in a view to give more ways of getting to a dead-end.

**How to Obtain the Lists —** Previous section has shown how to use minimal lists of states which cannot be forbidden simultaneously so as to perform the reachability analysis. An essential question that we still have to answer is how to obtain these lists. This is an indirect process as it consists in 1- looking for *maximal* lists of states which *can* be forbidden simultaneously, then in 2- adding a state to turn them into *minimal* lists of states which *cannot* be forbidden simultaneously.

As we have defined the notion of "list of states which *cannot* be forbidden simultaneously", we define the opposite notion of "list of states which *can* be forbidden simultaneously":

$$L^{\odot}_{(s,a)} = \left\{ \begin{array}{l} l \subseteq S \text{ s.t. } \sum_{s' \in S \setminus l} Pr^{\max}_{(s'|s,a)} \geq 1 \text{ and} \\ s' \in l \Rightarrow Pr^{\min}_{(s'|s,a)} = 0 \,\&\, Pr^{\max}_{(s'|s,a)} > 0 \end{array} \right\}.$$

But we only need to consider the subset of these lists which are "maximal":

$$L^{\max\odot}_{(s,a)} = \left\{ \begin{array}{l} l \in L^{\odot}_{(s,a)} \text{ s.t. } \forall l' \in L^{\odot}_{(s,a)} : \\ l \cup l' = l \text{ or } l \cup l' \notin L^{\odot}_{(s,a)} \end{array} \right\}.$$

Indeed, adding any reachable state to such a list turns it into a list from $L^{\oslash}_{(s,a)}$. Obtaining the minimal lists required for the reachability analysis requires then two algorithms:

- one to create $L^{\max\odot}_{(s,a)}$ ($\forall(s,a) \in S \times A$), and
- one to turn any set $L^{\max\odot}_{(s,a)}$ in the corresponding set $L^{\min\oslash}_{(s,a)}$.

**Experiments —** The various algorithms developped to perform the reachability and danger analyses have been developped and tested on several problems (see [Buffet, 2004]). The three main remarks coming from these experiments are the following:

1. In some problems, $Pr(s'|s,a) = \epsilon$ can be sufficient to consider that transition $(s,a) \rightarrow s'$ can be forbidden (because of "attracting" parts of the state space which nearly behave like dead-ends).
2. The analyses require enumerating the whole state-space, whereas this is often not feasible. This is all the more unfortunate that one of (L)RTDP's main advantage is to avoid visiting the complete state-space.
3. The reachability analysis for uncertain SSPs can be very time consuming.

The second point is a major subject for future work, with the idea that we should turn our algorithms into a symbolic analysis. Next section shows how to easily address the third point through a simple preprocessing phase.

## 4 Improved Reachability Analysis

The improved algorithm we propose here is based on the idea that, if the reachability analysis for uncertain SSPs is time consuming, in many cases only a small part of the model requires a special treatment. A lot of information can already be obtained through analyses performed on chosen certain SSPs.

More precisely, we apply the certain reachability and danger analyses on an optimistic and a pessimistic model first, to quickly classify most states. Then, the uncertain algorithms only need to be run on states which remain unclassified. As detailed below, this process can be viewed as lower- and upper-bouding a solution with simple technics before using an exact –but costly– computation.

### 4.1 Upper- and Lower-Bounding Reachability Graphs

The precomputation phases work on two reachability graphs obtained from the original uncertain SSP:

- **the lower-bounding reachability graph $G_{lo}$:** in which $s'$ is reachable from $s$ if and only if there exists an action $a$ such that $Pr^{\min}(s'|s,a) > 0$, and
- **the upper-bounding reachability graph $G^{up}$:** in which $s'$ is reachable from $s$ if and only if there exists an action $a$ such that $Pr^{\max}(s'|s,a) > 0$.

$G_{lo}$ represents all transitions which are certainly valid, and $G^{up}$ represents all transitions which could be valid. Yet, these graphs should not be seen as an "optimistic" and a "pessimistic" graph, as the point of view may differ depending on which analysis is being performed.

### 4.2 Principle

The optimistic, pessimistic and exact-computation phases are the following:

1. **optimistic:**

   (a) use $G^{up}$ to perform a certain reachability analysis and get states which *may be* `reaching` (and subsequently those certainly not-`reaching`), and
   (b) use $G_{lo}$ to perform a danger analysis and get states which *are certainly* `dangerous`.

2. **pessimistic:**
   (a) use $G_{lo}$ to perform a certain reachability analysis and get states which *are certainly* `reaching`, and
   (b) ~~use $G^{up}$ to perform a danger analysis and get states which *may be* `dangerous`.~~ (useless step)
3. **exact-computation:** To complete the analyses, two graphs must be designed which embed states not yet certainly `reaching` (or `dangerous`) and their direct children. Then can be performed:
   (a) the construction of the required AND-OR graph,
   (b) the reachability analysis (starting with states known to potentially reach a goal), and
   (c) the danger analysis (starting with states known to be trapped).

Here, one could say that the planner is optimistic when the opponent is pessimistic (and conversely), what explains the inverted use of $G_{lo}$ and $G^{up}$ with the reachability and danger analyses. The former tells whether the planner has some hope to reach a goal state, and the later tells if the opponent has some hope to definitely avoid a goal state.

A useful implementation detail is that this complete process requires a three-state logic telling if a `property` is true, false or unknown.

### 4.3 Algorithms' Complexities

Here is a list of the most important parameters with respect to the algorithmic complexities of the various algorithms:

- $|S|$: number of states,
- $|A|$: maximum number of actions ($\max_{s \in S} |A(s)|$),
- $b_a$: maximum branching-factor for a state-action pair,
- $b_p$: maximum "reverse" branching-factor for a state (i.e. maximum number of parents for a state).

With this, we have the following worst-case complexities:

- constructing $Parents(\cdot)$ for a certain SSP: $O(|S|.|A|.b_a)$,
- certain reachability analysis (Alg. 2): $O(|S|.b_p)$, and
- danger analysis (Alg. 3): $O(|S|.b_p)$.

While these three algorithms remain reasonable, the uncertain reachability analysis creates many lists of states (often singletons) and performs various manipulations on them. This easily leads to a high increase in complexity. Due to the number of independent steps in the uncertain reachability analysis, it is a difficult task to give its algorithmic complexity through a formula. A good intuition can be obtained by computing the complexity of the various steps of this complex algorithm, as done in [Buffet, 2004], Appendix A.

The preprocessing quickly determines for most states if they are `reaching` or `dangerous`. This results in largely reducing the number of unidentified states which require an uncertain reachability analysis, therefore cutting down the complexity of this last algorithm.

## 5 Experiments

**Problems —** Experiments have been conducted on two different problems:

• One is the **mountain-car** problem as defined in [Sutton and Barto, 1998]: starting from the bottom of a valley, a car has to get enough momentum to reach the top of a mountain (see Fig. 3). The same dynamics as described in the mountain car software[4] have been employed, with the only difference that the left boundary has been moved from $-1.2$ to $-2.0$, creating a valley in which the car can be trapped. The objective is to minimize the number of time steps to reach goal.

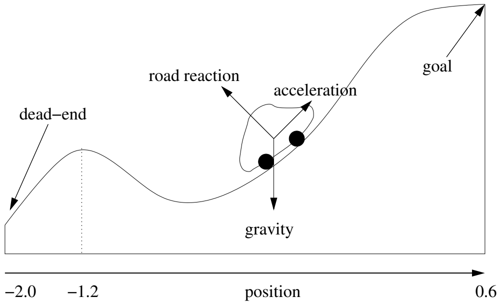


Figure 3: The mountain-car problem with a dead-end.

The continuous state-space is discretized ($32 \times 32$ grid) and the corresponding uncertain model of transitions is obtained by sampling $1000$ transitions from each state-action pair $(s, a)$. For each transition, we computed intervals in which the true model lies with $95\%$ confidence (cf. [Buffet and Aberdeen, 2004] Appendix B.1).

• The other is a **sailing** problem sharing some similarities with the mountain-car task. It's complete description can be found in [Vanderbei, 1996]. Here, the space is discretized to a $10 \times 10$ grid, $\times 8$ wind angles and $\times 8$ possible headings. The system's stochasticity is due to the random changes in the wind's direction. If there is here no true dead-end, rLRTDP is easily trapped in some parts of the state-space, forcing us to consider that a transition with probability $Pr^{\min}(s'|s, a) < 0.01$ can be forbidden. The uncertain model is also learned by drawing $1000$ samples for each state-action pair, using the same $\alpha = 0.05$.

**Results —** As we have just seen, branching factors play a noticeable role. This, and the important number of available actions, may explain the dramatic increase in observed computation time in the sailing problem, as shown on Table 1, column "sailing"-"raw". Yet, the preprocessing obviously helps quickly determining for most states if they are `reaching` or not, hence the huge speed-up observed for each problem's reachability analysis (columns "help"). In both problems,

[4] `http://www.cs.ualberta.ca/~sutton/··· MountainCar/MountainCar.html`

most of the state space is effectively handled through the certain analyses, only a small part depending on “uncertain” dynamics.

| | mountain-car | | sailing | |
|---|---|---|---|---|
| $\lvert S\rvert$ | 1024 | | 6400 | |
| $\lvert A\rvert$ | 2 | | 8 | |
| | raw | help | raw | help |
| Init | 0.7780 | 0.7801 | 5.8647 | 5.8670 |
| Reach$^{\text{ability}}$ | 0.2810 | 0.0277 | 167.7658 | 0.4468 |
| rLRTDP | 10.6862 | 10.6447 | 1.4320 | 0.5442 |

Table 1: Average performance (duration in seconds) obtained with 100 executions for the 3 phases: 1- model `Init`ialization (including the statistical modeling), 2- `Reach`ability analysis and 3- `rLRTDP` itself.
(“raw”= “no preprocessing”, “help”= “with preprocessing”)

A surprising observation is that rLRTDP is much faster on the sailing problem when a preprocessing phase is used. This may be linked to the fact that the computer has no problem handling memory in this case, what may slow down rLRTDP if used after the expensive reachability analysis on a complete uncertain graph. The same experiment on a lake of $4 \times 4$ instead of $10 \times 10$ shows little difference between both cases: without (0.0161s) and with (0.0197s) preprocessing.

## 6 Conclusion

The goal reachability checked through the algorithm presented here is an essential tool for robust RTDP [Buffet and Aberdeen, 2004; 2005]. This paper briefly describes how to perform reachability and danger analyses in certain and uncertain SSPs, and explains how the analyses for uncertain SSPs can be speeded up through a simple preprocessing phase.

An open question is how to use the information obtained through the reachability analysis. If one does not want to forbid states which are `reaching` and `dangerous`, the cost function is not sufficient for decision-making and a new (non-classical ?) preference criterion has to be introduced.

The main remaining issue is then how to avoid enumerating the complete state-space. In a structured domain, as in temporal planning, it would be of great interest to conduct a symbolic analysis, as it has been done for other purposes for Finite State Automata [Coudert *et al.*, 1990] by using BDDs [Bryant, 1985]. The major problem should be the algorithm producing the minimal lists in $L^{\min\emptyset}_{(s,a)}$, what would enable a symbolic characterization of the AND-OR graph.

Finally, it is important to notice that the core of the algorithms presented in this document is not specific to decision-making, but rather to certain and uncertain Markov chains (with end states). It would be simple to rewrite the various procedures to that end, as Markov chains could be described as SSPs with no costs and a single available action per state.

**Acknowledgments**
National ICT Australia is funded by the Australian Government. This work was also supported by the Australian Defence Science and Technology Organisation.

## References

[Bagnell *et al.*, 2001] J.A. Bagnell, A. Y. Ng, and J. Schneider. Solving uncertain markov decision problems. Technical Report CMU-RI-TR-01-25, Robotics Institute, Carnegie Mellon U., 2001.

[Barto *et al.*, 1995] A.G. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72, 1995.

[Bellman, 1957] R. Bellman. *Dynamic Programming*. Princeton U. Press, Princeton, New-Jersey, 1957.

[Bertsekas and Tsitsiklis, 1996] D.P. Bertsekas and J.N. Tsitsiklis. *Neurodynamic Programming*. Athena Scientific, 1996.

[Bryant, 1985] R.E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *ACM/IEEE Design Automation*, pages 688–694, 1985.

[Buffet and Aberdeen, 2004] O. Buffet and D. Aberdeen. Planning with robust (l)rtdp. Technical report, National ICT Australia, 2004.

[Buffet and Aberdeen, 2005] O. Buffet and D. Aberdeen. Robust planning with (l)rtdp. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI’05)*, 2005.

[Buffet, 2004] O. Buffet. Robust (l)rtdp: Reachability analysis. Technical report, National ICT Australia, 2004.

[Coudert *et al.*, 1990] O. Coudert, J.-C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *Proc. of the Workshop on Computer-Aided Verification*, 1990.

[Givan *et al.*, 2000] R. Givan, S. Leach, and T. Dean. Bounded parameter markov decision processes. *Artificial Intelligence*, 122(1-2):71–109, 2000.

[Hosaka *et al.*, 2001] M. Hosaka, M. Horiguchi, and M. Kurano. Controlled markov set-chains under average criteria. *Applied Mathematics and Computation*, 120(1-3):195–209, 2001.

[Munos, 2001] R. Munos. Efficient resources allocation for markov decision processes. In *Advances in Neural Information Processing Systems 13 (NIPS’01)*, 2001.

[Nilim and Ghaoui, 2004] A. Nilim and L. El Ghaoui. Robustness in markov decision problems with uncertain transition matrices. In *Advances in Neural Information Processing Systems 16 (NIPS’03)*, 2004.

[Strehl and Littman, 2004] A. L. Strehl and M. L. Littman. An empirical evaluation of interval estimation for markov decision processes. In *Proc. of the 16th Int. Conf. on Tools with Artificial Intelligence (ICTAI’04)*, 2004.

[Sutton and Barto, 1998] R. Sutton and G. Barto. *Reinforcement Learning: an introduction*. Bradford Book, MIT Press, Cambridge, MA, 1998.

[Vanderbei, 1996] Robert J. Vanderbei. Optimal sailing strategies, statistics and operations research program, 1996. U. of Princeton, http://www.sor.princeton.edu/˜rvdb/sail/sail.html.

# Putting Olfaction into Action: Anchoring Symbols to Sensor Data Using Olfaction and Planning

**Amy Loutfi, Silvia Coradeschi, Lars Karlsson and Mathias Broxvall**
Center for Applied Autonomous Sensor Systems
Örebro University
Örebro, Sweden 701-82
www.aass.oru.se

## Abstract

Olfaction is a challenging new sensing modality for intelligent systems. With the emergence of electronic noses (e-noses) it is now possible to train a system to detect and recognise a range of different odours. In this work, we integrate the electronic nose on a multi-sensing mobile robotic platform. We plan for perceptual actions and examine when, how and where the e-nose should be activated. Finally, experiments are performed on a mobile robot equipped with an e-nose together with a variety of sensors and used for object detection.

## 1 Introduction

Mobile robots are becoming equipped with more diverse and numerous sensing modalities. Still, however, many of these modalities focus on obtaining the structural properties of an environment using for example, cameras, sonars and lasers. In object detection applications, objects may be discriminated more than just on their structural properties. One example is the property of smell. With the development of electronic olfaction and small electronic noses, gas sensors can be effectively integrated on mobile platforms. Furthermore such an integration could be of benefit to a number of robotics related applications for rescue robots, home-care robots and exploratory robots.

Electronic noses (e-noses) offer the potential to systematically quantify odours and this ability is attractive to many research and industrial applications. E-noses have also become commercially available, facilitating the integration of artificial olfaction in the AI community. Over the past two decades e-noses have been used in the detection of substances in the context of quality control in the food industry, detection of toxins, and discrimination of odours [Persaud and Dodd, 1982]. In this work we present a new kind of application where e-noses are added in the context of a larger sensing system, including vision, and planning. The e-nose is used for detecting objects where the odour characteristic is considered in a symbolic context as an additional property of the object. We focus in particular on the use of planning to determine the sensory actions needed to correctly collect the olfactory data.

Using an electronic nose in the context of object detection has several benefits. Firstly, descriptions of objects can include an odour and the respective recognition can be based on smell. For example, in the detection of “a cup of ethanol”, vision may serve to detect certain properties of the cup or even the colour of the substance but the actual substance of ethanol is best determined through chemical analysis. Another benefit emerges when two similar objects create an ambiguous case for the decision-making system. The property of smell can be used to disambiguate and distinguish between these objects. Even in non-ambiguous cases, acquiring the odour characteristic can be used to confirm the belief that the desired object has in fact been found. Finally, in the case of reacquiring an object, the odour property can be determined, stored, and reused in order to find the same object again.

A naïve approach to implementing odour recognition in the experiments mentioned above, may be to use an electronic nose as a passive sensor i.e., constantly smelling the environment and associating the odour characteristic to the object in focus. There are however, several problems to passive sensing in this case. First, there is the conceptual problem that questions the validity of associating a dispersed odour in the air to an object physically located at a distance from the actual point of detection. There is also the practical problems of the sensing mechanism that include long sampling time (2-5 minutes), high power consumption (pumps and heaters), and long processing time for the multivariate sensor data. In a real time application that considers a mobile platform with multiple sensing modalities, the electronic nose is an expensive sensor. For these reasons, the electronic nose in this work is used instead as an active sensor that is explicitly called upon inside a complex decision making system.

The work in this paper presents an overview of a system that receives as input a symbolic description of an object and locates that object in a complex environment. The system has different sensing modalities available (in this case olfaction and vision) and should determine when to use these modalities in order to most effectively reach the goal. The system also reasons about its perceptions on a symbolic level. For this reason, a planner and an anchoring module are included within the system architecture. The purpose of the planner is to treat ambiguities, generate plans, call the execution of appropriate behaviours (that may include the command to smell an object) and determine when the goal has been reached. The anchoring component is important since it creates and maintains the connection between the symbols denoting objects at the symbolic level and the perceived physical objects [Coradeschi and Saffiotti, 2000].

The paper begins with a description of related works in Section 2. In Section 3, an overview of the system and its components is given. In Section 4, different scenarios such as disambiguating between objects and reacquiring objects are tested using vision together with olfaction on a robotic platform.

## 2 Related Work

To the knowledge of the authors, very little if not any attempts have been made to integrate an electronic nose on a multi-sensing mobile platform that specifically executes object recognition tasks. Although significant contributions have been made in the areas on odour source localization using mobile robots, it should be emphasized that this topic is outside the focus of this paper. Despite the absence of similar works however, there is well-developed research from both the electronic nose community and the AI community that facilitates the integration of odour recognition into intelligent systems.

In [Gardner and Bartlett, 1999] a general definition of an electronic nose includes both an array of gas sensors of partial selectivity and a respective pattern recognition process to detect simple or complex odours. Both a variety of sensing technologies (metal oxide semiconductor, conducting polymers, acoustic wave devices and fibre-optic sensors) as well as pattern recognition techniques have been applied in research, industrial and commercial domains. The most common of the pattern recognition used has been artificial neural networks, which are trained on odour categories. Although ANN's have provided good results in applications with limited amount of odours categories as shown in [Keller *et al.*, 1996], using black-box classification fails to address the problem of representing the knowledge of odour categories in order to classify a larger spectrum of odours. A study on the meaning of categorisation by Dubois [2000] has attributed the absence of fixed standards in odour classification to the lack of correlation between the chemical composition of an odour and the common name that refers to it. Consequently, chemical models such as Dravniek's [2000] character profiles or Amoore's [1965] primary odour tables, have not adequately represented in a generally accepted manner odour categorisations due to the large possibility of odour categories and descriptions. Thus, in recent years, there has been a movement to treat e-nose data in a more human-like manner by either relying on expert knowledge using fuzzy-based logic in tailored applications [Lazzerini *et al.*, 2001] or using explicit symbolic descriptions in order to relate odours categories to one another [**?**].

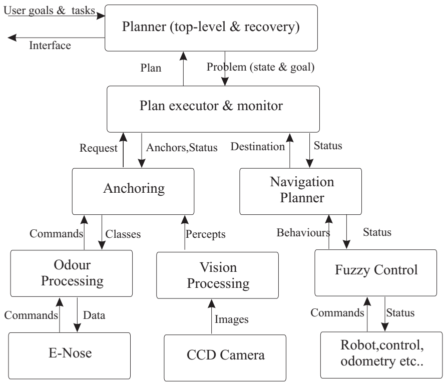


Figure 1: Overview of the robotic complete system which uses an anchoring module. On the left side of the arrows information is flowing downward and on the right side information is flowing upward.

Aside from performing navigation by smell [Lilienthal *et al.*, 2001], little work has explored the possibility of using an electronic nose together with other sensors on robotic platforms. Some studies such as [Sundic *et al.*, 2000] have integrated e-noses on multi-sensing stationary platforms for the purpose of sensor fusion, but have not considered an application with intelligent or autonomous systems. Conversely, many intelligent or autonomous systems that consider planning for perceptual actions [Barrouil *et al.*, 1998; Kovacic *et al.*, 1998; Wasson *et al.*, 1998] and perceptual anchoring [Coradeschi and Saffiotti, 2003], have focussed primarily on vision-based sensors.

## 3 An overview of the complete system

The components of the complete system used in our experiments and their interrelations are shown in Figure 1. Significant attention is placed on the olfactory module with a description of the sensor operation and data processing. Further details regarding the other components of the system can be found in their respective references.

### 3.1 Planner

The planner, PTLplan, is a conditional progressive planner [Karlsson, 2001]. It has the capacity to:

- reason about incomplete and uncertain information: it might be unknown whether a cup contains ethanol or some other substance.
- reason about informative actions: by smelling the cup, one might determine whether it contains ethanol or something else.
- generate plans that contain conditional branches: if the cup contains ethanol, pick it up; otherwise, check the other cup.

PTLplan is used on two different levels: for constructing the plan for achieving the current main goal, and for constructing repair plans in case problems occur while the main plan is executed. The repair planning facility mainly deals with actions and plans for disambiguating an ambiguous situation, e.g. by finding out which of the two cups is the one that contains ethanol [**?**].

The planner functions by searching in a space of belief states. A belief state represents the agent's incomplete and uncertain knowledge about the world at some point in time. A belief state can be considered to represent a set of hypotheses about the actual state of the world. The planner can reason about perceptive actions and these actions have the effect that the agents makes observations that may help it to distinguish between different hypotheses. Each different observation will result in a separate new and typically smaller belief state, and in each belief state the robot will know more than before. To summarize, the planner searches for plans that maximize the probability of success, as well as mimimize the cost. Cost can be defined simply in terms of the number of steps, or by associating each action with a numerical cost.

### 3.2 Plan executor

The plan executor takes the individual actions of the plan and translates them into tasks that can be executed by the control system (the Thinking Cap). These tasks typically consist of navigation from one locality or object to another. Planning actions might also be translated into requests to the anchoring system to find or reacquire an object that is relevant to the plan, either in order to act on that object (e.g. moving towards it) or simply to acquire more information about the object.

The plan executor also reacts when the execution of an action fails, e.g. due to ambiguities when it tries to anchor an object. In such cases, the repair planning facility is invoked.

### 3.3 Anchoring Module

The anchoring module creates and maintains the connection between symbols referring to physical objects and sensor data provided by the vision and olfaction sensors. The symbol-data correspondence is represented by a data structure called an anchor, that includes pointers to both the symbol and the sensor data connected to it. The anchors also maintain a set of properties that are useful to re-identify an object e.g., colour and position. These properties can also be used as input to the control routines. Different functionalities are included in the anchoring modules. In this work, two functionalities in particular are used.

**Find** is used to link the symbol e.g., "cup-22" to a percept such as a region in an image that matches the description "red cup containing ethanol". The output of Find is an anchor that contains properties such as the *(x,y)* position or the odour of the cup.

**Reacquire** is used to update the properties of an existing anchor. This may be useful if the object goes out of view or a period of time elapses resulting in a change of object properties (e.g., chemical characteristic).

The anchoring functionalities are typically called by the planner via the plan executor. To be able to execute actions referring to an object, the planner interacts with the anchoring module by referring to objects using a symbolic name and a description expressed in terms of predicates. For instance, we can execute the command "move-near cup-25" where "cup-25" is described as a "green cup".

Since all properties of an object are not always accessible, the anchoring module also considers cases of *partial matchings*. We consider a matching between a description and the perceptual properties of an object partial when all perceived properties of the object match the description, but there still remains properties in the description that have not been perceived. This is a typical case for olfaction that requires that the sensors are close to the odour source for detection.

The anchoring module is also able to determine whether objects have been previously perceived, so as to not create new anchors for existing objects. Ambiguous cases such as when two objects partially match a given description, and failure to find an object are detected by the module and dealt with at the planner level.

A more detailed description of the anchoring module and its functionalities can be found in [Coradeschi and Saffiotti, 2000].

### 3.4 Thinking Cap

In this system, execution monitoring on a mobile robot is controlled by a hybrid architecture evolved from [Saffiotti *et al.*, 1995] called the Thinking Cap. The Thinking Cap (TC) consists of a fuzzy behaviour-based controller, and a navigation planner. In order to achieve a goal the planner selects a number of behaviours to be executed in different situations. Depending on the current situation, the different behaviours are activated to different degrees.

### 3.5 Vision Module

In addition to the sonars used by Thinking Cap to navigate and detect obstacles, the system also uses vision to perceive objects. This is done by continuously receiving images from a CCD camera connected to the robot and using standard image recognition techniques for image segmentation. The segmented images are used for recognising a number of predetermined classes of objects and properties such as shape, colour and relative position. The resulting classified objects are delivered to the rest of the system at approximately 1fps.

The result of these classifications are collected over time and presented to the planning and anchoring system. The system tries to represent objects so that they are persistent over time but due to uncertainties in our sensing this is not always possible and ambiguities which has to be dealt with at the planning level may arise.

In order to maintain and predict sensed object currently outside the camera's viewpoint an odometry based localisation is used. As long as our movements are limited this makes it easy for the system to reacquire objects based on their stored position, if however the objects move or if accumulation of odometry errors is large this might lead to reacquisition ambiguities which can only be resolved using other sensors.

### 3.6 Olfactory Module

The olfactory module consists of a commercially available electronic nose. This e-nose contains 32 thin-film carbon-black conducting polymer sensors of variable selectivity. Each sensor consists of two electrical leads on an aluminium substrate. Thin films are deposited across the leads creating

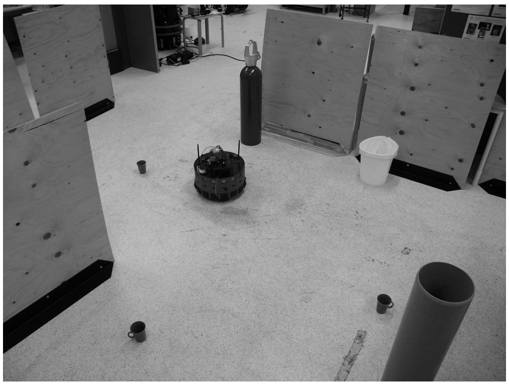


Figure 2: A scenario where the robot discriminates four visually similar green cup objects based on their smell property

a chemiresistor. Upon exposure to an analyte, a polymer matrix absorbs the analyte and increases in volume. This increase in volume is reflected in a increase in resistance across the sensor. Each polymer in the array is unique and designed to absorb gases to different degrees, creating a pattern of responses across the array. The array of sensors are contained in a portable unit also consisting of pumps, valves and filters that are required to expose the sensor array to a vapour gas.

The sampling of an odour occurs in three different phases. The first phase is a baseline purge, where the sensors are exposed to a steady state condition, for example the air in the room. The duration of the purge is 180 seconds. The second phase is a sampling cycle where a valve is switched to allow vapours from a sampling inlet to come into contact with the sensing array. The duration of this cycle is 60 seconds. Finally, a sequence of purging cycles is used to remove the sampled odour from the unit and restore the sensor response to the baseline values. A total of 30 seconds are dedicated to purging the unit however the time for full recovery of the sensors may vary according to the odour being sampled.

The signals are gathered in a response vector where each sensor's reaction is represented by the fractional response of the reaction phase to the baseline. The response vectors are then normalised using a simple weighting method and autoscaled. Classification of new odours is performed by first collecting a series of training data. With this type of data, the authors have used an unsupervised technique based on fuzzy clustering such as that presented in [**?**]. However, for the the experiments described below and in order to optimise computation time in a real-time environment, a minimum distance classifier was implemented. The result from this classifier provides a linguistic name which refers to the class to which the unknown odour belongs. Later work intends to explore the ability to output not only the class name but also a degree of membership, which could then be used in the planning process.

## 4 Experiments

The experimental setup consists of a Magellan Pro Research robot, called Pippi, equipped with a CCD camera, sonars, infrared sensors, compass and a Cyranose 320 electronic nose.

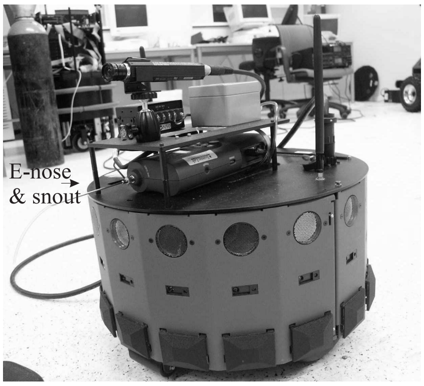
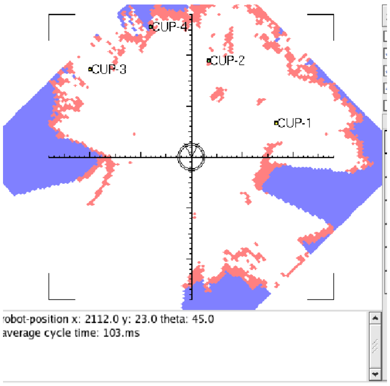


Figure 3: A closer view of the Pippi and the electronic nose.



```
robot-position x: 2112.0 y: 23.0 theta: 45.0
average cycle time: 103.ms
```

Figure 4: The local perceptual space of Pippi given 4 similar cups and obstacles. Pippi is located in the center of the space.

A snout protrudes from the robot so that sampling of an object can be done easily see Figure 3. The nose has been previously trained on a number of different substances including those used in the first experiment.

### 4.1 Disambiguiting Objects

In this scenario, we consider a situation where Pippi is in a room where there are several different objects present as shown in Figure 4. On the floor of the room are visually identical green cups. Given the request to identify an object such as cup of ethanol, the planner in this case activates a Find functionality creating anchors for the matching objects. An example of the call to Find may look like:

(find c1 ((shape = cup) (smell = ethanol)))

The anchoring module examines the percepts sent from the vision module, and finds several percepts that match the indicated shape of c1. The percepts are considered as partially matching the description as the smell still remains to be perceived. The anchoring module classifies this as an ambiguous situation and creates several candidate anchors, one for each cup.

The repair planner is invoked, and considers the properties of the requested cup c1 and of the new percepts. It finds that c1 is expected to smell ethanol, but that there is no smell associated with either anchor (the case of two identical cups). It then automatically generates an initial state consisting of three hypotheses: that only anchor-1 smells ethanol and is the target for anchoring c1; the corresponding for the other anchors; and finally that no anchors smell ethanol. This initial state, and the goal to either have c1 anchored or to determine it cannot be anchored, is given as input to the planning algorithm, which then produces the following conditional plan:

```
((move-near anchor-1) (smell-obj anchor-1)
 (cond
    ((smell anchor-1 = ethanol) (anchor c1 anchor-1) :success)
    ((smell anchor-1 = not-ethanol) (move-near anchor-2)
     (smell-obj anchor-2)
     (cond
        ((smell anchor-2 = ethanol)
         (anchor c1 anchor-2) :success)
        ((smell anchor-2 = not-ethanol)
         (anchor c1 :fail) :success)))))
```

Note that each conditional branch ends with a decision to anchor c1 to one of the candidates, or to none (:fail).

The plan is executed and Pippi moves close enough to the object denoted by anchor-1 so that the snout of the nose is within reasonable sampling proximity (see Figure 4). The planner sends to the anchoring module a request to smell the object. The anchoring module receives the command and requests the sensing data from the e-nose. Now the e-nose invokes a 4-minute sampling procedure where the result is a classification based on the comparison with the trained values. The classification is sent and anchor-1 is updated. An indication that the smelling process has finished sampling is sent to the planner from the anchoring module. If the smell was found to be ethanol, the planner decides to associate the symbol c1 to anchor-1. If the object denoted by anchor-1 did not smell like ethanol then the planner would proceed to approach the second object. If neither of the objects denoted by the anchors return the desired classification, the planner registers that no anchors for c1 have been found.

A number of configurations of the above scenario were tested, where the number of cups were 2, 3, 4 or 5, each cup with a different content. The contents of the cups were one of the following: Ethanol, Hexanal, 3-Hexanol, Octanol, or Linalool. These substances are part of an ASTM atlas of odour descriptions [Dravnieks, 2000] whose characters are best described as alcoholic, sour, woody, oily and fragrant respectively.

Table 1 summarizes the results from different configurations involving different numbers of candidate objects. Note that in order to execute the smell action Pippi needs to move close to the objects. As a result, errors may arise from either the olfaction module (misclassification of odours) and/or the vision module (accumulated odometry errors cause the anchoring module to lose track of an object). The table also provides information regarding the source of failures in the unsuccessful cases.

Analysis of the results shows that visual failures provoke olfactory failures. This is due to the fact that the e-nose performs best when close to an object. Depending on the odour (rate of vapourization), the distance to accurately recognize the odour range is between 5 cm and 23 cm. Most visual failures are due to the odometry and as the number of candidate objects increase, Pippi needs to move a longer distance and a larger odometry error is accumulated. There are cases in which the e-nose misclassifies the odour independently of visual failures. These misclassification errors slightly increase when the number of different odours increase. The source of this error can be the e-nose's inability to discriminate between classes of odours. However, in our case the error was actually due to the sensing parameters given in the sampling process. In particular, when two cups were separated by a short distance there was an inadequate recovery time between "sniffs" and this resulted in a misclassification of samples. This recovery time depends on the type of odour being sampled.

Table 1: Experimental results from Disambiguating visually similar objects

| No.of Odours | Trials | Olfactory Failures (Vision, Odometry) | Olfactory Failures (Classification) |
|---|---|---|---|
| 2 | 11 | 18% | 0 % |
| 3 | 15 | 20% | 0 % |
| 4 | 21 | 19% | 4.7% |
| 5 | 25 | 16% | 8% |

## 4.2 Reacquiring Objects

The purpose with this experiment is to show a different application of an electronic nose in which the e-nose is used to acquire the specific odour of an object. Assuming that the odour characteristic is a unique property of that object, this information can be used to reacquire the object again. In this scenario, the robot is in a room and a cup is located on the floor. Pippi is first given the task to find the cup and then to acquire its odour. Pippi first looks for the cup, finds it, and moves close to it. It then requests the e-nose to start sampling. In this case, however, the objective is not to recognise the smell but instead acquire it in order to use it for identifying the cup at future occasions. The e-nose samples the odour and stores the sensor signals as a new pattern within the training depository. A new name is generated for the odour. Finally, the anchoring module stores the information that this particular cup has the acquired smell.

The robot then wanders throughout the room. Meanwhile, an additional cup of similar shape and colour is added and the original cup is displaced so that it cannot be recognised by its position. The planner then requests the anchoring module to reacquire the original cup. Two possible candidates are found, the original cup and the new cup, and the planner is informed that there is an ambiguous situation. A plan is created consisting of first going to one of the cups and smelling it, if the classification of the odour matches the one that was stored during the first acquire then the plan succeeds. Otherwise, the second cup is checked. In our experiment, Pippi successfully reacquired the cup.

## 5 Conclusions

Olfaction is a valued sense in humans, and robotic systems, which interact with humans, and/or execute human-like tasks also benefit from the ability to perceive and recognise odours. While previously gas sensors were difficult to use and needed certain expertise to successfully implement odour recognition, commercial products have now made it possible to successfully employ electronic olfaction in new domains. One such domain is intelligent systems that rely on multi-sensing processes to perform autonomous tasks. The integration of electronic olfaction presents interesting challenges with respect to the use of AI techniques in robotic platforms. Some of these challenges are due to the properties of the sensing mechanism, such as long sampling time, and close proximity required for smelling an object.

In this work, we show how an electronic nose could be successfully used as a tool for the recognition of objects. We also show that successful integration requires that the e-nose is explicitly called within the system at the appropriate occasion. Planning is essential for this task. The result is a system capable of using odour recognition to disambiguate between visually similar objects of different odour property and reacquire them at a later time.

**Acknowledgments** This work has been supported by: Vetenskapsrådet, and by ETRI (Electronics and Telecommunications Research Institute, Korea) through the project ”Embedded Component Technology and Standardization for URC(2004-2008)”.

## References

[Amoore, 1965] J. Amoore. Psychophysics of odor. In *Cold Spring Harbor Symposia in Quantitative Biology*, volume 30, pages 623–637, 1965.

[Barrouil *et al.*, 1998] C. Barrouil, C. Castel, P. Fabiani, R. Mampey, P. Secchi, and C. Tessier. Perception strategy for a surveillance system. In *Proc. of ECAI*, pages 627–631, 1998.

[Coradeschi and Saffiotti, 2000] S. Coradeschi and A. Saffiotti. Anchoring symbols to sensor data: preliminary report. In *Proc. of the 17th American Association for Artificial Intelligence Conf. (AAAI)*, pages 129–135, 2000.

[Coradeschi and Saffiotti, 2003] S. Coradeschi and A. Saffiotti. An introduction to the anchoring problem. *Robotics and Autonomous Systems*, 43(2-3):85–96, 2003.

[Dravnieks, 2000] A. Dravnieks. *Atlas of Odor Character profiles (ASTM Data Series Publication DS 61)*. American Society for Testing, USA, 2000.

[Dubois, 2000] D. Dubois. Categories as acts of meaning: The case of categories in olfaction and audition. *Cognitive Science Quaterly*, 1:35–68, 2000.

[Gardner and Bartlett, 1999] J. Gardner and P. Bartlett. *Electronic Noses, Principles and Applications*. Oxford University Press, New York, NY, USA, 1999.

[Karlsson, 2001] L. Karlsson. Conditional progressive planning under uncertainty. In *Proc. of the 17th Int. Joint Conferences on Artificial Intelligence (IJCAI)*, pages 431–438, 2001.

[Keller *et al.*, 1996] P. Keller, L. Kangas, L. Liden, S. Hashem, and R. Kouzes. Electronic noses and their applications. In *World Congress on Neural Networks (WCNN)*, pages 928–931, San Diego, CA, USA, 1996.

[Kovacic *et al.*, 1998] S. Kovacic, A. Leonardis, and F. Pernus. Planning sequences of views for 3-D object recognition and pose determination. *Pattern Recognition*, 31:1407–1417, 1998.

[Lazzerini *et al.*, 2001] B. Lazzerini, A. Maggiore, and F. Marcelloni. Fros: a fuzzy logic-based recogniser of olfactory signals. *Pattern Recognition*, 34(11):2215–2226, 2001.

[Lilienthal *et al.*, 2001] A. Lilienthal, A. Zell, M. Wandel, and U. Weimar. Sensing odour sources in indoor environments without a constant airflow by a mobile robot. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 4005–4010, Seoul, South Korea, 2001.

[Persaud and Dodd, 1982] K. Persaud and G. Dodd. Analysis of discrimination mechanisms of the mammalian olfactory system using a model nose. *Nature*, 299:352–355, 1982.

[Saffiotti *et al.*, 1995] A. Saffiotti, K. Konolige, and E. H. Ruspini. A multivalued-logic approach to integrating planning and control. *Artificial Intelligence*, 76(1-2):481–526, 1995.

[Sundic *et al.*, 2000] T. Sundic, S. Marco, A. Perera, A. Pardo, J. Samitier, and P.Wide. Potato creams recognition from electronic nose and tongue signals: feature extraction/selection and r.b.f neural networks classifiers. In *Proc. of the IEEE 5th Seminar on Neural Network Applications in Electrical Engineering (NEUREL)*, pages 69–74, 2000.

[Wasson *et al.*, 1998] G. Wasson, D. Kortenkamp, and E. Huber. Integrating active perception with an autonomous robot architecture. In *Proc. of the 2nd Int. Conf. on Autonomous Agents (Agents)*, pages 325–331, 1998.

# Using LTL Assumptions to Generate Safe Plans for Partially Known Domains

**Alexandre Albore, Piergiorgio Bertoli**
ITC-IRST
Via Sommarive 18, 38050 Povo, Trento, Italy
{albore,bertoli}@irst.itc.it

## Abstract

Planning for partially known domains is an extremely demanding task. However, it is often possible to formulate assumptions over the expected dynamics of the domain; these can be used to effectively cut the search, dramatically improving plan generation. In turn, the execution of assumption-based plans must be monitored to prevent run-time failures that may happen if assumptions turn out to be untrue, and to replan in that case. In this paper, we use an expressive temporal logics, LTL ([Emerson, 1990]), to describe assumptions, and we provide two main contributions. First, we describe an effective, symbolic forward-chaining mechanism to build (conditional) assumption-based plans for partially known domains. Second, we constrain the algorithm to generate *safe* plans, i.e. plans guaranteeing that, during their execution, the monitor will be able to unambiguously distinguish whether the domain behavior is one of those planned for or not. This is crucial to inhibit any chance of useless replanning episodes. We experimentally show that exploiting LTL assumptions highly improves the efficiency of plan generation, and that enforcing safety improves plan execution, inhibiting useless, expensive replanning episodes without significantly affecting plan generation.

## 1 Introduction

Many realistic scenarios require the ability to generate plans for domains whose behavior is not completely known a priori; on top of this, the internal status of these domains is often only partially observable. Planning under these premises is an extremely challenging task: only rarely, in this situation, *strong* solutions that guarantee reaching a given goal exist, and finding them requires traversing a huge search space. However, in many cases, it is possible to express reasonable assumptions over the expected dynamics of the domain, e.g. by identifying "nominal" behaviors; using these assumptions to constrain the search may greatly ease the planning task, allowing the efficient construction of assumption-based solutions. Of course, assumptions taken when generating a plan may turn out to be incorrect when executing it. For this reason, assumption-based plans must be executed within reactive architectures such as [Muscettola *et al.*, 1998; Myers and Wilkins, 1998], where a monitoring component traces the status of the domain, in order to replan when an unexpected behavior has compromised the success of the plan. However, due to the incomplete run-time knowledge on the domain state, it may not be possible for a monitor to establish unambiguously whether the domain status is evolving as expected or not; then, replanning must occur whenever a dangerous state *may* have been reached. But if the actual domain state is one of those planned for, replanning is unnecessary and undesired. These situations can only be avoided if states not planned for can unambiguously be identified at plan execution time; in turn, whether this is possible crucially depends on the actions performed by the plan. In this paper, we model partially known domains as partially observable, nondeterministic finite state machines, and we consider an expressive language that provides us with the key ability to specify assumptions over the domain dynamics, called linear temporal logics (LTL, [Emerson, 1990]). In this framework, we provide two main contributions. First, we provide an effective, symbolic mechanism to constrain forward and-or search to generate (conditional) LTL assumption-based solutions for nondeterministic, partially observable domains. Second, we further constrain the search to obtain *safe* LTL assumption-based solutions, i.e. plans that not only guarantee that the goal is reached when the given assumption holds, but also guarantee that, during their execution, the monitor will be able to unambiguously distinguish whether the current domain status has been planned for or not. In this way, no unneeded plan abortion (and consequent replanning) may be triggered by the monitor. We experimentally show that generating LTL assumption-based solutions can be dramatically more effective than generating strong plans, and that enforcing safety can highly improve plan execution, since it inhibits costly and useless replanning episodes without significantly affecting plan generation.

The paper is organized as follows. Section 2 provides the basic background notions. Section 3 introduces LTL assumptions, and defines LTL assumption-based solutions for a planning problem. Section 4 provides the key notion of safe (LTL) assumption-based plan. Section 5 describes a forward-chaining procedure to generate LTL assumption-based solutions, and in particular safe ones, using symbolic representation techniques. Section 6 provides an experimental evaluation of the approach. Section 7 draws conclusions and illustrates future work directions.

## 2 Domain, Goals, Plans

To represent uncertainty over the nature and dynamics of a planning domain, we model it as a nondeterministic, partially observable finite state machine. Following [Bertoli *et al.*, 2001], we allow for initial state uncertainty, non-deterministic action outcomes, and partial observability with noisy sensing:

**Definition 1 (Planning domain).** *A* non-deterministic planning domain with partial observability *is a 6-tuple* $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{U}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$*, where:*

- $\mathcal{S}$ *is the set of* states.
- $\mathcal{A}$ *is the set of* actions.
- $\mathcal{U}$ *is the set of* observations.
- $\mathcal{I} \subseteq \mathcal{S}$ *is the set of* initial states*; we require* $\mathcal{I} \neq \emptyset$*.*
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow 2^{\mathcal{S}}$ *is the* transition function*; it associates with each current state* $s \in \mathcal{S}$ *and with each action* $a \in \mathcal{A}$ *the set* $\mathcal{T}(s,a) \subseteq \mathcal{S}$ *of next states.*
- $\mathcal{X} : \mathcal{S} \rightarrow 2^{\mathcal{U}}$ *is the* observation function*; it associates with each state s the set of possible observations* $\mathcal{X}(s) \subseteq \mathcal{U}$*. Some observation must be possible for any given state:* $\mathcal{X}(s) \neq \emptyset$*.*

We indicate with $[\![ o ]\!]$ the set of states compatible with the observation $o$: $[\![ o ]\!] = \{s \in \mathcal{S} : o \in \mathcal{X}(s)\}$. We say that action $\alpha$ is *executable* on state $s$ iff $\mathcal{T}(s,\alpha) \neq \emptyset$, and we denote with $\alpha(s) = \{s' : s' \in \mathcal{T}(s,\alpha)\}$ its execution. An action is executable on a set of states $B$ (also called a *belief*) iff it is executable on every state of the set; its execution is denoted $\alpha(B) = \{s' : s' \in \mathcal{T}(s,\alpha),\ s \in B\}$.

Moreover, we assume the existence of a set of basic propositions $\mathcal{P}$, and of a labeling of states with the set of propositions holding on them. We denote with $\mathcal{P}rop$ the propositional formulæ over $\mathcal{P}$, and with $[\![ \varphi ]\!]$ the states satisfying a formula $\varphi \in \mathcal{P}rop$.

A planning problem is a pair $\langle \mathcal{D}, \mathcal{G} \rangle$, where $\mathcal{G} \subseteq \mathcal{S}$ is a set of goal states. We solve such problems considering conditional plans that may branch on the basis of observations:

**Definition 2 (Conditional Plan).** *The set of conditional plans* $\Pi$ *for a domain* $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{U}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$ *is the minimal set such that:*

- $\varepsilon \in \Pi$*;*
- *if* $\alpha \in \mathcal{A}$ *and* $\pi \in \Pi$*, then* $\alpha \circ \pi \in \Pi$*;*
- *if* $o \in \mathcal{U}$*, and* $\pi_1, \pi_2 \in \Pi$*, then* if $o$ then $\pi_1$ else $\pi_2 \in \Pi$.

Intuitively, $\varepsilon$ is the empty plan, $\alpha \circ \pi$ indicates that action $\alpha$ has to be executed before plan $\pi$, and *if o then* $\pi_1$ *else* $\pi_2$ indicates that either $\pi_1$ or $\pi_2$ must be executed, depending on whether the observation $o$ holds. A plan is a finite state machine that controls the domain by executing synchronously with it: at each step, the plan evolves on the basis of the current observation and of its internal state, producing an action which, in turn, makes the domain evolve and produce a new observation.

An execution of a conditional plan can be described as a trace, i.e. a sequence[1] of traversed domain states and associated observations, connected by the actions of the plan.

Since a plan may attempt a non-executable action, we have to distinguish failure traces from non failure traces:

**Definition 3 (Traces of a plan).** *A trace of a plan is a sequence* $[s^0, o^0, \alpha^0, \ldots, s^n, o^n, End]$*, where* $s^i, o^i$ *are the domain state and observation at step i of plan execution (i.e.,* $s^0 \in \mathcal{I}$*), and* $\alpha^i$ *is the action produced by the plan on the basis of* $o^i$ *(and of the plan's internal state).* End *can either be* Stop*, indicating that the plan has terminated, or* $Fail(\alpha^n)$*, indicating execution failure of action* $\alpha^n$ *on* $s^n$*. We also write a trace* $\langle \bar{s}, \bar{o}, \bar{\alpha} \rangle$*, splitting it into sequences of states, observations, and actions, respectively, and omitting the final symbol.*

*A trace t is a* goal trace *for problem* $\langle \mathcal{D}, \mathcal{G} \rangle$ *iff t is not a failure trace, and its final state, denoted* $final(t)$*, belongs to* $\mathcal{G}$*. We indicate with* $Trs(\pi, \mathcal{D})$*, the set of traces associated to plan* $\pi$ *in domain* $\mathcal{D}$*.* $TrsFail(\pi, \mathcal{D})$ *and* $TrsG(\pi, \mathcal{D}, \mathcal{G})$ *are, respectively, the subsets of the failure and goal traces in* $Trs(\pi, \mathcal{D})$*.*

A plan is called a strong solution for a planning problem $\langle \mathcal{D}, \mathcal{G} \rangle$ iff every execution does not fail, and ends in $\mathcal{G}$:

**Definition 4 (Strong solution).** *A plan* $\pi$ *is a strong solution for a problem* $\langle \mathcal{D}, \mathcal{G} \rangle$ *iff* $Trs(\pi, \mathcal{D}) = TrsG(\pi, \mathcal{D}, \mathcal{G})$

## 3 Assumptions, Assumption-based solutions

To express assumptions over the behavior of $\mathcal{D}$, we adopt *Linear Temporal Logic* (LTL) [Emerson, 1990], whose underlying ordered structure of time naturally models the dynamic evolution of domain states. LTL expresses properties over sequences of states, by introducing the temporal operators **X** (next) and **U** (until):

**Definition 5 (LTL syntax).** *The language* $\mathcal{L}$ *of the LTL formulæ* $\varphi$ *on* $\mathcal{P}$ *is defined by the following grammar, where* $q \in \mathcal{P}$*:*

$$\varphi := q \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi$$

The derived operators **F** (future) and **G** (globally) are defined on the basis of **U**: $\mathbf{F}\varphi = \top \mathbf{U} \varphi$, and $\mathbf{G}\varphi = \neg\mathbf{F}\neg\varphi$.

The semantics of LTL are given inductively on state sequences, see [Manna and Pnueli, 1992].

**Definition 6 (LTL semantics).** *Given an infinite state sequence* $\sigma = [s^0, \ldots, s^n, \ldots]$*, an LTL formula* $\varphi$ *holds at a position* $i \geq 0$ *in* $\sigma$*, denoted by* $(\sigma, i) \models \varphi$*, iff*

$$\begin{array}{rl} \varphi \in \mathcal{P}rop & \textit{and } s^i \models \varphi \textit{, or} \\ \varphi = \neg\psi & \textit{and } (\sigma,i) \not\models \psi \textit{, or} \\ \varphi = \psi \wedge \gamma & \textit{and } (\sigma,i) \models \psi \wedge (\sigma,i) \models \gamma \textit{, or} \\ \varphi = \mathbf{X}\psi & \textit{and } (\sigma,i+1) \models \psi \textit{, or} \\ \varphi = \psi\mathbf{U}\gamma & \textit{and } \exists i, (\sigma,i) \models \gamma \wedge \forall j \textit{ s.t. } j \leq i,\ (\sigma,j) \models \psi. \end{array}$$

We say that an LTL formula $\varphi$ is satisfiable over a plan trace $\langle \bar{s}, \bar{o}, \bar{\alpha} \rangle \in Trs(\pi, \mathcal{D})$ iff it holds at position 0 for some prolongation of $\bar{s}$. Thus, given an LTL assumption $\mathcal{H}$, the traces $Trs(\pi, \mathcal{D})$ of a plan $\pi$ can be partitioned into those traces for which $\mathcal{H}$ is satisfiable, and those for which it is not, denoted $Trs_{\mathcal{H}}(\pi, \mathcal{D})$ and $Trs_{\bar{\mathcal{H}}}(\pi, \mathcal{D})$ respectively. The failure traces $TrsFail(\pi, \mathcal{D})$ are partitioned analogously into $TrsFail_{\mathcal{H}}(\pi, \mathcal{D})$ and $TrsFail_{\bar{\mathcal{H}}}(\pi, \mathcal{D})$, and so for $TrsG(\pi, \mathcal{D}, \mathcal{G})$, partitioned into $TrsG_{\mathcal{H}}(\pi, \mathcal{D}, \mathcal{G})$ and $TrsG_{\bar{\mathcal{H}}}(\pi, \mathcal{D}, \mathcal{G})$.

[1] We use the standard notation $\bar{x} = [x^1 \ldots x^n]$ for a sequence of n elements, whose $i$-th element is indicated with $x^i$. Concatenations of two sequences $\bar{x}_1$ and $\bar{x}_2$ is denoted $\bar{x}_1 \circ \bar{x}_2$. The length of $\bar{x}$ is denoted $|\bar{x}|$, and $\lfloor x \rfloor_L$ is its prefix of length L.

If every possible execution for which $\mathcal{H}$ is satisfiable succeeds, then the plan is a solution under assumption $\mathcal{H}$:

**Definition 7 (Solution under Assumption).** *A plan $\pi$ is a solution for the problem $\langle \mathcal{D}, \mathcal{G} \rangle$ under the assumption $\mathcal{H}$ iff* $Trs_{\mathcal{H}}(\pi, \mathcal{D}) = TrsG_{\mathcal{H}}(\pi, \mathcal{D}, \mathcal{G})$

**Example 1.** *We introduce a simple navigation domain for explanatory purposes, see Fig. 1. A mobile robot, initially placed in room I, must reach room $K_3$. The shaded cells $K_1, K_2, K_3$ are kitchens, while the other ones are normal rooms. The robot is only equipped with a smell sensor **K**, that allows to detect whether it is in a kitchen room. The robot moves at each step in one of the four compass directions $(n, e, s, o)$; moving onto a wall is not possible.*

*We do not know the speed of the robot: a movement might terminate in any of the rooms in the movement direction (for instance, moving north from I may end up in $K_1, R_2$ or in $R_3$).*

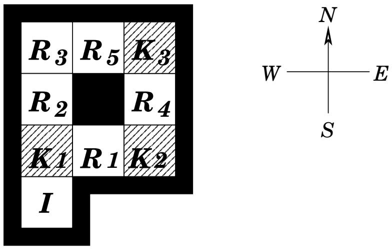


Figure 1: An example domain.

*A strong solution for this problem does not exist, as it is easy to see. However, solutions exists if we assume that the robot steps of one room at a time, at least until it reaches the goal. The considered assumption formula is $\mathcal{H}_0 = (\mathbf{X}(\delta = 1)\,\mathbf{U}\,K_3)$, where $\delta$ models the Manhattan distance between two successive cells. The simple plan $\pi_1 = n \circ n \circ n \circ e \circ e \circ \varepsilon$ is a solution under $\mathcal{H}_0$. This plan has the following possible traces $Trs(\pi_1, \mathcal{D})$:*

$t_1 : [I, \overline{K}, n, K_1, K, n, R_2, \overline{K}, n, R_3, \overline{K}, e, R_5, \overline{K}, e, K_3, K, Stop.]$

$t_2 : [I, \overline{K}, n, K_1, K, n, R_3, \overline{K}, Fail(n)]$

$t_3 : [I, \overline{K}, n, R_2, \overline{K}, n, R_3, \overline{K}, Fail(n)]$

$t_4 : [I, \overline{K}, n, R_3, \overline{K}, Fail(n)]$

$t_5 : [I, \overline{K}, n, K_1, K, n, R_2, \overline{K}, n, R_3, \overline{K}, e, K_3, K, Fail(e)]$

*Indeed $Trs_{\mathcal{H}_0}(\pi_1, \mathcal{D}) = TrsG_{\mathcal{H}_0}(\pi_1, \mathcal{D}, \mathcal{G}) = \{t_1\}$.*

## 4 Safe assumption-based plans

Assumptions may not hold at run-time; for this reason, assumption-based plans are usually executed within a reactive framework (see e.g. [Bertoli *et al.*, 2001; Muscettola *et al.*, 1998]), where an external beholder of the execution (called *monitor*) observes the responses of the domain to the actions coming from the plan, and triggers replanning when such observations are not compatible with some expected behavior.

However, the monitor may not always be able to decide whether the domain is reacting as expected or not to the stimuli of a plan $\pi$. This happens when, given a sequence of actions $\bar{\alpha}$, the domain produces a sequence of observations $\bar{o}$ compatible both with some assumed domain behavior, and with some behavior falsifying the assumption, i.e. $\exists \bar{s}, t : \langle \bar{s}, \bar{o}, \bar{\alpha} \rangle \circ t \in Trs_{\mathcal{H}}(\pi, \mathcal{D})$ and $\exists \bar{s}', t' : \langle \bar{s}', \bar{o}, \bar{\alpha} \rangle \circ t' \in Trs_{\overline{\mathcal{H}}}(\pi, \mathcal{D})$. In this case, if $final(\langle \bar{s}', \bar{o}, \bar{\alpha} \rangle)$ is a state for which the next plan action is not applicable, the monitor will trigger replanning, in order to rule out any chance of run-time failures. This is undesirable: the assumption might actually hold, in which case plan execution should proceed.

**Example 2.** *Consider the plan $\pi_1$ from example 1. Trace $t_2$ produces the same observations (namely $[\bar{K}, K, \bar{K}]$) of the successful trace $t_1$, in response to the same actions $(n \circ n)$, up to its failure. Thus, after the first n move, the monitor knows the robot is in $K_1$; after the second, the monitor cannot distinguish if the robot is in $R_2$ or $R_3$. In case it is in $R_3$, the next north action is inapplicable, so the plan execution has to be stopped. This makes $\pi_1$ practically useless, since its execution is always halted after two actions, even if the assumption under which it guarantees success actually holds.*

Similarly, an assumption-based solution plan may terminate without the monitor being able to distinguish whether the goal has been reached or not, therefore triggering replanning at the end of plan execution.

Whether such situations occur depend on the nature of the problem and, crucially, of the plan chosen as an assumption-based solution. We are interested in characterizing and generating assumption-based solutions such that these situations do not occur, i.e. every execution that causes action inapplicability, or ends up outside the goal, must be distinguishable from every successful execution.

Thus we define safe LTL assumption-based plans as those plans that (a) achieve the goal whenever the assumption holds, and (b) guarantee that each execution where an assumption failure compromises the success of the plan is observationally distinguishable (from the monitor's point of view) from any successful execution.

**Definition 8 (Distinguishable traces).** *Let t and $t'$ be two traces; let $L = min(|t|, |t'|)$, $\lfloor t \rfloor_L = \langle \bar{s}, \bar{o}, \bar{\alpha} \rangle$ and $\lfloor t' \rfloor_L = \langle \bar{s}', \bar{o}', \bar{\alpha}' \rangle$. Then $t, t'$ are distinguishable, denoted $Dist(t, t')$, iff $(\bar{o} \neq \bar{o}') \vee (\bar{\alpha} \neq \bar{\alpha}')$.*

**Definition 9 (Safe LTL assumptions-based solution).** *A plan $\pi$ is a safe assumption-based solution for LTL assumption $\mathcal{H}$ iff the conditions below are met:*

*a)* $Trs_{\mathcal{H}}(\pi, \mathcal{D}) = TrsG_{\mathcal{H}}(\pi, \mathcal{D}, \mathcal{G})$

*b)* $\forall t \in Trs_{\overline{\mathcal{H}}}(\pi, \mathcal{D}) \backslash TrsG_{\overline{\mathcal{H}}}(\pi, \mathcal{D}, \mathcal{G})$,

$\forall t' \in Trs_{\mathcal{H}}(\pi, \mathcal{D}) \cup TrsG_{\overline{\mathcal{H}}}(\pi, \mathcal{D}, \mathcal{G}) : Dist(t, t')$

**Example 3.** *Consider the plan $\pi_2 = n \circ e \circ e \circ n \circ n \circ \varepsilon$. This is a safe assumption-based solution for the problem of example 1, as it is easy to see. The traces $Trs(\pi_2, \mathcal{D})$ are:*

$t'_1 : [I, \overline{K}, n, K_1, K, e, R_1, \overline{K}, e, K_2, K, n, R_4, \overline{K}, n, K_3, K, Stop]$

$t'_2 : [I, \overline{K}, n, K_1, K, e, R_1, \overline{K}, e, K_2, K, n, K_3, K, Fail(n)]$

$t'_3 : [I, \overline{K}, n, K_1, K, e, K_2, K, Fail(e)]$

$t'_4 : [I, \overline{K}, n, R_2, \overline{K}, Fail(e)]$

$t'_5 : [I, \overline{K}, n, R_3, \overline{K}, e, R_5, \overline{K}, e, K_3, K, Fail(n)]$

$t'_6 : [I, \overline{K}, n, R_3, \overline{K}, e, K_3, Fail(e)]$

*We have $TrsG_{\mathcal{H}_0}(\pi_2, \mathcal{D}, \mathcal{G}) = Trs_{\mathcal{H}_0}(\pi_2, \mathcal{D}) = \{t'_1\}$, and every trace $t'_2, t'_3, t'_4, t'_5, t'_6$ is distinguishable from $t'_1$.*

# 5 Generating safe LTL assumption-based plans

We intend to efficiently generate safe LTL assumption-based plans for partially observable, nondeterministic domains. For this purpose, we take as a starting point the plan generation approach presented in [Bertoli *et al.*, 2001], where an and-or graph representing an acyclic prefix of the search space of beliefs is iteratively expanded: at each step, observations and actions are applied to a fringe node of the prefix, removing loops. Each node in the graph is associated with a belief in the search space, and to the path of actions and observations that is traversed to reach it. When a node is marked success, by goal entailment or by propagation on the graph, its associated path is eligible as a branch of a solution plan. The implementation of this approach by symbolic techniques has proved very effective in dealing with complex problems, where uncertainty results in manipulating large beliefs.

To generate plans under an LTL assumption $\mathcal{H}$, using this approach, we have to adapt this schema so that the beliefs generated during the search only contain states that can be reached if $\mathcal{H}$ is satisfiable. Moreover, in order to constrain the algorithm to produce safe plans, success marking of a leaf node $n$ must require the safety conditions of def.9 (recast to those traces in $Trs_{\mathcal{H}}$ and $Trs_{\bar{\mathcal{H}}}$ that can be traversed to reach $n$). We now describe in detail these adaptations, and the way they are efficiently realized by means of symbolic representation techniques.

## 5.1 Assumption-induced pruning

In order to prune states that may only be reached if the assumption is falsified, we annotate each state $s$ in a (belief associated to a) search node with an LTL formula $\varphi$, representing the current assumption on how $s$ will evolve, and we evolve the pair $\langle s,\varphi\rangle$ in a way conceptually similar to [Kabanza *et al.*, 1997]. Thus, a graph node will now be associated to a set $\{\langle s,\varphi\rangle : s \in \mathcal{S}, \varphi \in \mathcal{L}\}$ called *annotated belief.* Formulæ will be expressed by unrolling top-level **U**s according to the tableau expansion rules [Somenzi and Bloem, 2000], so that they can be evaluated on the current state:

$$
\begin{aligned}
&Unroll(\psi) = \psi \quad \text{if } \psi \in \{\top,\bot\} \cup \mathcal{P}rop\\
&Unroll(\neg\varphi) = \neg Unroll(\varphi)\\
&Unroll(\varphi \star \psi) = Unroll(\varphi) \star Unroll(\psi) \quad \text{if } \star \in \{\vee,\wedge\}\\
&Unroll(\mathbf{X}\varphi) = \mathbf{X}\varphi\\
&Unroll(\varphi\,\mathbf{U}\,\psi) = Unroll(\psi) \vee (Unroll(\varphi) \wedge \mathbf{X}(\varphi\,\mathbf{U}\,\psi))
\end{aligned}
$$

Thus, the initial graph node will be

$$\mathcal{B}_I = \left\{\langle s, Unroll(\mathcal{H})|_s\rangle : s \in \mathcal{I},\ Unroll(\mathcal{H})|_s \neq \bot\right\}$$

where $\varphi|_s$ denotes the formula resulting from $\varphi$ by replacing its subformulaæ outside the scope of temporal operators with their evaluation over state $s$. Notice that the formulæ associated to states have the form $\bigwedge \mathbf{X}\varphi_i \wedge \bigwedge \neg\mathbf{X}\psi_j$, with $\varphi_i, \psi_j \in \mathcal{L}$. When a fringe node in the graph is expanded, its associated annotated belief $\mathcal{B}$ is progressed as follows:

- if an observation $o$ is applied, $\mathcal{B}$ is restricted to

$$\mathcal{E}_O(\mathcal{B},o) = \{\langle s,\varphi\rangle \in \mathcal{B} : s \in [\![o]\!]\}$$

- if an (applicable) action $\alpha$ is applied, each pair $\langle s,\varphi\rangle \in \mathcal{B}$ is progressed by rewriting $\varphi$ to refer to the new time instant, expanding untils, and evaluating it on every state in $\alpha(s)$:

$$\mathcal{E}_A(\mathcal{B},\alpha) = \{\langle s', X^{-1}(\varphi)|_{s'}\rangle : \langle s,\varphi\rangle \in \mathcal{B},\ s' \in \alpha(s), X^{-1}(\varphi)|_{s'} \neq \bot\}$$

where $X^{-1}(\varphi)$ is defined as follows:

$$
\begin{aligned}
&X^{-1}(\psi) = \psi \quad \text{if } \psi \in \{\top,\bot\}\\
&X^{-1}(\neg\psi) = \neg X^{-1}(\psi)\\
&X^{-1}(\varphi \star \psi) = X^{-1}(\varphi) \star X^{-1}(\psi) \quad \text{if } \star \in \{\vee,\wedge\}\\
&X^{-1}(\mathbf{X}\varphi) = Unroll(\varphi)
\end{aligned}
$$

An annotated belief associated to a graph node contains the final states of the traces $Trs_{\mathcal{H}}(p,\mathcal{D})$ which traverse the path $p$ associated to the node.

## 5.2 Enforcing safety

Def. 9.a is easily expressed as an entailment between the states of the current annotated belief and the goal. To efficiently compute the distinguishability (requirement 9.b), we associate to a search node the sets of undistinguishable final states of $Trs_{\mathcal{H}}(p,\mathcal{D})$ and $Trs_{\bar{\mathcal{H}}}(p,\mathcal{D})$, storing them within a couple of annotated beliefs $\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle$. When $\mathcal{B}_{\mathcal{H}}$ and $\mathcal{B}_{\bar{\mathcal{H}}}$ only contain goal states, this indicates that the only undistinguishable behaviors lead to success, so the path satisfies requirement 9.b (in particular, if $\mathcal{B}_{\mathcal{H}}$ and $\mathcal{B}_{\bar{\mathcal{H}}}$ are empty, this indicates that the monitor will be able to distinguish any assumption failure along $p$).

During the search, $\mathcal{B}_{\mathcal{H}}$ and $\mathcal{B}_{\bar{\mathcal{H}}}$ are progressed similarly as the annotated belief representing the search node; but on top of this, they are pruned from the states where the success or failure of the assumption can be distinguished by observations. While progressing $\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle$, we detect situations where undistinguishable assumption failures may compromise action executability. These situations inhibit the safety of the plan, and as such we cut the search on these branches of the graph.

Initially, we compute $\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle$ by considering those states of $\mathcal{I}$ for which $\mathcal{H}$ (resp. $\bar{\mathcal{H}}$) is satisfiable, and by eliminating from the resulting couple the states distinguishable with the initial observation, i.e.
$\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle = \left\langle prune\left(\mathcal{B}^0_{\mathcal{H}}, \mathcal{B}^0_{\bar{\mathcal{H}}}\right), prune\left(\mathcal{B}^0_{\bar{\mathcal{H}}}, \mathcal{B}^0_{\mathcal{H}}\right)\right\rangle$, where

$$
\begin{aligned}
\mathcal{B}^0_{\mathcal{H}} &= \left\{\langle s, Unroll(\mathcal{H})|_s\rangle : s \in \mathcal{I},\ Unroll(\mathcal{H})|_s \neq \bot\right\}\\
\mathcal{B}^0_{\bar{\mathcal{H}}} &= \left\{\langle s, Unroll(\bar{\mathcal{H}})|_s\rangle : s \in \mathcal{I},\ Unroll(\bar{\mathcal{H}})|_s \neq \bot\right\}
\end{aligned}
$$

$$prune(\mathcal{B},\mathcal{B}') = \left\{\langle s,\varphi\rangle \in \mathcal{B} :\ \exists\langle s',\varphi'\rangle \in \mathcal{B}' : \mathcal{X}(s) \cap \mathcal{X}(s') \neq \emptyset\right\}$$

When a node is expanded, its associated pair is expanded as follows:

- if an observation $o$ is applied, $\mathcal{B}_{\mathcal{H}}$ and $\mathcal{B}_{\bar{\mathcal{H}}}$ are simply pruned from the states not compatible with $o$:

$$\mathcal{E}_O(\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle, o) = \langle \mathcal{E}_O(\mathcal{B}_{\mathcal{H}}, o), \mathcal{E}_O(\mathcal{B}_{\bar{\mathcal{H}}}, o)\rangle$$

- if the node is expanded by an action $\alpha$, and $\alpha$ is not applicable on some state of $\mathcal{B}_{\bar{\mathcal{H}}}$, then a "dangerous" action is attempted on a state that can be reached by an undistinguishable assumption failure. This makes the plan unsafe: as such, we mark the search node resulting from this expansion as failure.

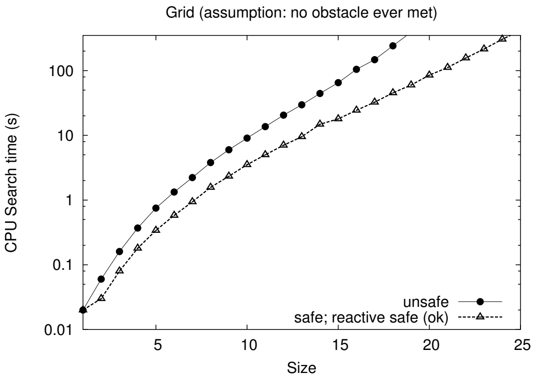


Figure 2: Tests for grid.

- If the node is expanded by an action $\alpha$, and $\alpha$ is applicable on every state of $\mathcal{B}_{\mathcal{H}}$ and $\mathcal{B}_{\bar{\mathcal{H}}}$, then $\mathcal{E}_A(\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, \alpha) = \langle prune(\mathcal{B}'_{\mathcal{H}}, \mathcal{B}'_{\bar{\mathcal{H}}}), prune(\mathcal{B}'_{\bar{\mathcal{H}}}, \mathcal{B}'_{\mathcal{H}}) \rangle$ where $\mathcal{B}'_{\mathcal{H}} = \mathcal{E}_A(\mathcal{B}_{\mathcal{H}}, \alpha)$ and $\mathcal{B}'_{\bar{\mathcal{H}}} = \mathcal{E}_A(\mathcal{B}_{\bar{\mathcal{H}}}, \alpha)$.

### 5.3 Symbolic annotated beliefs

Progressing annotated belief states by explicitly enumerating each state becomes soon unfeasible when beliefs contain large sets of states. To scale up on significant problems, we exploit symbolic techniques based on BDDs ([Bryant, 1986]) that allow efficiently manipulating sets of states; such techniques are indeed the key behind the effectiveness of approaches such as [Bertoli *et al.*, 2001]. To leverage on BDD primitives, we group together states associated with the same formula, and represent annotated beliefs as sets of pairs $\langle B, \varphi \rangle$, where $B$ is a set of states (a belief). Inside a *symbolic annotated belief*, we do not impose that two beliefs are disjoint. A state belonging to two pairs $\langle B_1, \varphi_1 \rangle$ and $\langle B_2, \varphi_2 \rangle$ is in fact associated to $\varphi_1 \vee \varphi_2$. Then, to progress a symbolic annotated belief, we observe that, given a belief $B$ and an LTL formula $\varphi$, $\varphi$ induces a coverage of $B$, where each element (a subset of $B$) collects the states for which a disjunct $t$ in the disjunctive normal form of $\varphi$ is satisfiable. If we denote with $Prop(t)$ the propositional part of a disjunt $t$ (i.e. the conjunction of its literals that appear outside any temporal modality), and $Time(t)$ the complementary temporal part, the coverage is $\{B \cap Prop(t) : t \in Dnf(Unroll(t))\}$, where $Prop(t)$ and $Time(t)$ identify the propositional and temporal portion of a disjunct respectively. Applying this idea, the initialization and progression of search nodes given in Sec. 5.1 are represented as follows:

$$\begin{aligned}
\mathcal{B}_I = \{ & \langle \mathcal{I} \cap Prop(t), Time(t) \rangle : \\
& t \in Dnf(Unroll(\mathcal{H})), \mathcal{I} \cap Prop(t) \neq \emptyset \} \\
\mathcal{E}_O(\mathcal{B}, o) = \{ & \langle B \cap [\![ o ]\!], \varphi \rangle : \langle B, \varphi \rangle \in \mathcal{B}, B \cap [\![ o ]\!] \neq \emptyset \} \\
\mathcal{E}_A(\mathcal{B}, \alpha) = \{ & \langle \alpha(B) \cap Prop(t), Time(t) \rangle : \\
& \langle B, \varphi \rangle \in \mathcal{B}, t \in Dnf(X^{-1}(\varphi)), \alpha(B) \cap Prop(t) \neq \bot \}
\end{aligned}$$

The initialization and progression of the undistinguishability sets are expressed similarly; we omit them for lack of space.

## 6 Experimental evaluation

Our experiments intend to evaluate the impact of exploiting LTL assumptions, and of enforcing safety, when generating plans and executing them in a reactive framework. For these purposes, the MBP planner inside SYPEM reactive platform [Bertoli *et al.*, 2001] has been modified so that it generates safe LTL assumption-based plans. We name SLAM (Safe LTL Assumption-based MBP) our extension of MBP; SLAM can also be run with the safety check disabled, thus performing unsafe LTL assumption-based planning.

We also adapted the monitoring component of SYPEM to detect the failure of LTL assumptions; we will name SALPEM the reactive platform including SLAM and the adapted monitor. Basically, the monitor of SALPEM progresses the symbolic annotated belief associated to the assumption via the $\mathcal{E}_A$ function, and prunes away states inconsistent with the actual observations, signaling assumption failure when such annotated belief is empty.

We tested SALPEM on a set of problems, using a Linux equipped 2GHz Pentium 4 with 224MB of RAM memory.

We first consider a navigation problem where a robot must traverse a square grid, from the south-west corner to the north-east one. There can be obstacles (persons) in the grid, and of course the robot cannot step over them; the number and positions of these obstacles is unknown. The robot can move of one cell at a time in the four compass directions, and it can interrogate its proximity sensors to detect obstacles around him. A strong plan does not exist; however, if we assume that no obstacle is ever met by the robot, a plan can be found rather easily. Fig. 2 reports the timings for safe and unsafe planning. Enforcing safety in this case causes a pruning of the search that reflects on an improved search time. Moreover, when the assumption holds, safe plans reach the goal without replanning, while unsafe plans cause infinite replanning episodes. This is because, while safe plans exploit sensing at every step to allow monitoring the assumption, unsafe plans do not; but then, the monitor cannot establish whether moving is actually possible and, to avoid a possible run-time failure, replans, possibly even before any progress is made by the plan. We also experimented with a more relaxed assumption, namely that the persons are "fair" and always leave at least one free direction for the robot to get closer to its goal. While unsafe and safe plans become more complex, we obtained qualitatively similar results: enforcing safety imposes a very minor overhead, but prevents useless and infinite replanning episodes. We omit the results for reasons of space.

To evaluate the convenience of using assumptions, even when strong plans exist, let us consider the following example. A production chain must transform an initially raw item by applying a sequence of manufacture phases to get the item in a final, refined form. Each phase, modeled as an action, prepares the item for the next one, but may also fail once, in which case the item is ruined and we have to "undo" the phase by performing an ad-hoc operation. An item inspection operation can be commanded, which allows detecting whether the item is ruined. Fig. 3 shows that strong planning becomes very hard but for small domain instances, due to the high number of branches to be considered; if we consider the assumption that no failure ever occurs, safe and un-

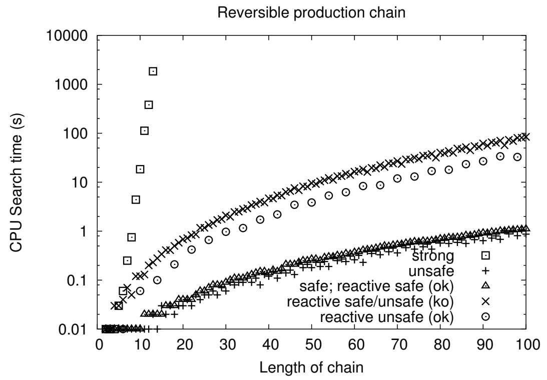


Figure 3: Tests for production chain.

safe assumption-based plan generation scale up much better, exhibiting similar performances. Once more, when the assumption holds, the execution of safe solutions achieves the goal without replanning, while unsafe solutions may (and in our experiments do) cause infinite sequences of replannings, due to the attempt of performing manufacturing phases prior to checking the status of the item. To prevent infinite replannings when safety is not enforced, we also experimented with an ad-hoc heuristic that forces initial inspections; but also in this case, useless replanning episodes do take place, degrading the overall plan-execution loop performance: basically, then, the performance of the plan-execution is independent on whether the assumption actually holds.

The results of our experiments clearly show the advantages of exploiting LTL assumptions, and of enforcing safety for LTL assumption-based plans. In particular, we observe that, at plan generation time, the safety-induced search pruning appears to balance the cost of computing safety conditions; thus the overhead of imposing safety is very limited, if at all present, and is more than compensated by the highly improved run-time behavior of the obtained solutions.

## 7 Conclusions

In this paper, we tackled the problem of efficiently building assumption-based solutions for partially observable and partially known domains, using an expressive logics, LTL, to describe assumptions over the domain dynamics. Moreover, we constrained the search to generate safe solutions, which allow a monitor to unambiguously identify, at run-time, domain behaviors unplanned for. Both of our contributions are, to the best of our knowledge, novel, and substantially extend existing works on related research lines.

In particular, LTL has been used as a means to describe search strategies (and goals) in ([Bacchus and Kabanza, 2000]). While also that work exploits LTL to restrict the search, it focuses on the much simpler framework of fully deterministic and observable planning. This greatly simplifies the problem: no ambiguous monitoring result is possible (thus safety is guaranteed), and since the nodes of the search space are single states, it is possible to simply evaluate the progression of formulæ over states, without recurring to symbolic representation techniques to progress beliefs.

A first, less general notion of safety is first presented in [Albore and Bertoli, 2004], considering the limited setting of propositional assumptions over the initial state. The ability of representing assumptions over the dynamics of the domain is crucial to the applicability of assumption-based planning; none of the examples presented here could have been handled within such a setting. At the same time, using temporal logic assumptions implies a much more complex treatment of how beliefs are progressed during the search.

Several future directions of work are open. First, in our work, assumptions are an explicit input to the planning process; e.g. they could be provided by a user with a knowledge of the domain. While it is often practically feasible to formulate assumptions (e.g. on nominal behaviors), it would be extremely useful to be able to (semi-)automatically extract and formulate assumptions from the domain description, possibly learning by previous domain behaviors. Second, in certain cases a more qualitative notion of safety can be useful to practically tune the search to accept 'almost safe' plans, or score the quality of plans depending on a degree of safety.

## References

[Albore and Bertoli, 2004] A. Albore and P. Bertoli. Generating Safe Assumption-Based Plans for Partially Observable, Nondeterministic Domains. In *Proc. of AAAI04*, 2004.

[Bacchus and Kabanza, 2000] F. Bacchus and F. Kabanza. Using Temporal Logic to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[Bertoli *et al.*, 2001] P. Bertoli, A. Cimatti, and M. Roveri. Conditional Planning under Partial Observability as Heuristic-Symbolic Search in Belief Space. In *Proc. of ECP'01*, 2001.

[Bryant, 1986] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Emerson, 1990] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*. 1990.

[Kabanza *et al.*, 1997] F. Kabanza, M. Barbeau, and R. St-Denis. Planning Control Rules for Reactive Agents. *Artificial Intelligence*, 95(1):67–113, 1997.

[Manna and Pnueli, 1992] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Printer-Verlag, 1992.

[Muscettola *et al.*, 1998] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[Myers and Wilkins, 1998] Karen L. Myers and David E. Wilkins. Reasoning about locations in theory and practice. *Computational Intelligence*, 14(2):151–187, 1998.

[Somenzi and Bloem, 2000] Fabio Somenzi and Roderick Bloem. Efficient B uchi Automata from LTL Formulæ. *Computed-Aided Verification*, 1855(LNCS), 2000.

# Path planning for Unmanned Underwater Vehicles

**Clément Pêtrès and Pedro Patrón**
Heriot Watt University
School of Engineering and Physical Sciences
Edinburgh, EH14 4AS
{cp23, P.Patron}@hw.ac.uk

## Abstract

Efficient path planning algorithms for embedded systems are a crucial issue for modern unmanned underwater vehicles. This paper proposes a method which is able to find paths from continuous environments prone to fields of force in a reliable and efficient manner. Classical path planning algorithms in artificial intelligence have limited performance and they are not designed to cope with real-time constraints of systems moving in a hostile underwater environment. We present a novel approach based on an advanced numerical technique called the Fast Marching algorithm to solve the following three issues. First, we extract a continuous path in an environment evenly mapped to a discrete grid. Secondly, the vehicle kinematics is introduced as a constraint on the optimal path curvature, and thirdly we take underwater currents into account thanks to an efficient extension of the original Fast Marching algorithm. Finally, a multiresolution scheme based on adaptive mesh generation is compared to incremental search techniques to speed up the overall process. A flexible platform is eventually presented to simulate path searching behaviors in real-time.

## 1 Introduction

National security is a priority for governments all over the world. The increasing political importance of human losses and the current state of technologies allow us to move forward in the development of unmanned vehicles prototypes for battlefield access. Huge investments in time and money are currently being dedicated to this field of research.

One of the environments that is most promising is the underwater world. High stealth levels combined with high mobility allow access to places that were inaccessible before without risking human lives. In this environment, impact or entrapment could produce the loss of an expensive vehicle and, maybe more importantly, the failure of the mission. It is very important to have a reliable deliberative obstacle avoidance system.

### 1.1 Underwater environment

In mobile robotics path planning, research has focussed on wheeled robots moving on 2D surfaces fitted out with high rate communication modules. The underwater environment is much more demanding: it is difficult to communicate; it is prone to currents; and the set of possible paths is a 3D space. Moreover, underwater torpedo like vehicles are strongly nonholonomic, whereas wheeled robots are easily capable of stopping and rotating.

Although theoretical studies have been carried out, little work has been done in the underwater field on systems that actually implement real obstacle avoidance systems. Most of the approaches suffer from a lack of efficiency when they are moved to the real-time frame.

### 1.2 Current approaches

Several studies have been realized in the Ocean System Laboratory (OSL) over the last few years for real time applications [Petillot *et al.*, 2001; D.M.Lane *et al.*, 2001; Y. Wang, 1999]. They were developed as a safety module for projects not directly related with obstacle avoidance. They used potential field algorithms with constructive solid geometry (CSG) data structures coming from the sensor fusion that allows them to run embedded in real vehicles.

Modern deliberative solutions of high-level obstacle avoidance and path planning systems use some form of local mapping system. There are few references to works in more than two spatial dimensions. One of the reasons for this could be the lack of resources to collect three-dimensional information from the local environment of the vehicle [CodaOctopus Ltd., 2004][Zimmerman, 2004].

### 1.3 Main contributions

Previous research suggests that methods such as potential field or roadmap methods are useful for path planning. We will briefly review these methods. However, they have proved to be inefficient when embedded in real vehicles moving in complex and dynamic environments.

We propose to rate another direction in path planning which uses cell decomposition. Cell decomposition approaches are widely used in mobile robotics because the are suitable for sensor images mapped to a grid of pixels. The key issue is then to use an efficient grid-search algorithm to find an acceptable path.

Breadth-first, depth-first and hybrid search algorithms are extensively used in artificial intelligence (AI) because of their low complexity. But these discrete graph-search algorithms are not consistent in the continuous domain, they assimilate the vehicle as a static point and they do not deal with directional constraints.

The main contribution of the authors is to present the Sethian's Fast Marching algorithm (FM) as an advanced tool for path planning. With the same low complexity of the above classical grid-search algorithms FM converges to the right continuous path when it is implemented on a discrete grid. This specificity is crucial for the following two properties of our method.

First, the precision of Fast Marching allows the curvature of the final path to be constrained. This property enables us to take the vehicle kinematics into account.

Secondly, we show that Fast Marching based path planners are able to deal with smooth fields of force. We show an application with underwater currents but the concept can be generalized for any kind of directional constraints.

Finally we propose a multiresolution scheme to speed up the overall method. This is achieved by coupling an octree decomposition with an adaptive mesh generation.

The methods proposed are validated using a complete simulator developed over the last few years in OSL. It uses same interfaces as the real prototype and has been demonstrated to be a reliable tool for 'hardware in the loop' simulations.

## 2 Path planning overview

Many methods were proposed in the literature to address the path planning problem. We give here an overview of the most popular ones.

### 2.1 Potential field methods

Potential field methods use the physics of electrical potentials as an heuristic to guide the search for a path [Latombe, 1991]. Movements of the robot (represented as a particle) are governed by a potential field which is usually comprised of two components, an attractive potential drawing the robot towards the goal and a repulsive potential pushing the robot away from obstacles. The main drawback with these methods is their susceptibility to local minima.

### 2.2 Roadmap methods

The idea behind roadmap approaches is to reduce the map to a network of one dimensional curves. If start and goal points are linked to this network then path planning becomes a graph searching problem. The key issue is the method used to construct the roadmap.

**Visibility graphs**

A visibility graph is constructed by considering all the vertices of the obstacles and the start and goal points as the graph nodes [Liu and Arimoto, 1992]. The graph links are the line segments which connect two nodes without intersecting any obstacle. This method becomes complex in more than two dimensions.

**Voronoi diagrams**

A Voronoi diagram is the set of points that are equidistant from two or more obstacles (see for example [Takahashi and Schilling, 1989]. The result is a roadmap where the edges (and therefore the generated path) stay away from the obstacles. This method was extended for path planning in arbitrary dimensions (the silhouette methods [Canny, 1988]). It has not been widely used as a practical solution but has been a tool for analyzing the complexity of motion planning.

**Probabilistic Path Planning**

Probabilistic Path Planning (PPP) is a general planning scheme which includes probabilistic roadmap methods (PRM) (see [Svestka and Overmars, 1998] for a survey) and rapidly exploring random trees (RRT) developped by Lavalle [LaValle, 2005]. Both methods share the same idea. A reachability graph is incrementally built between the start and the goal positions based on randomized intermediate positions. A local operator is used to locally link positions in a feasible way for the the robot (avoiding obstacles and respecting kinematic constraints for example). PPP is probabilistically complete and allows one to deal with high dimensional configuration spaces. Nonetheless this method may fail to find a solution when the environment presents singularities (the 'narrow passage' problem) and it is heavy to implement for real-time applications in dynamic environments.

### 2.3 Classical grid-search algorithms

In this section we assume that the environment is sampled on a uniform grid. The key issue is then to use a suitable search algorithm to find an optimal path for a particular criterion, usually defined by a metric.

**Metric definition**

The metric $\rho$ which we will refer to from now on, is defined as:

$$\rho(x, x') = \int_C \tau(C(s))ds \tag{1}$$

where $C$ is a path between two points $x$ and $x'$, and $\tau$ is a strictly positive cost function.

This metric defines the distance to be the cost-to-go for a specific robot moving in an environment described in the cost function $\tau$.

**Grid-search principle**

The next three types of graph-search algorithms (called grid-search algorithm when they are implemented on a grid) are very popular in AI for planning paths. Their principle is always to build a 'minimum cost-to-go' map $U$ defined as follows:

$$U(x) = \inf_{A_{x_0,x}} \rho(x_0, x) \tag{2}$$

where $A_{x_0,x}$ is the set of all paths between the source $x_0$ and the current point $x$.

The map $U$ can be seen as a distance map weighted by the costs of the features of the image.

All grid-search algorithms (including the Fast Marching algorithm described in the next section) share the necessary backtracking step. Once the distance map is built until the goal point the optimal path is the one which follows the steepest descent from the goal to the start point. It is equivalent to say that we solve a functional minimization problem. The overall method is robust as no local minima are exhibited during the exploration process.

**Breadth-first search**

Breadth-first (BF) algorithm explores the grid points in order of their distance from the start point [Korf, 1998]. At each step, the next point $p$ to be computed is one whose distance $U(p)$ is the lowest. This algorithm is also known as Dijkstra's single source shortest-path algorithm, and one can show that its complexity is in $O(n \log(n))$, where $n$ is the number of grid points (or pixels in an image).

The set of points between the points already computed and the points not yet explored is an interface which can be seen as a front propagating outward the start point (like a flame in a landscape).

**Depth-first search**

Depth-first (DF) algorithm is very similar to BF search. The only difference is in the choice of the next point to compute. In a depth-first algorithm the priority of the point $p$ is given heuristically by a distance function $h(p)$ which is an estimate of the residual distance between the point $p$ and the goal point. Instead of developing a front around the start point, this method tends to focus the search directly to the goal point.

**Hybrid search**

The A* algorithm is probably the most popular hybrid search algorithm in artificial intelligence. It combines features of BF and DF to efficiently compute acceptable solutions. A* is an algorithm in which the priority for a point $p$ to be computed is given by a mixed distance function $f(p) = U(p) + h(p)$, where $U(p)$ and $h(p)$ are the distance functions defined above.

All these grid-search algorithms are *resolution complete* but they suffer from 'metrication errors'. Results from these discrete search algorithms can be improved by taking a larger neighborhood as a structuring element, giving better approximations of $\rho$ in some directions like $\sqrt{2} \cdot \tau$ for the diagonals. However, there will always be an error in some direction that will be invariant to the grid resolution. It is not the case with the Sethian's Fast Marching algorithm [Sethian, 1999].

## 3 Fast Marching based path planning

The FM algorithm is also a Dijkstra's like graph-search algorithm but is consistent in the continuous domain. The idea behind the FM algorithm is to improve the update of computed points by approximating the first derivative of the distance function $U$. For the same $O(n \log(n))$ complexity, this algorithm provides a better approximation as we can see on the figure 1.

### 3.1 Fast Marching algorithm (FM)

Sethian proposed using the Godunov Hamiltonian [Rouy and Tourin, 1992] which is a one-sided derivative. It looks to the up-wind direction of the moving front, and thereby avoids the classical over-shooting of finite differences schemes. At each pixel $(i, j)$, the unknown $u$ satisfies:

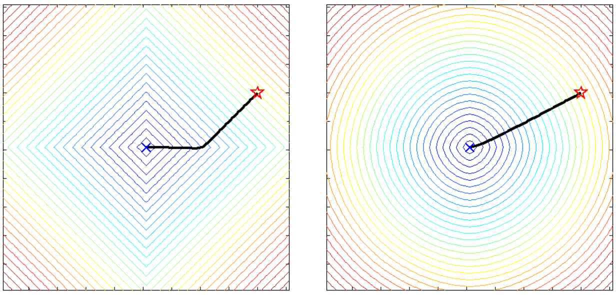


Figure 1: In these examples the cost function $\tau$ is constant on the entire image, so the $\rho$ metric can be seen as the Euclidean distance. On the left, a 4-connexity breadth-first algorithm gives square level sets. On the right, the distance is computed with a 4-connexity FM, giving circles. Therefore optimal paths are different for the same criterion.

$$
\begin{aligned}
&(\max\{u - U_{i-1,j}, u - U_{i+1,j}, 0\})^2 + \\
&(\max\{u - U_{i,j-1}, u - U_{i,j+1}, 0\})^2 = \tau_{i,j}^2 \qquad (3)
\end{aligned}
$$

yielding the correct viscosity solution $u$ for $U_{i,j}$. See figure 2 for an illustration of the expansion of FM.

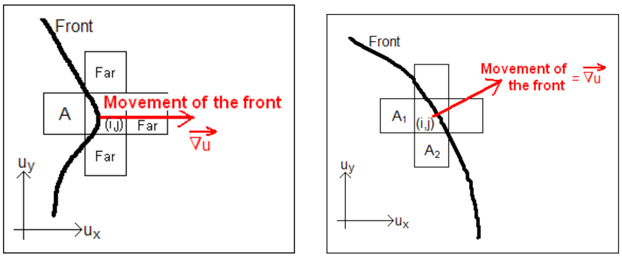


Figure 2: On the left, an example of a new expanded point only surrounded (in 4-connexity) by one other computed point. On the right, a new expanded point surrounded by at least two other computed points.

**Curvature constraints**

It has been shown [Cohen and Kimmel, 1997] that the curvature radius $r$ along the geodesic given by performing a FM which minimizes the functional $\int_D \tau(C(s))ds$ on the image domain $D$ is explicitly bounded by:

$$
r \geq \frac{\inf_D \{\tau\}}{\sup_D \{\|\nabla \tau\|\}} \qquad (4)
$$

This result gives us a nice interpretation of the connection between the cost function $\tau$ and the curvature along the resulting path. It is useful because we can know *a priori* if the path will be reachable or not by the vehicle. We just have to compare the turning radius $R$ of the vehicle with the lower bound $r_{lim}$ of the optimal path curvature radius:

- if $R < r_{lim}$, it is certain that the path will be reachable.
- if $R \geq r_{lim}$ there is a risk of collision. In that case we just have to smooth $\tau$ to increase the curvature limit until $r_{lim} > R$ (see figure 3).

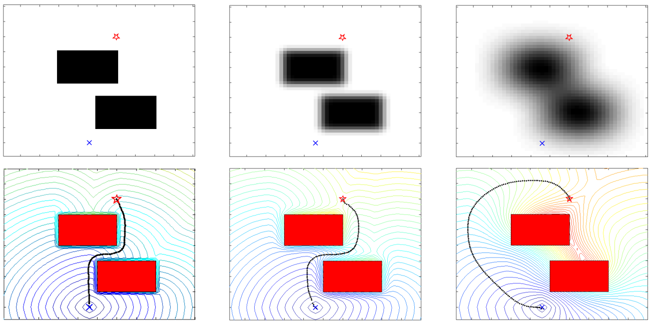


Figure 3: On the top, from left to right, cost functions corresponding respectively to $r_{lim} = 14, 34, 140$ (in arbitrary unit). On the bottom, the related optimal paths.

**Directional constraints**

Classical grid-search algorithms and the original FM algorithm are based on an implicit isotropic assumption of the free space. However, as FM explicitly estimates the gradient of the distance map, which can be interpreted as the moving direction, it is possible to take directional constraints such as winds or currents into account.

The theory of anisotropic Fast Marching was first developed by Vladimirsky [Vladimirsky, 2003]. The principle is to make the cost function $\tau$ dependent not only from scalar representation of obstacles but also dependent on vectorial forces. Vladimirsky formally demonstrates how the characteristic of the distance map can be used for this purpose.

In this section we propose a simplified implementation of his method by considering the gradient of $U$ as an approximation of the characteristic. This is equivalent to assume that the field of force $\vec{F}$ is smooth. Figure 4 shows an example of the influence of currents on our path planner.

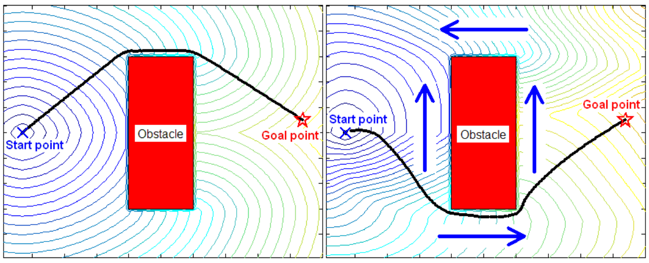


Figure 4: On the left, an isotropic Fast Marching. On the right our anisotropic version with currents symbolized by arrows.

Original Sethian's FM rapidity relies on the resolution of the simple quadratic equation 3 for $u$. Since $\tau$ appears as a square in this equation our idea is to build a new cost function $\tilde{\tau}$ linearly dependent on $u$. We split the cost function $\tau$ in two parts: $\tilde{\tau} = \tau_{obst} + \tau_{vect}$. $\tau_{obst}$ remains linked to obstacles as previously and $\tau_{vect}$ is defined as follows:

$$\tau_{vect}(i,j) = \alpha\left(1 - \frac{\langle \nabla u_{i,j} \cdot \vec{F_{i,j}} \rangle}{Q_{i,j}}\right) \geq 0 \qquad (5)$$

where $\alpha$ is a positive gain and $Q$ is a normalization term so that $\forall (i,j) \in D \left\| \frac{\langle \nabla u_{i,j} \cdot \vec{F_{i,j}} \rangle}{Q_{i,j}} \right\| \leq 1$.

It is equivalent to say that a force favors the vehicle when both force and vehicle are pointing in the same direction (see figure 5).

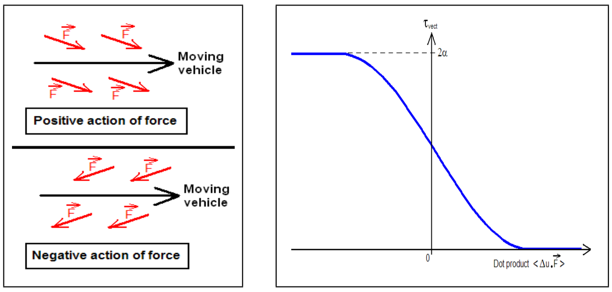


Figure 5: On the left, positive and negative actions of force applied to a mobile vehicle. On the right, appearance of $\tau_{vect}$.

With our new cost function $\tilde{\tau}$ we show in [Pêtrès *et al.*, 2005] that the path curvature radius is bounded by:

$$r \geq \frac{\inf_D\{\tau_{obst}\}}{\sup_D\{\|\nabla \tau_{obst}\|\} + \frac{2\alpha}{\inf_D\{Q\}}\|J_F\|_\infty} \qquad (6)$$

where $J_F$ is the Jacobian of $\vec{F}$ on $D$ and $\|\cdot\|_\infty$ is the $L_\infty$ norm.

Similarly to the isotropic case we have two main choices to increase the curvature radius $r$:

- Smoothing $\tau_{obst}$ to decrease $\sup_D \{ \|\nabla \tau_{obst}\| \}$.
- Smoothing the field of force $\vec{F}$ to decrease $\|J_F\|_\infty$.

**Heuristic**

Similarly to the A*, which is a BF algorithm speeded up by an heuristic, we can easily implement an isotropic or anisotropic FM*. Nevertheless we loose the nice curvature property of the resulting path.

Classical grid-search algorithms and Fast Marching share the same $O(n \log(n))$ complexity, where $n$ is the number of pixels. Since $n$ grows exponentially with the number of dimensions, these methods are limited by their memory requirement.

## 3.2 Real-time path planning

After a brief overview of incremental and multiresolution methods, we describe our novel approach to speed up graph-search algorithms. We then discuss the novel results.

### Incremental search

Incremental search reuses information from previous searches to find solutions to a series of similar search problems. Incremental methods are potentially faster than solving each search problem from scratch. This is important in our underwater path planning application since our system may have to adapt its plans continuously to changes in (its knowledge of) the world.

Focussed Dynamic A* (D*) [Stentz, 1994] and the new and simpler Lifelong Planning A* (LPA*) [Koenig *et al.*, 2004] are probably the two most popular solutions used with real robots. The closer the changes are to the goal point, the larger the advantage of the LPA* because modifications take place in the lower levels of the search tree.

In our robotic application, newly detected objects are usually observed close to the robot location. This leads us to launch the LPA* algorithm from the fixed goal point towards the moving vehicle position. To minimize the number of expensive initial iterations a scrolling map can be implemented which only moves when the robot-agent is close to the boundaries.

These incremental methods are indeed efficient but the initial computations may be unacceptable for uniformly high resolution complex maps.

### Multiresolution path planning

Multiresolution methods start with the idea that it is not necessary to represent the entire grid with a high uniform resolution.

- **Grid decomposition**

  The octree decomposition (or quadtree decomposition in 2D) is one of the most popular multiresolution approach. It is a recursive decomposition of a uniform grid into blocks. The size of blocks can depend either on the information into them (classical octree decomposition [Kambhampati and Davis, 1986]) or on their distance from the robot [Behnke, 2004].

  Octrees allow efficient partitionning of the environment since single blocks can be used to encode large empty regions. However, two main drawbacks remain. First, since the initial space (or image) is transformed in a tree data structure it is not easy to define the spatial neighborhood of each block. Secondly, paths generated by octrees are suboptimal because they are constrained to segments between the centers of blocks. The framed-quadtree technique [Yahja *et al.*, 1998] improves this problem but it is only applicable for sparse environments.

- **BF and FM on adaptive meshes**

  The method proposed by the authors is to couple the quadtree decomposition with an adaptive mesh generation. The Delaunay triangulation is a good candidate as fast and robust implementations exist.

  The input of this mesh generation is the set of nodes $q$ with their cost $\tau(q)$ given by the quadtree decomposition, and the output is a net of vertices linked to their neighbors by edges. We implemented versions of BF and FM algorithms on this kind of unstructured meshes [Sethian and Vladimirsky, 2000]. We partially overcome the problem of suboptimal paths by interpolating the distance function (computed on vertices) to the entire grid and performing a gradient descent backtracking.

- **Results**

  A first interesting result shows the much better behavior of the BF on meshes than on uniforms grids, see figures 1 and 6 to compare the results. The accuracy gap between BF and FM is not so wide any more.

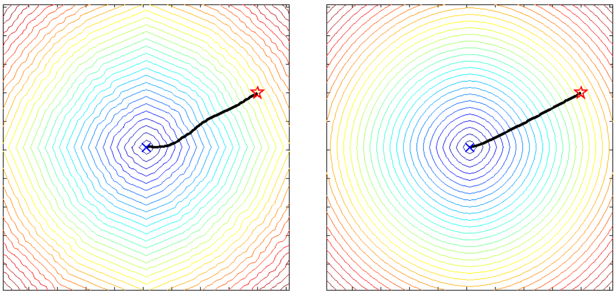


Figure 6: BF (on the left) and FM (on the right) implementations on meshes look similar (contrarily to those depicted in figure 1).

  The second interesting feature of both BF and FM on adaptive meshes compared to their implementation on uniform grids is their computation time, which is approximately divided by 1000 for similar looking paths (see figure 7).

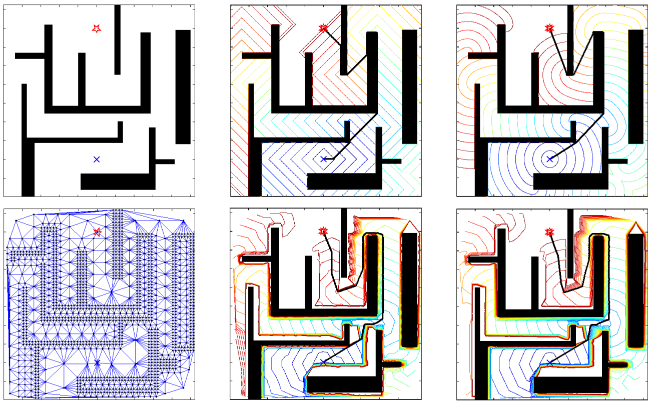


Figure 7: On the top, the original 1000x1000 image (left), optimal paths found with BF (middle) and FM (right) based methods over the entire grid of pixels take about 100 seconds. On the bottom, the adaptive mesh with only 1400 vertices (left), optimal paths found with BF (middle) and FM (right) based methods over the mesh take about 0.1 second. Computation time is divided by 1000 for similar looking paths.

At this stage we need to compare results from what we think to be among the best solutions for underwater vehicles, that is to say incremental search or BF like search methods on meshes. For this purpose analytical tools have still to be designed in order to compare the quality of paths generated by these advanced methods.

# 4 Application

As an attempt to compare results between the different approaches in a real world situation, a general COAE (Collision Obstacle Avoidance and Escape) module is being developed in the OSL laboratory. Its communication protocol and its transparent interface have been designed in a flexible way. Consequently this platform allows us to interchange different path planning methods and switch between the simulator and the real vehicle without the need of any tedious migration.

## 4.1 Control architecture

The COAE module can be viewed as a trajectory verification system. The higher-level mission planner sends a desired trajectory via a waypoint request to COAE. The deliberative part of the COAE module attempts to verify this trajectory against its local representation. If the route is clear, the trajectory is calculated and sent to the vehicle's control system. However, if the route is obstructed, or the route becomes obstructed during the manoeuvre, a course of avoidance is calculated and transmitted to the vehicle as an updated trajectory.

## 4.2 Communication protocol

The modules within the COAE use a communication protocol that provides transparent access so that the interface remains the same for both the simulator and the real vehicle. An illustration is depicted in figure 8 (taken from [Lane, 1998]). Data are generated in a common format for either real or simulated sensors and actuators of the vehicle. This data is then sent to the internal systems, which may be simulated or real devices. This architecture allows combinations of simulations to take place as new systems are built and integrated.

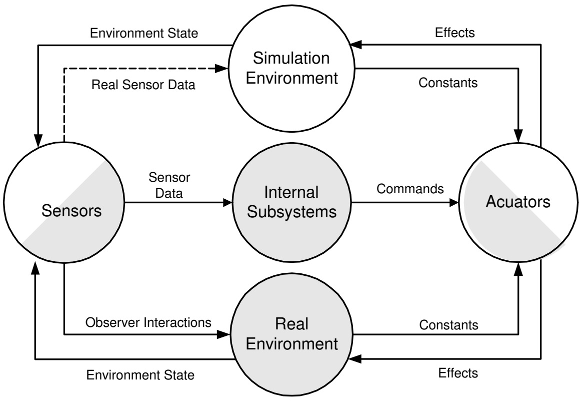


Figure 8: Using a communication protocol that provides transparent access, the interface of the system remains the same for both simulator and real vehicle.

## 4.3 Environment representation

A scrolling local map is used to compute the path in real-time. When new objects are detected in the vicinity of the robot they are inserted into the map following the occupancy grid schema. Their probability (or grey levels) are based on the number of times they have been sensed.

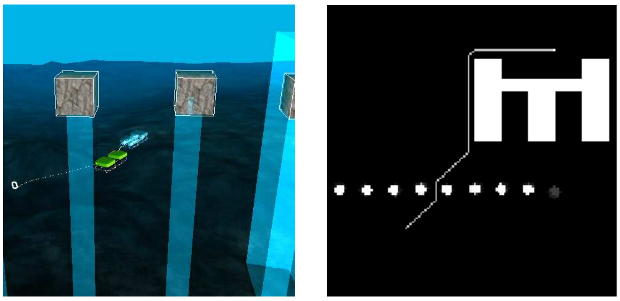


Figure 9: On the left, the OSL vehicle simulator following a trajectory designated by the deliberative obstacle avoidance module. On the right, a scrolling local map representation of the environment

## 4.4 Path planning algorithms

Due to the availability of the sensors, we have initially reduced the complexity of the underwater environment to a depth dependent horizontal scrolling map that moves locally with the vehicle (seen as an agent) inside the environment.

Path planning algorithms are applied to the planes generated through this mapping technique. The implementation of our approach on the 3D domain representation is straight forward as soon as more sophisticated sensors become available.

This application is used to test and verify algorithms and new theoretical approaches in a real situation. This can be easily done by using the transparency provided by the interface oriented architecture.

Additionally, when environmental features, like water currents or levels of terrain, are sensed they can be modelled into the system by weighting the input space and representing the estimated difficulty of traversing them. In this way, we can prioritize certain places in the map and penalize others by playing with the inputs of the algorithms.

Finally, by computing the lower bound of the path curvature radius, we can smooth the input space accordingly until the trajectory is reliable for a specific underwater vehicle.

# 5 Conclusion

The underwater world is a very demanding environment for path planning algorithms. However,a great effort is currently being made to develop autonomous systems as underwater technology becomes more mature. Several key issues for underwater path planning are improved on in this paper.

Reliability of path planners is improved by introducing an efficient algorithm called Fast Marching. We present a solution to take the vehicle kinematics into account by smoothing the map of the environment. A practical implementation of anisotropic Fast Marching is proposed to make our path planning method robust to underwater currents.

Incremental search algorithms are efficient in computing environmental modifications without recalculating from scratch. However, in this case, because we process all the information, it is necessary to deal with a high number of states. A mesh conversion of the input data can drastically reduce the computation time by reducing the input data set of the path search algorithm. However, this reduction produces a loss of information that can affect the optimality of the resulting path. This discussion highlights the need for future work to find an analytical tool to measure the path acceptability.

A flexible real-time simulator architecture based on transparency with vehicle interfaces has been demonstrated to be an excellent platform for testing path planning algorithms before moving onto the real vehicle.

## 6 Future work

Off shore trials for BAUUV project are planned for the end of year 2005. The OSL experimental vehicle RAUVER [Hamilton, 2005] will be used as the platform during these trials. Although RAUVER is a hovering vehicle, it has been adapted to react as a torpedo. This will allow us to demonstrate the path planning algorithms in a real vehicle situation.

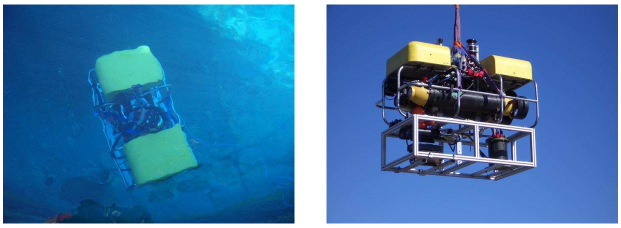


Figure 10: RAUVER, the autonomous underwater prototype of Ocean Systems Laboratory.

## Acknowledgments

This research was sponsored by MoD, under contracts "Battlespace Access for Unmanned Underwater Vehicles" (BAUUV), inside the workpackage Strategies for Collision Obstacle Avoidance and Escape (contract number WP220304 CCMM2, monitored by SEA (Group) Ltd.).

## References

[Behnke, 2004] Sven Behnke. Local multiresolution path planning. In *Proceedings of 7th RoboCup International Symposium*, Padua, Italy, 2004.

[Canny, 1988] J. F. Canny. *The complexity of Robot Motion Planning*. MIT Press, 1988.

[CodaOctopus Ltd., 2004] Echosounder, 2004. "http://www.codaoctopus.com/3d_ac_im/".

[Cohen and Kimmel, 1997] L.D. Cohen and R. Kimmel. Global minimum for active contour models: A minimal path approach. *Internationnal Journal of Computer Vision*, 24(1):57–78, 1997.

[D.M.Lane *et al.*, 2001] D.M.Lane, J. Falconer G, and G. Randall. Interoperability and synchronisation of distributed hardware-in-the-loop simulation for underwater robot development: Issues and experiments. In *IEEE International Conference on Robotics and Automation*, Seoul, Corea, May 2001.

[Hamilton, 2005] K. Hamilton. Rauver mkii - autonomous hover-capable intervention and close inspection vehicle, 2005. http://www.ece.eps.hw.ac.uk/oceans/.

[Kambhampati and Davis, 1986] S. Kambhampati and L. S. Davis. Multiresolution path planning for mobile robots. *IEEE journal of Robotics and Automation*, RA-2(3):135–145, September 1986.

[Koenig *et al.*, 2004] S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning a*. *Artificial Intelligence*, 155(1-2):93–146, May 2004.

[Korf, 1998] Richard E. Korf. Artificial intelligence search algorithms. *CRC Handbook of Algorithms and Theory of Computation*, pages 36–1–36–20, 1998.

[Lane, 1998] D.M. Lane. Mixing simulations and real subsystems for subsea robot development. 1998.

[Latombe, 1991] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publisher, 1991.

[LaValle, 2005] Steven M. LaValle. *Planning Algorithms*. [Online], 2005. Available at http://msl.cs.uiuc.edu/planning/.

[Liu and Arimoto, 1992] Y. Liu and S. Arimoto. Path planning using a tangent graph for mobile robots among polygonal and curved obstacles. *The International Journal of Robotics Research*, pages 312–317, 1992.

[Petillot *et al.*, 2001] Y. Petillot, I. Tena Ruiz, and D.M. Lane. Underwater vehicle obstacle avoidance and path planning using a multi-beam forward looking sonar. *IEEE J. of Oceanic Engineering*, 26:240–251, 2001.

[Pêtrès *et al.*, 2005] C. Pêtrès, Y. Pailhas, J. Evans, Y. Petillot, and D. Lane. Underwater path planning using fast marching algorithms. In *Proceedings of Oceans 2005 Conference*, Brest, France, June 21 2005. In press.

[Rouy and Tourin, 1992] E. Rouy and A. Tourin. A viscosity solution approach to shape-from-shading. *SIAM Journal of Numerical Analyzis*, 29:867–884, 1992.

[Sethian and Vladimirsky, 2000] J.A. Sethian and A. Vladimirsky. Fast methods for the eikonal and related hamilton-jacobi equations on ustructured meshes. *Applied Mathematics*, 97(11):5699–5703, May 23 2000.

[Sethian, 1999] J.A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, Cambridge, Massachusetts, 1999.

[Stentz, 1994] Anthony Stentz. Optimal and efficient path planning for partially-known environment. IEEE International Conference on Robotics and Automation, May 1994.

[Svestka and Overmars, 1998] P. Svestka and M.H. Overmars. Probabilistic path planning. In *Lecture Notes in Control and Information Sciences 229*. Springer, 1998.

[Takahashi and Schilling, 1989] O. Takahashi and R.J. Schilling. Motion planning in a plane using generalized voronoi diagrams. pages 143–150. IEEE International Conference on Robotics and Automation, 1989.

[Vladimirsky, 2003] A. Vladimirsky. Ordered upwind methods for static hamilton-jacobi equation: Theory and algorithms. *SIAM Journal of Numerical Analysis*, 41(1):325–363, 2003.

[Y. Wang, 1999] D.M. Lane Y. Wang. An adaptive constrained optimisation approach for robot path planning in n-dimensions. *International Journal of Systems Science*, 1999.

[Yahja *et al.*, 1998] Alex Yahja, Anthony Stentz, Sanjiv Singh, and Barry Brumitt. Framed-quadtree path planning for mobile robots operating in sparse environments. In *Proceedings IEEE Conference on Robotics and Automation (ICRA '98)*, Leuven, Belgium, May 1998.

[Zimmerman, 2004] Matthew J. Zimmerman. A 3d, forward looking, phased array, obstacle avoidance sonar for autonomous underwater vehicles. *http://www.farsounder.com/products/FS-3/index.php*, Farsounder Inc, 2004.

# Speeding Up Learning in Real-time Search via Automatic State Abstraction

**Vadim Bulitko** and **Nathan Sturtevant** and **Maryia Kazakevich**
Department of Computing Science, University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
{bulitko|nathanst|maryia}@cs.ualberta.ca

## Abstract

Situated agents which use learning real-time search are well poised to address challenges of real-time path-finding in robotic and computer game applications. They interleave a local lookahead search with movement execution, explore an initially unknown map, and converge to better paths over repeated experiences. In this paper, we first investigate how three known extensions of the most popular learning real-time search algorithm (LRTA*) influence its performance in a path-finding domain. Then, we combine automatic state abstraction with learning real-time search. Our scheme of dynamically building a state abstraction allows us to generalize updates to the heuristic function, thereby speeding up learning. The novel algorithm converges up to 80 times faster than LRTA* with only one fifth of the response time of A*.

## 1 Introduction

In this paper, we consider a simultaneous planning and learning problem. More specifically, we require an agent to navigate on an initially unknown map under real-time constraints. As an example, consider a robot driving to work every morning. Imagine the robot to be a newcomer to the town. The first route the robot finds may not be optimal because the traffic jams, road conditions, and other factors are initially unknown. With a passage of time, the robot continues to learn and eventually converges to a nearly optimal commute. Note that planning and learning happen while the robot is driving and therefore are subject to time constraints.

Present-day mobile robots are often plagued by localization problems and power limitations, but simulation counterparts already allow researchers to focus on the planning and learning problem. For instance, the RoboCup Rescue simulation [Kitano *et al.*, 1999] requires real-time planning and learning with multiple agents mapping out unknown terrain.

Similarly, many current-generation real-time strategy games employ *a priori* known maps. Full knowledge of the maps enables complete search methods such as A*. Prior availability of the maps allows path-finding engines to precompute data (e.g., visibility maps) to speed up on-line navigation. Neither technique will be applicable in forthcoming generations of commercial and academic games [Buro, 2002] which will require the agent to cope with the initially unknown maps via exploration and learning during the game.

To compound the problem, the dynamic A* (D*) [Stenz, 1995] and D* Lite [Koenig & Likhachev, 2002], frequently used in robotics, work well when the robot's movements are slow with respect to its planning speed. In real-time strategy games, however, the AI engine can be responsible for hundreds to thousands of agents traversing the map simultaneously and the planning cost becomes a major factor. We thus discuss the following three questions.

First, how planning time per move and, particularly the first-move delay, can be minimized so that each agent moves smoothly and responds to user requests nearly instantly. Second, given the local nature of the agent's reasoning and the initially unknown terrain, how the agent can learn a better global path. Third, how learning can be accelerated so that only a few repeated path-finding experiences are needed before converging to a near-optimal path.

In the rest of the paper, we first make the problem settings concrete and derive specific performance metrics based on the questions above. Then we discuss the challenges that incremental heuristic search faces when applied to real-time path-finding. As an alternative, we will review a family of learning real-time search algorithms which are well poised for use by situated agents. Starting with the most popular real-time search algorithm, LRTA*, we make our initial contribution by evaluating three known complementary extensions in the context of real-time path-finding. The resulting algorithm, LRTS, exhibits a 46-fold speed-up in the travel until convergence while having one sixth of the first-move delay of an A* agent. Despite the improvements, the learning and search still happen on a large ground-level map. Thus, all states are considered distinct and no generalization is used in learning. We then make the primary contribution by introducing an effective mechanism for building and repairing a hierarchical abstraction of the map. This allows us to constrain the search space, reduce the amount of learning required for convergence, and generalize learning in each individual state onto neighboring states. The novel algorithm, PR-LRTS, is then empirically evaluated.

## 2 Problem Formulation

In this paper, we focus on a particular real-time path-finding task. Specifically, we will assume that the agent is tasked to travel from the start state $(x_s, y_s)$ to the goal state $(x_g, y_g)$.

The coordinates are on a two-dimensional rectangular grid. In each state, up to eight moves are available leading to the eight immediate neighbors. Each straight move (i.e., north, south, west, east) has the *travel cost* of $1$ while each diagonal move has the travel cost of $\sqrt{2}$. Each state on the map can be passable or occupied by a wall. In the latter case, the agent is unable to move into it. Initially, the map in its entirety is unknown to the agent. In each state $(x, y)$ the agent can see the status (occupied/free) of the neighborhood of the *visibility radius* $v$: $\{(x', y') \mid |x' - x| \leq v \;\; \& \;\; |y' - y| \leq v\}$. The agent can choose to remember the observed parts of the map and use that information in subsequent planning.

A *trial* is defined as a finite sequence of moves the agent takes to travel from the start to the goal state. Once the goal state is reached, the agent is reset to the start state and the next trial begins. A *convergence run* is defined as the first sequence of trials such that the agent does not learn or explore anything new on the subsequent trials.

Each problem instance is fully specified by the map and start and goal coordinates. We then run the agent until convergence and measure the cumulative travel cost of all moves (*convergence travel*), the average delay before the first move (*first-move lag*), and the length of the path found on the final trial (*final solution length*). The last measure is used to compute the *amount of suboptimality* defined as percentage of the length excess.

## 3 Incremental Search

Classical A* search is inapplicable due to an initially unknown map. Specifically, it is impossible for the agent to plan its path through state $(x, y)$ unless it is either positioned within the visibility radius of the state or has visited this state on a prior trial.

A simple solution to this problem is to generate the initial path under the assumption that the unknown areas of the map contain no occupied states (the free space assumption [Koenig, Tovey, & Y., 2003]). With the *octile distance*[1] as the heuristic, the initial path is close to the straight line since the map is assumed empty. The agent follows the existing path until it runs into an occupied state. During the travel, it updates the explored portion of the map in its memory. Once the current path is blocked, A* is invoked again to generate a new complete path from the current position to the goal. The process repeats until the agent arrives at the goal. It is then reset to the start state and a new trial begins. The convergence run ends when no new states are seen.

To increase efficiency, several methods of re-using information over subsequent planning episodes have been suggested. The two popular versions are D* [Stenz, 1995] and D* Lite [Koenig & Likhachev, 2002]. Unfortunately, these enhancements do not reduce the first-move lag time. Specifically, after the agent is given the destination coordinates, it has to conduct an A* search from its position to the destination before it can move. Even on small maps, this delay can be substantial. Consider, for instance, a map from BioWare's game "Baldur's Gate" shown in Figure 1. Before an A*-controlled agent can make its first move, a complete path from start to goal state has to be generated. This is in contrast to LRTA* [Korf, 1990], which only performs a small local search to select the first move. As a result, several orders of magnitude more agents can calculate and make their first move in the time it takes one A* agent.

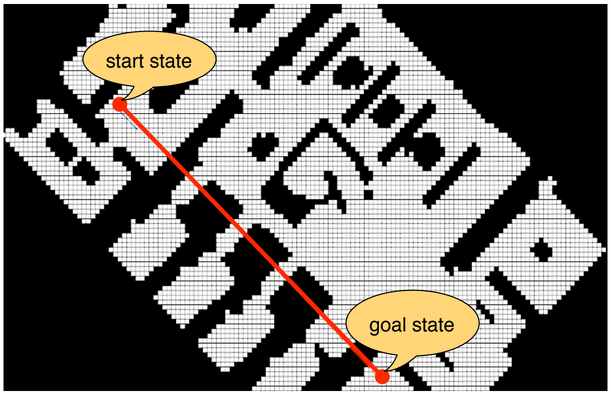


Figure 1: A sample map from a BioWare's game.

A thorough comparison between D* Lite and an extended version of LRTA* is found in [Koenig, 2004]. It investigates the conditions under which real-time search outperform incremental search. Since our paper focuses on real-time search and uses incremental search only as a reference point and because D*/D* Lite does not reduce the first-move lag on the final trial, we use the simpler incremental A* in our experiments.

## 4 Real-time Search

Real-time search was pioneered by [Korf, 1990] with the presentation of RTA* and LRTA* algorithms. Unlike A*, which can freely traverse its open list, each RTA*/LRTA* search assumes the agent to be in a single current state that can be changed only by taking moves and, thereby, incurring travel cost. From its state, the agent conducts a full-width fixed-depth local forward search (called lookahead) and, similarly to minimax game-playing agents, uses its heuristic $h$ to evaluate the frontier states (Figure 2). It then takes the first move towards the most promising frontier state (i.e., the state with the lowest $g + h$ value where $g$ is the cost of traveling from the current state to the frontier state) and repeats the cycle. The initial heuristic is set to the octile distance. On every move, the heuristic value of the current state is increased to the $g + h$ value of the most promising state.[2] As discussed in [Barto, Bradtke, & Singh, 1995], this operation is analogous to the "backup" step used in value iteration reinforcement learning agents with the learning rate $\alpha = 1$ and no discounting. LRTA* will refine an initial admissible heuristic to the perfect heuristic along a shortest path. This constitutes a convergence run. The updates to the heuristic also guarantee that LRTA* will not get trapped in infinite cycles. We

[1] Octile distance is a natural adaptation of Euclidian distance to the case of the eight discrete moves and can be computed in a closed form.

[2] As [Shimbo & Ishida, 2003], we do not decrement $h$ of any state. Convergence to optimal paths is still possible as the initial heuristic is admissible but the convergence is accelerated.

```
LRTA*
initialize the heuristic: $h \leftarrow h_0$
reset the current state: $s \leftarrow s_{\text{start}}$
while $s \neq s_{\text{goal}}$ do
  expand children one move away
  find the state $s'$ with the lowest $f = g + h$
  update $h(s)$ to $f(s')$
  execute the action to get to $s'$
end while
```

Figure 2: LRTA* algorithm with the lookahead of one.

Table 1: **Top:** Effects of the lookahead depth $d$ on deliberation time per unit of distance and average travel per trial in LRTA*. **Middle:** Effects of the optimality weight $\gamma$ on suboptimality of the final solution and total travel in LRTA* ($d = 1$). **Bottom:** Effects of learning quota $T$ on amount of first trial and total travel.

| $d$ | Deliberation per move (ms) | Travel per trial |
|---|---|---|
| 1 | 0.0087 | 661.5 |
| 3 | 0.0215 | 241.8 |
| 5 | 0.0360 | 193.3 |
| 7 | 0.0514 | 114.9 |
| 9 | 0.0715 | 105.8 |

| $\gamma$ | Suboptimality | Convergence travel |
|---|---|---|
| 0.1 | 6.19% | 9,300 |
| 0.3 | 4.92% | 8,751 |
| 0.5 | 2.41% | 9,435 |
| 0.7 | 1.23% | 13,862 |
| 0.9 | 0.20% | 25,507 |
| 1.0 | 0.00% | 31,336 |

| $T$ | First trial travel | Convergence travel |
|---|---|---|
| 0 | 434 | 457 |
| 10 | 413 | 487 |
| 50 | 398 | 592 |
| 1,000 | 390 | 810 |
| 5,000 | 235 | 935 |

```
LRTS($d, \gamma, T$)
1  initialize: $h \leftarrow h_0$, $s \leftarrow s_{\text{start}}$, $u \leftarrow 0$
2  while $s \neq s_{\text{goal}}$ do
3    expand children $i$ moves away, $i = 1 \ldots d$
4        on level $i$, find state $s_i$ with the lowest $f = \gamma \cdot g + h$
5    update $h(s) \leftarrow \max_{1 \leq i \leq d} f(s_i)$
6    increase amount of learning $u$ by $|\Delta h|$
7    if $u \leq T$ then
8      execute $d$ moves to get to $s_d$
9    else
10     execute $d$ moves to backtrack to previous $s$, set $u = T$
11   end if
12 end while
```

Figure 3: LRTS algorithm unifies LRTA*, $\varepsilon$-LRTA*, and SLA*T.

now make the first contribution of this paper by evaluating the effects of three known complementary extensions in the context of real-time path-finding.

First, increasing lookahead depth increases the amount of deliberation per move but, on average, causes the agent to take better moves, thereby finding shorter paths. This effect is demonstrated in Table 1 with averages of 50 convergence runs over 10 different maps. Hence, the lookahead depth can be selected dynamically depending on the amount of CPU time available per move and the ratio between the planning and moving speeds [Koenig, 2004].

Second, the distance from the current state to the state on the frontier (the $g$-function) can be weighted by the $\gamma \in (0, 1]$. This allows us to trade-off the quality of the final solution and the convergence travel. This extension of LRTA* is equivalent to scaling the initial heuristic by the constant factor of $1+\varepsilon = 1/\gamma$ [Shimbo & Ishida, 2003]. Bulitko [2004] proved that $\gamma$-weighted LRTA* will converge to a solution no worse than $1/\gamma$ of optimal. In practice, much better paths are found (Table 1). A similar effect is observed in weighted A*: increasing the weight of $h$ (i.e., decreasing the relative weight of $g$) dramatically reduces the number of states generated, at the cost of longer solutions [Korf, 1993].

Third, backtracking within LRTA* was first proposed in [Shue & Zamani, 1993]. Their SLA* algorithm used the lookahead of one and the same update rule as LRTA*. However, upon updating (i.e., increasing) the heuristic value in a state, the agent moved (i.e., backtracked) to its previous state. Backtracking increases travel on the first trial but reduces the convergence travel (Table 1). Note that backtracking does not need to happen after *every* update to the heuristic function. SLA*T, introduced in [Shue, Li, & Zamani, 2001], backtracks only after the cumulative amount of updates to the heuristic function made on a trial exceeds the learning quota ($T$). We will use an adjusted implementation of this idea which enables us to bound the length of the path found on the *first* trial by $(h^*(s_{\text{start}}) + T)/\gamma$ where $h^*(s_{\text{start}})$ is the actual shortest distance between the start and goal.

An algorithm combining all three extensions (lookahead $d$, optimality weight $\gamma$, and backtracking control $T$) operates as follows. In the current state $s$, it conducts a lookahead search of depth $d$ (line 3 in Figure 3). At each ply, it finds the most promising state (line 4). Assuming that the initial heuristic $h_0$ is admissible, we can safely increase $h(s)$ to the maximum among the $f$-values of promising states for all plies (line 5). If the total learning amount $u$ exceeds the learning quota $T$, the agent backtracks to the previous state (lines 7, 10). Otherwise, it executes $d$ moves forward towards the most promising frontier state (line 8). In the rest of the paper, we will refer to this combination of three extensions as LRTS (learning real-time search).

LRTS with domain-tuned parameters converges two orders of magnitude faster than LRTA* while finding paths within 3% of optimal. At the same time, LRTS is about five times faster on the first move than incremental A* as shown in Table 2. Despite the improvements, LRTS takes hundreds of moves before convergence is achieved, even on smaller maps with only a few thousand states.

## 5 Novel Method: Path-refinement LRTS

The problem with LRTA* and LRTS described in the previous section stems from the fact that the heuristic is learnt in a tabular form. Each entry in the table corresponds to a single state and no generalization is attempted. Consequently, thousands of heuristic values have to be incrementally computed

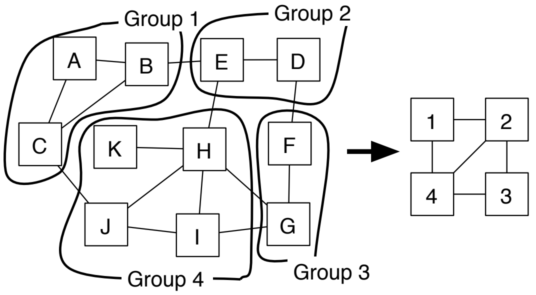


Figure 4: The process of abstracting a graph.

via individual updates – one per move of the agent. Thus, significant traveling costs are incurred before the heuristic function converges. This is not the way humans and animals appear to learn a map. We do not learn at the micro-level of individual states but rather reason over *areas* of the map as if they were single entities. Thus, the primary contribution of this paper is extension of learning real-time heuristic search with a state abstraction mechanism.

## 5.1 Building a State Abstraction

State abstraction has been studied extensively in reinforcement learning [Barto & Mahadevan, 2003]. While our approach is fully automatic, many algorithms, such as MAXQ [Dietterich, 1998], rely on manually engineered hierarchical representation of the space.

Automatic state abstraction has precedents in heuristic search and path-finding. For instance, Hierarchical A* [Holte *et al.*, 1995] and AltO [Holte *et al.*, 1996] used abstraction to speed up classical search algorithms. Our approach to automatically building abstractions from the underlying state representation is similar to Hierarchical A*.

We demonstrate the abstraction procedure on a hand-traceable micro-example in Figure 4. Shown on the left is the original graph of 11 states. In general, we can use a variety of techniques to abstract the map, and we can also process the states in any order. Some methods and orderings may, however, work better in specific domains. In this paper, we look for cliques in the graph.

For this example, we begin with the state labeled A, adding it and its neighbors, B and C, to abstract group 1, because they are fully connected. Their group becomes a single state in the abstract graph. Next we consider state D, adding its neighbor, E, to group 2. We do not add H because it is not connected to D. We continue to state F, adding its neighbor, G, to group 3. States H, I, and J are fully connected, so they become group 4. Because state K can only be reached via state H, we add it to group 4 with H. If all neighbors of a state have already been abstracted, that state will become a single state in the abstract graph. As states are abstracted, we add edges between existing groups. Since there is an edge between B and E, and they are in different groups, we add an edge between groups 1 and 2 in the abstract graph. We proceed similarly for the remaining inter-group edges. The resulting abstracted graph of 4 states is shown in the right portion of the figure.

Table 2: Incremental A*, LRTA*, LRTS averaged over 50 runs on 10 maps. The average solution length is 59.5. LRTA* is with the lookahead of 1. LRTS is with $d = 10, \gamma = 0.5, T = 0$. All timings are taken on a dual G5, 2.0GHz with gcc 3.3.

| Algorithm | 1st move time | Conv. travel | Suboptimality |
|---|---|---|---|
| A* | 5.01 ms | 186 | 0.0% |
| LRTA* | 0.02 ms | 25,868 | 0.0% |
| **LRTS** | **0.93 ms** | **555** | **2.07%** |

We repeat the process iteratively, building an abstraction hierarchy until there are no edges left in the graph. If the original graph is connected, we will end up with a single state at the *highest* abstraction level, otherwise we will have multiple disconnected states. Assuming a sparse graph of $V$ vertices, the size of all abstractions is at most $O(V)$, because we are reducing the size of each abstraction level by at least a factor of two. The cost of building the abstractions is $O(V)$. Figure 5 shows a micro example.

Because the graph is sparse, we represent it with a list of states and edges as opposed to an adjacency matrix. When abstracting an entire map, we first build its connectivity graph and then abstract this graph in two passes. Our abstractions are most uniform if we remove 4-cliques in a first pass, and then abstract the remaining states in a second pass.

## 5.2 Repairing Abstraction During Exploration

A new map is initially unknown to the agent. Under the free space assumption, the unknown areas are assumed empty and connected. As the map is explored, obstacles are found and the initial abstraction hierarchy needs to be repaired to reflect these changes. This is done with four operations: remove-node, remove-edge, add-node, and add-edge. We describe the first two in detail here.

In the abstraction, each edge either abstracts into another edge in the parent graph, or becomes internal to a state in the parent graph. Thus, each abstract edge must maintain a count of how many edges it is abstracting from the lower level. When remove-edge removes an edge, it decrements the count of edges abstracted by the parent edge, and recursively removes the parent if the count falls to zero. If an edge is abstracted into a state in the parent graph, we add that state to a *repair queue* to be handled later. The remove-node operation is similar. It decrements the number of states abstracted by the parent, removing the parent recursively if needed, and

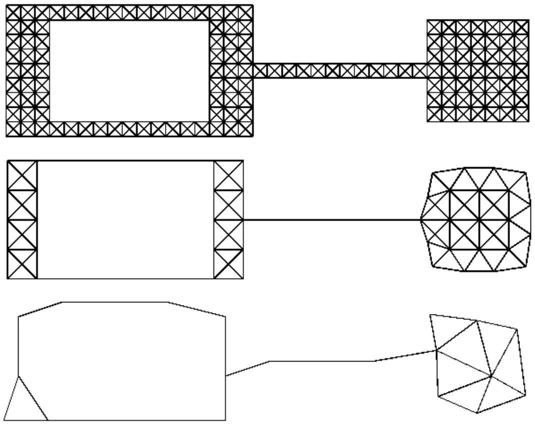


Figure 5: Abstraction levels 0, 1, and 2 of a toy map. The number of states is 206, 57, and 23 correspondingly.

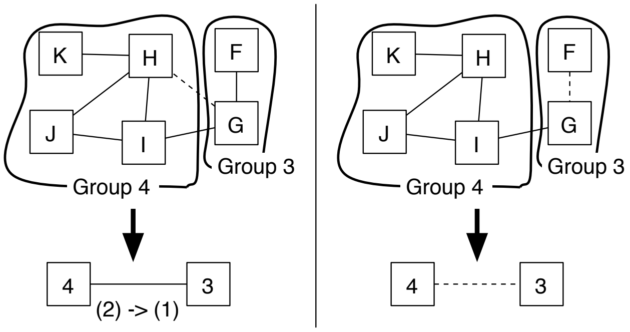


Figure 6: Repairing abstractions.

then adds the parent state to a repair queue. This operation also removes any edges incident to the state.

When updating larger areas of the map in one pass, using a repair queue allows us to share the cost of the additional steps required to perform further repairs in the graph. Namely, there is no need to completely repair the abstraction if we know we are going to make other changes. The repair queue is sorted by abstraction level in the graph to ensure that repairs do not conflict.

In a graph with $n$ states, the remove-node and remove-edge operations can, in the worst case, take $O(\log n)$ time. However, their time is directly linked to how many states are affected by the operation. If there is one edge that cuts the entire graph, then removing it will take $O(\log n)$ time. However, in practice, most removal operations have a local influence and take time $O(1)$. Handling states in the repair queue is an $O(\log n)$ time operation in the worst case, but again, we only pay this cost when we are making changes that affect the connectivity of the entire map. In practice, there will be many states for which we only need to verify their internal connectivity.

Figure 6 illustrates the repair process. Shown on the left is a subgraph of the 11-state graph from Figure 4. When in the process of exploration it is found that state H is not reachable from G, the edge (H,G) will be removed (hence shown with a dashed line). Thus, the abstraction hierarchy needs to be repaired. The corresponding abstracted edge (4,3) represents two edges: (G,H) and (G,I). When (G,H) is removed, the edge count of (4,3) is decremented from 2 to 1.

Suppose it is subsequently discovered that edge (F,G) is also blocked. This edge is internal to the states abstracted by group 3 and so we add group 3 to the repair queue. When we handle the repair queue, we see that states abstracted by group 3 are no longer connected. Because state G has only a single neighbor, we can merge it into group 4, and leave F as the only state in group 3. When we merge state G into group 4, we also delete the edge between groups 3 and 4 in the abstract graph (right part of Figure 6).

### 5.3 Abstraction in Learning Real-time Search

Given the efficient on-line mechanism for building state abstraction, we propose, implement, and evaluate a new algorithm called PR-LRTS (Path-Refining Learning Real-Time Search). A PR-LRTS agent operates at several levels of abstraction. Each level from 0 (the ground level) to $N \geq 0$ is "populated" with A* or LRTS. At higher abstract levels, the heuristic distance between any two states is Euclidian distance between them, where the location of a state is the average location of the states it abstracts. This heuristic is not admissible with respect to the actual map. Octile distance is used as the heuristic at level 0.

```
PR LRTS
assign A*/LRTS to abstraction levels $0, \ldots, N$
initialize the heuristic for all LRTS-levels
reset the current state: $s \leftarrow s_{\text{start}}$
reset abstraction level $\ell = 0$
while $s \neq s_{\text{goal}}$ do
    if algorithm at level $\ell$ reached the end of corridor $c_\ell$ then
        if we are at the top level $\ell = N$ then
            run algorithm at level $N$
            generate path $p_N$ and corridor $c_{N-1}$
            go down abstraction level: $\ell = \ell - 1$
        else
            go up abstraction level: $\ell = \ell + 1$
        end if
    else
        run algorithm at level $\ell$ within corridor $c_\ell$
        generate path $p_\ell$ and corridor $c_{\ell-1}$
        if $\ell = 0$ then execute path $p_0$
        else continue refinement: $\ell = \ell - 1$
    end if
end while
```

Figure 8: Path refinement learning real-time search.

At the beginning of each trial, no path has been constructed at any level. Thus, the algorithm at level $N$ is invoked. It works at the level $N$ and produces the path $p_N$. In the case of A*, $p_N$ is a complete path from the $N$-level parent of the current state to the $N$-level parent of the goal state. In the case of LRTS, $p_N$ is the first $d$ steps towards the abstracted goal state at level $N$. The process now repeats at level $N-1$ resulting in path $p_{N-1}$. But, when we repeat the process, we restrict any planning process at level $N-1$ to a *corridor* induced by the abstract path at level $N$. Formally, the corridor $c_{N-1}$ is the set of all states which are abstracted by states in $p_N$. To give more leeway for movement and learning, the corridor can also be expanded to include any states abstracted by the $k$–step neighbors of $p_N$. In this paper, we choose $k = 1$. While executing $p_0$, new areas of the map may be seen. The state abstraction hierarchy will be repaired as previously described. This path-refining approach, summarized in Figure 8, benefits path-finding in three ways.

First, algorithms running at the higher levels of abstraction reason over a much smaller (abstracted) search space (e.g., Figure 5). Consequently, the number of states expanded by A* is smaller and the execution is faster.

Second, when LRTS learns at a higher abstraction level, it maintains the heuristic at that level. Thus, a single update to the heuristic function effectively influences the agent's behavior on a set of ground-level states. Such a generalization via state abstraction reduces the convergence time.

Third, algorithms operating at lower levels are *restricted* to the corridors $c_i$. This focuses their operation on more promising areas of the state space and speeds up search (in the case of A*) and convergence (in the case of LRTS).

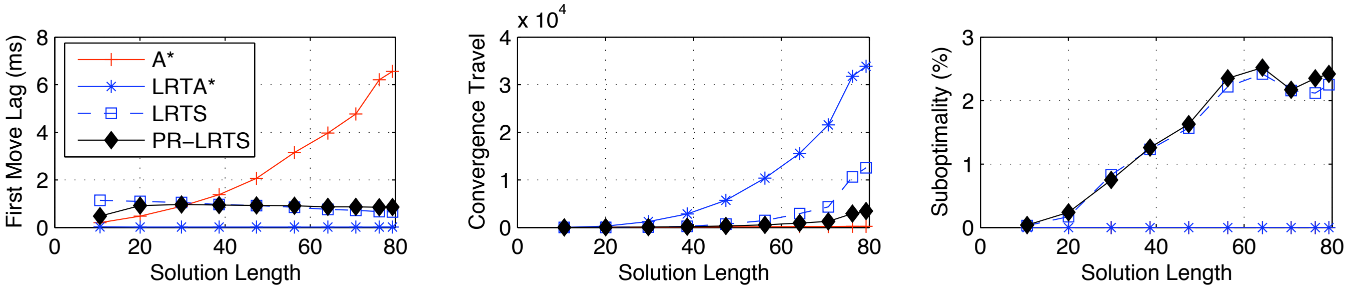


Figure 7: First-move lag, convergence travel, and final solution suboptimality over the optimal solution length.

## 6 Empirical Evaluation

We evaluated the benefits of state abstraction in learning real-time heuristic search by running PR-LRTS against the incremental A* search, LRTA*, and LRTS for path-finding on 9 maps from Bioware's "Baulder's Gate" game. The maps ranged in size from 64 × 60 to 144 × 148 cells, averaging 3212 passable states. Each state had up to 8 passable neighbors and the agent's visibility radius was set to 10. LRTS and PR-LRTS have been run with a number of parameters and a representative example is found in Table 3. Starting from the top entry in the table: incremental A* shows an impressive convergence travel (only three times longer than the shortest path) but has a substantial first-move lag of 5.01 ms. LRTA* with the lookahead of 1 is about 250 times faster but travels 140 times more before convergence. LRTS($d = 10$, $\gamma = 0.5$, $T = 0$) has less than 20% of A*'s first-move lag and does only 2% of LRTA*'s travel. State abstraction in PR-LRTS (with A* at level 0 and LRTS(5,0.5,0.0) at level 1) reduces the convergence travel by an additional 40% while preserving the lag time of LRTS. The trade-offs are summarized in Table 4. Figure 7 plots the performance measures over the optimal solution length. PR-LRTS appears to scale well and its advantages over the other algorithms become more pronounced on more difficult problems.

PR-LRTS exhibits approximately the same level of suboptimality and first-move lag as LRTS but converges up to 3.6 times faster. Compared to incremental A*, PR-LRTS has up to six times shorter first-move lag. Compared to LRTA*, PR-LRTS converges up to 10 times faster.

## 7 Conclusions and Future Work

We have considered some of the challenges imposed by real-time path-finding as faced by mobile robots in unknown terrain and characters in computer games. Such situated agents must react quickly to the commands of the user while at the same time exhibiting reasonable behavior. As the first result, combining three complementary extensions of the most popular real-time search algorithm, LRTA*, yielded substantially faster convergence for path-finding tasks. We then introduced state abstraction for learning real-time search. The dynamically built abstraction levels of the map increase performance by: (i) constraining the search space, (ii) reducing the amount of updates made to the heuristic function, thereby accelerating convergence, and (iii) generalizing the results of learning over neighboring states.

Table 3: Typical results averaged over 50 convergence runs on 10 maps. The average shortest path length is 59.6.

| Algorithm | 1st move time | Conv. travel | Suboptimality |
|---|---|---|---|
| A* | 5.01 ms | 186 | 0.0% |
| LRTA* | 0.02 ms | 25,868 | 0.0% |
| LRTS | 0.93 ms | 555 | 2.07% |
| **PR-LRTS** | **0.95 ms** | **345** | **2.19%** |

Table 4: Summary of the trade-offs.

| Algorithm | compared to PR-LRTS |
|---|---|
| A* | slower per move, converges faster, optimal |
| LRTA* | converges slower, faster per move, optimal |
| LRTS | converges slower |

Future research will investigate if the savings in memory gained by learning at a higher abstraction level will afford application of PR-LRTS to moving target search. The previously suggested MTS algorithm [Ishida & Korf, 1991] requires learning $O(n^2)$ heuristic values which can be prohibitive even for present-day commercial maps. Additionally, we are planning to investigate how the A* component of PR-LRTS compares with the incremental updates to the routing table in Trailblazer search [Chimura & Tokoro, 1994] and its hierarchical abstract map sequel [Sasaki, Chimura, & Tokoro, 1995]. Finally, we will investigate sensitivity of PR-LRTS to the control parameters as well as the different abstraction schemes in path-finding and other domains.

## Acknowledgments

The funding was provided by the National Science and Engineering Research Council, Alberta Ingenuity Centre for Machine Learning, iCORE, and the University of Alberta.

## References

[Barto & Mahadevan, 2003] Barto, A. G., and Mahadevan, S. 2003. Recent advances in hierarchical reinforcement learning. *DEDS* 13:341 – 379.

[Barto, Bradtke, & Singh, 1995] Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to act using real-time dynamic programming. *AIJ* 72(1):81–138.

[Bulitko, 2004] Bulitko, V. 2004. Learning for adaptive real-time search. Technical Report http://arxiv.org/abs/cs.AI/0407016, Computer Science Research Repository (CoRR).

[Buro, 2002] Buro, M. 2002. ORTS: A hack-free RTS game environment. In *Proceedings of Int. Comp. and Games Conf.*, 12.

[Chimura & Tokoro, 1994] Chimura, F., and Tokoro, M. 1994. The Trailblazer search: A new method for searching and capturing moving targets. In *Proceedings of AAAI*, 1347–1352.

[Dietterich, 1998] Dietterich, T. G. 1998. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of ICML*, 118–126.

[Holte *et al.*, 1995] Holte, R.; Perez, M.; Zimmer, R.; and MacDonald, A. 1995. Hierarchical A*: Searching abstraction hierarchies efficiently. Technical Report tr-95-18, U. of Ottawa.

[Holte *et al.*, 1996] Holte, R.; Mkadmi, T.; Zimmer, R. M.; and MacDonald, A. J. 1996. Speeding up problem solving by abstraction: A graph oriented approach. *AIJ* 85(1-2):321–361.

[Ishida & Korf, 1991] Ishida, T., and Korf, R. 1991. Moving target search. In *Proceedings of IJCAI*, 204–210.

[Kitano *et al.*, 1999] Kitano, H.; Tadokoro, S.; Noda, I.; Matsubara, H.; Takahashi, T.; Shinjou, A.; and Shimada, S. 1999. Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *IEEE Conf. on Man, Systems, and Cybernetics*.

[Koenig & Likhachev, 2002] Koenig, S., and Likhachev, M. 2002. D* Lite. In *Proceedings of the National Conference on Artificial Intelligence*, 476–483.

[Koenig, Tovey, & Y., 2003] Koenig, S.; Tovey, C.; and Y., S. 2003. Performance bounds for planning in unknown terrain. *Artificial Intelligence* 147:253–279.

[Koenig, 2004] Koenig, S. 2004. A comparison of fast search methods for real-time situated agents. In *Proceedings of the 3rd Int. Joint Conf. on Autonomous Agents and Multiagent Systems - vol. 2*, 864 – 871.

[Korf, 1990] Korf, R. 1990. Real-time heuristic search. *AIJ* 42(2-3):189–211.

[Korf, 1993] Korf, R. 1993. Linear-space best-first search. *AIJ* 62:41–78.

[Sasaki, Chimura, & Tokoro, 1995] Sasaki, T.; Chimura, F.; and Tokoro, M. 1995. The Trailblazer search with a hierarchical abstract map. In *Proceedings of IJCAI*, 259–265.

[Shimbo & Ishida, 2003] Shimbo, M., and Ishida, T. 2003. Controlling the learning process of real-time heuristic search. *AIJ* 146(1):1–41.

[Shue & Zamani, 1993] Shue, L.-Y., and Zamani, R. 1993. An admissible heuristic search algorithm. In *Proceedings of the 7th Int. Symp. on Methodologies for Intel. Systems (ISMIS-93)*, volume 689 of *LNAI*, 69–75.

[Shue, Li, & Zamani, 2001] Shue, L.-Y.; Li, S.-T.; and Zamani, R. 2001. An intelligent heuristic algorithm for project scheduling problems. In *Proceedings of the 32nd Annual Meeting of the Decision Sciences Institute*.

[Stenz, 1995] Stenz, A. 1995. The focussed D* algorithm for real-time replanning. In *Proceed. of the Int. Conf. on Artificial Intel.*, 1652–1659.

# Generating Temporally Contingent Plans

**Janae N. Foss** **Nilufer Onder**
Department of Computer Science
Michigan Technological University
1400 Townsend Drive
Houghton, MI 49931
{jnfoss,nilufer}@mtu.edu

## Abstract

Uncertainty applies to many aspects of planning problems. Much research has been done to deal with problems where actions have uncertain effects. In reality, many planning problems also involve actions with uncertain durations, but this type of uncertainty has not been widely studied in planning until the recent development of several planners which incorporate durational uncertainty. Additionally, theoretical work has been done on characterizing the level of controllability in plans involving actions with uncertain durations. We have developed an approach for finding temporally contingent plans, i.e., plans with branches that are based on the duration of an action at execution time. More specifically, the problems we are studying satisfy the following criteria: (1) there is more than one solution plan, (2) solution plans are ranked by a metric that is not fully based on makespan, (3) actions have uncertain durations, (4) the start and/or end times of some actions are constrained, and (5) as actions require more time to complete, plans that are judged highly by the metric become invalid. We describe our approach for determining the time points that cause an unsafe situation and for inserting temporal contingency branches. Experimental results with both sequential and parallel plans are discussed.

## 1 Introduction

Uncertainty applies to many aspects of planning problems. Much research has been done to deal with problems where actions have uncertain effects. A classification of planners that can handle this kind of uncertainty is given in [Dearden *et al.*, 2003]. In reality, many planning problems also involve actions with uncertain durations, but this type of uncertainty has not been widely studied in planning until the recent development of several planners which incorporate durational uncertainty [Younes and Simmons, 2004; Mausam and Weld, 2005; Little *et al.*, 2005]. Additionally, there has been theoretical work on characterizing the level of controllability in plans involving actions with uncertain durations [Vidal and Fargier, 1999]. Conservative planning (i.e., finding plans that are likely to be safe regardless of action duration) is one way to handle durational uncertainty. The advantage to this approach is that the resulting plans are robust. However, conservative planning often results in missed opportunities. To take advantage of available opportunities while still having a robust plan, we have developed an approach for finding temporally contingent plans (TCPs), i.e., plans with branches that are based on the duration of an action at execution time. Specifically, the problems we are studying satisfy the following criteria: (1) there is more than one solution plan, (2) solution plans are ranked by a metric that is not fully based on makespan[1], (3) actions have uncertain durations, (4) the start and/or end times of some actions are constrained, and (5) as actions require more time to complete, plans that are judged highly by the metric become invalid. We take an optimistic approach by first finding a plan that is valid when all actions complete quickly. We then use the methods described in [Dechter *et al.*, 1991] to determine when the plan may fail. At time points that cause an unsafe situation, temporal contingency branches are inserted.

As an example, consider the problem of traveling from home to a conference. One solution plan is to drive to the airport, fly to the destination city, take a shuttle to the conference venue, and finally register for the conference. Another solution plan could involve taking a taxi instead of a shuttle to the venue. Assuming the metric is to minimize money spent, the plan with the shuttle action would be preferred. However, the taxi may be faster than the shuttle and since there are constraints on the time one can register for the conference, depending on how long the flight takes, there may only be enough time for the more expensive taxi option. To always have a safe plan, and be able to save money when possible, our approach would generate a TCP to drive to the airport, fly to the destination, take the shuttle if there is enough time, otherwise take the taxi, and register for the conference. In this way, our approach ensures enough time for the worst case while making use of better options when time allows. Throughout this paper our running example will be the conference domain.

Our contributions are threefold: (1) we introduce the no-

[1] As we define it, a metric may either combine makespan with some nontemporal measure or simply be stated as a nontemporal measure.

tion of TCPs, (2) we provide a greedy iterative algorithm that inserts branches based on time rather than world conditions, (3) we show that viable plans can be generated in this framework. In the remainder of this paper we first define temporal uncertainty and explain our algorithm for creating TCPs. We then give a theoretical framework for characterizing solution plans. This is followed by preliminary experiments, general remarks, and a description of future work.

## 2 Planning with Temporal Uncertainty

Temporal planning and reasoning activities involve actions that have a temporal extent, such as an action duration, and general temporal constraints, such as a deadline for when an action must begin. Problems with temporal aspects may have actions with uncertainty present in one of the following three ways. First, the temporal aspects are certain while the effects on the world are uncertain. For example, eating a meal may take 30 minutes, but it is uncertain if hunger will be satisfied. Second, the changes in the world are certain, but the temporal aspects are uncertain. For example, hunger will be satisfied after eating a meal, but the duration of the meal is uncertain. Third, both the temporal aspects and changes in the world are uncertain. For example, a meal will be eaten but it is uncertain how long it will take and whether hunger will be satisfied. For simplicity, our current work is concerned with only the first type of uncertainty. We plan to deal the other two types of uncertainty in future work.

In our framework, uncertainty in action duration is represented with the interval [*min-d*, *max-d*], where *min-d* and *max-d* are minimum and maximum reasonable durations[2], respectively (*min-d* $> 0$ and *max-d* $< \infty$). We have extended PDDL2.2 [Edelkamp and Hoffman, 2004] to represent *interval durative* actions as opposed to single point durative actions. In temporal reasoning literature when an interval duration is assigned to an action, it is generally assumed that the user can select any value from the interval. In our work we build on the model described in [Vidal and Fargier, 1999] and assume that some action durations will be known only at execution time and thus are uncertain. Hence we define two types of interval durative actions. If the duration is *assignable*, the executing agent (user) can choose a duration between *min-d* and *max-d*. If the duration is *uncertain* the action will consume some time between *min-d* and *max-d*, but the exact duration is beyond the control of the agent. In the conference domain, the duration of a flight is uncertain, but the duration of eating a meal is assignable.

In Figure 1 our coding of the conference domain is given. Note that the `eat-meal` action has an assignable duration but the other actions have unassignable (i.e., uncertain) durations. As defined, `fly_airport2_airport1` has a duration between 45 and 90 time units, starting at time 30 and conference registration has a duration between 5 and 10 time units, starting between times 84 and 141, exclusive. These two actions show the syntax we have added for associating actions with their execution time constraints. In our framework we assume that there is no penalty involved with waiting to begin execution of an action. The addition of assignable and unassignable interval durations is a conceptual extension to PDDL2.2, whereas the `execution-time` syntax provides convenience in coding but can be represented indirectly by the timed initial literals of PDDL2.2 which are used to define temporal windows.

Domain description

```
(define (domain conference-travel)
  (:requirements :fluents :equality
   :interval-durative-actions :execution-times)
  (:predicates (at_airport1) (at_airport2) (at_hotel)
               (not_hungry) (attending_conference))
  (:functions   (money_spent))

  (:interval-durative-action fly_airport2_airport1
   :unassignable-interval-duration
        (and (min ?duration 45) (max ?duration 90))
   :condition (at start (at_airport1))
   :effect (and (at end (at_airport2))
    (at start (not (at_airport1)))
    (at start (increase (money_spent) 200)))
   :execution-time (start at 30))

  (:interval-durative-action taxi_hotel_airport2
   :unassignable-interval-duration
       (and (min ?duration 15) (max ?duration 20))
   :condition (at start (at_airport2))
   :effect (and (at end (at_hotel))
    (at start (not (at_airport2)))
    (at start (increase (money_spent) 120))))

  (:interval-durative-action shuttle_hotel_airport2
   :unassignable-interval-duration
        (and (min ?duration 30) (max ?duration 60))
   :condition (at start (at_airport2))
   :effect (and (at end (at_hotel))
    (at start (not (at_airport2)))
    (at start (increase (money_spent) 20))))

  (:interval-durative-action eat_meal
   :assignable-interval-duration
          (and (min ?duration 20) (max ?duration 60))
   :condition (at start (attending_conference))
   :effect (at end (not_hungry))
    (at start (increase (money_spent) 20))))

 (:interval-durative-action register_for_conference
  :unassignable-interval-duration
          (and (min ?duration 5) (max ?duration 10))
  :condition (over all (at_hotel))
  :effect (at end (attending_conference))
  :execution-time (and (start after 84) (start before 141))))
```

Problem description

```
(define (problem conference-travel-1)
  (:domain conference-travel)
  (:init        (at_airport1)
                (= (money-spent) 0))
  (:goal        (attending_conference))
  (:metric      minimize (money-spent)))
```

Figure 1: Conference travel domain and problem.

## 3 Creating Temporal Contingency Plans

When creating a TCP it is important to find a plan that is both *safe* and ranked highly by the plan metric. Intuitively, a plan

[2] These durations may be determined using a probabilistic distribution. The 95th percentile has been used to produce conservative plans (e.g. [Fox and Long, 2002]) and may be a good selection for *max-d*. Experimental work must be done to determine a reasonable percentile for *min-d*.

is safe if its validity is guaranteed, even when all of its uncertain actions require their maximum duration to complete. One approach to building safe plans in this context is to assume that all actions always require their maximum duration. Though the result is a robust plan, in our framework this pessimistic assumption leads to a plan that is not ranked well by the metric. We take an optimistic approach and assume that actions require only their minimum duration. Since this assumption may yield an unsafe plan, we build temporally contingent branches into it using a general Just-In-Case style algorithm [Drummond *et al.*, 1994] where we generate a seed plan, find points where it is likely to fail, and then insert contingency branches at those points. Our algorithm is given in Figure 2. To generate the seed plan, we make the optimistic assumption, i.e., for each action assign *min-d* as the duration. This yields a plan that is ranked highly by the metric (step 1.1). Next, we analyze the seed plan to find out, temporally speaking, when it becomes unsafe (steps 1.2 and 1.3). At any time point where the seed plan becomes unsafe, we generate and insert a branch that is safe. This technique creates a plan that includes a path that can be safely executed when all of its actions require their maximum duration, but also includes branches yielding a more desirable result that are executed when actions require less time.

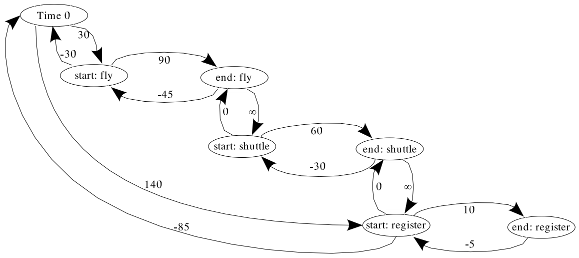
```
TCP Algorithm
 (1.1)  Generate a seed plan, $P$ with $n$ actions, assuming all actions require only
        their minimum duration
 (1.2)  Construct the distance graph $D$ of $P$
 (1.3)  $TCP \leftarrow$ MakeSafePlan($P$, $D$)

MakeSafePlan(Plan $P$, DistanceGraph $D$)
 (2.1)  Create $TCP$ with $P$ as main branch
 (2.2)  For each action $i = n$ to 1 in $P$
          (2.3)  $maxAllowedDuration \leftarrow$ shortestPathDistance($s_i$, $e_i$, $D$)
          (2.4)  While $maxAllowedDuration <$ maxDuration($i$)
                   (2.5)  $newMinDuration \leftarrow maxAllowedDuration + 1$
                   (2.6)  $TCB \leftarrow$ GetContingencyBranch($i$, $newMinDuration$)
                   (2.7)  Insert $TCB$ into $TCP$ to be executed when $i$ requires more time
                          than $newMinDuration$
                   (2.8)  $maxAllowedDuration \leftarrow$ shortestPathDistance($s_i$, $e_i$, $DB$)
          (2.9)  Modify $D$ so that $i$ is constrained to start at the latest time $i$ can
                 safely start
(2.10)  return $TCP$

GetContingencyBranch(Action $i$, Duration $newMinDuration$)
 (3.1)  Modify domain to assume $i$ requires $newMinDuration$
 (3.2)  Modify initial conditions of problem to reflect state of world at start of $i$
 (3.3)  $earliestStartTime \leftarrow -1 \times$ shortestPathDistance($s_i$, $s_0$, $D$)
 (3.4)  Modify problem to constrain $i$ to start at $earliestStartTime$
 (3.5)  Using the modified domain and problem, generate a branch $B$
 (3.6)  Construct the distance graph $DB$ of $B$
 (3.7)  $TCB \leftarrow$ MakeSafePlan($B$, $DB$)
 (3.8)  return $TCB$, $DB$
```

Figure 2: Algorithm for creating TCPs.

As stated above, when generating the seed plan (and subsequent branches), the duration of each action is set at *min-d*, removing all uncertainty at planning time. (For the remainder of this section, the term *plan* is used to refer to either a seed plan or branch plan.) This allows expression of the problem and domain in PDDL2.2 and plans can be generated with any planner that understands PDDL2.2. A plan $P$ returned by such a planner will be temporally deterministic. Our algorithm factors in temporal uncertainty by converting $P$ to a directed, edge-weighted graph called a *distance graph*, thus expressing $P$ as a simple temporal network (STN) [Dechter *et al.*, 1991]. STNs are widely used in temporal reasoning and include nodes representing time points and edges between pairs of nodes representing temporal constraints between time points. Figure 3 shows (a) the seed plan that would be generated for the problem in Figure 1 and (b) the corresponding distance graph. An in-depth description of how to perform step 1.2 of the algorithm is given next.

| Execution Time | Action |
|---|---|
| 30 | fly_airport2_airport1 |
| 76 | shuttle_hotel_airport2 |
| 107 | register_for_conference |

(a)



(b)

Figure 3: (a) A seed plan for the problem in Figure 1. Note that the times given by the seed plan assume actions require their minimum durations. (b) The distance graph for the seed plan in *(a)*, incorporating temporal uncertainty. For clarity, only the most important edges are shown.

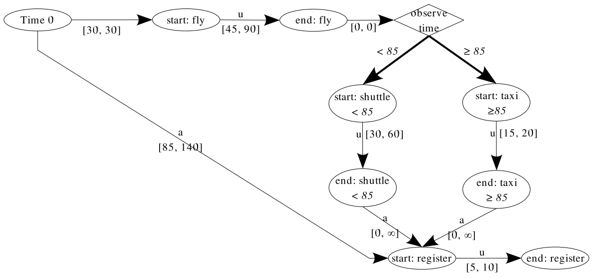
Since the execution times in $P$ are based on a deterministic temporal assumption, they are ignored during the construction of the distance graph $D$. Only the actions of $P$ are considered. In construction of $D$, each action is dealt with individually to allow any possible concurrency in $P$ to be present in $D$. The first step in constructing $D$ is to add two nodes for each action $i$, one for its start time $s_i$ and one for its end time $e_i$, and a node $s_0$ representing time 0. Edges are then added in pairs representing temporal relations and weighted with temporal distances. For each action $i$, a pair of edges is added between $s_i$ and $e_i$. The edge $s_i \rightarrow e_i$ is weighted with *max-d* of $i$ and the edge $e_i \rightarrow s_i$ is weighted with -1 × (*min-d* of $i$). Next, pairs of edges are added between $s_0$ and each $s_i$ node. (Pairs of edges can also be added from $s_0$ to each $e_i$, but are not necessary and add no new temporal information.) If an action $i$ has a constrained start time, the edge $s_0 \rightarrow s_i$ is weighted with the *latest start time* of $i$ and the edge $s_i \rightarrow s_0$ is weighted with -1 × (*earliest start time* of $i$). This is shown with the fly and register actions in Figure 3(b). When an action does not have a constrained start time, the edge $s_0 \rightarrow s_i$ is weighted with ∞ and the edge $s_i \rightarrow s_0$ is weighted with 0, signifying that the start of action $i$ comes after time 0, but there are no other constraints. For clarity, these edges are not included in Figure 3(b).

The final step in constructing the distance graph is to insert edges that represent relationships between actions. Though *P* contains a sequence of steps, some concurrency may be possible. To properly discover and encode this in *D*, causal links and threats in *P* must be identified. This is done using an algorithm similar to the one described by [R-Moreno *et al.*, 2002]. For every condition *c* of each action *i*, a causal link is added to the closest action *j* in the plan that appears before *i* and produces *c* as an effect. The causal link forces the producer action to occur before the consumer. A threat link is added between an action *i* and an action *j* when an effect of *j* negates a condition of *i*.[3] This algorithm discovers no knowledge about temporal distance, so pairs of edges labeled with 0 and ∞ are added to the graph simply expressing that one action must occur before the other. There is no concurrency in the plan of Figure 3, so edges are added from the start of each action to the end of the previous action, representing causal links. There are no threats in this example.

Since *D* contains all temporal constraints given in the domain, it can be used to determine when *P* becomes unsafe. This procedure is given by the *MakeSafePlan* function of Figure 2. In [Dechter *et al.*, 1991] it is proved that the absolute bounds on the temporal distance between any two time points represented by nodes *a* and *b* (assuming $a \prec b$) in *D*, is given by the interval [-1 × *(weight of shortest path from b to a), weight of shortest path from a to b*]. The shortest path can be found using an algorithm such as the *Bellman-Ford* single source shortest path algorithm with a runtime of $O(|V||E|)$. In Figure 3(b) we see that the duration of the fly action is expressed by the interval [45, 90]. However, using the shortest path method (step 2.3 in Figure 2), it is found that the absolute bounds on the duration of the fly action are expressed by the interval [45, 80]. This indicates that if the fly action takes more than 80 time units, the rest of the plan becomes unsafe. To have a safe solution, a contingency must be generated that can reach the goal safely when the fly action takes more than 80 time units, so the loop in step 2.4 will be entered. Currently, the loop considers actions in reverse order. We plan to experiment with different orderings in the future. This loop allows multiple contingency branches to be generated for the same time point. The *GetContingencyBranch* function modifies the domain and problem to reflect the state of the world when the branch occurs and then generates a branch plan which is verified in the same way as the seed plan. Hence, branches may themselves contain branches. After a safe contingency branch has been generated, it is inserted into the seed plan (step 2.7). For the example problem, the contingency branch that will be generated is `taxi_hotel_airport2`, `register_for_conference`. After leaving the loop of 2.4, it is known that action *i* can safely execute, even if it requires its maximum duration. In step 2.9, *D* is modified to ensure that actions occurring before *i* complete early enough so that *i* has enough time to consume its maximum duration if necessary. Once all actions have been verified, the TCP is safe. The TCP of the example problem is given in Figure 4(a). In the next section, we formally explain a data structure that can be used to represent TCPs.

```
At time 30: fly_airport2_airport1
IF (time < 85)
   Before time 85: shuttle_hotel_airport2
   Before time 140: register_for_conference
ELSE
   Before time 120: taxi_hotel_airport2
   Before time 140: register_for_conference
```

(a)



(b)

Figure 4: (a) A TCP for the problem in Figure 1. (b) The TCN for the plan in *(a)*.

## 4 Temporal Contingency Networks

We represent TCPs using a new data structure called a *Temporal Contingency Network* (TCN). TCNs are an extension of STNs and are inspired by the STPU model defined in [Vidal and Fargier, 1999]. TCNs extend STNs in two dimensions. First, interval durations are labeled as user assignable or not; second, some nodes represent decisions based on observations of time. The second aspect enables the representation of TCPs. Part (b) of Figure 4 depicts a TCN for the TCP in part (a).

Formally, a TCN is a quadruple <**T, O, E, B**>. **T** is a set of nodes representing start and end times of actions. A node representing the absolute start time is also included in **T**. Each node in **T** is referred to as a *time point*. Nodes in **T** that are not included in all paths of execution contain a context label [Peot and Smith, 1992] identifying the branch of execution they belong to. Oval nodes in the figure belong to **T**. The shuttle and taxi nodes contain context labels because these actions do not belong to all paths of execution. **O** is a (possibly empty) set of observation nodes representing decisions about which subsequent actions to execute. Observations of time are assumed to be executable at any time (no preconditions) and instantaneous; and should be executed immediately after the preceding time point. A TCN with observation nodes is safe if all the possible paths are safe. In the figure, the diamond represents an observation node. **E** is a set of interval labeled edges representing constraints between time points. Edges in **E** can be marked as uncertain, assignable, or unmarked. The non-bold

[3] In the future, we plan to extend this algorithm to discover threats that may be caused by actions consuming the same resource. Currently, actions of this sort are disallowed.

edges in the figure belong to **E**. Edges representing assignable durations are marked with *a* and those representing uncertain durations are marked with *u*. Edges with intervals representing an exact amount of time (such as *Time 0* → *start: fly*) are unmarked. **B** is a set of temporally labeled edges leaving observation nodes. The bold edges in the figure belong to *B*. As shown, these edges are given a label indicating when each branch can safely be taken. This data structure provides a rich context for reasoning about TCPs.

## 5 Experiments and Discussion

In this section we provide preliminary experimental results. To the best of our knowledge, there are no planners that prepare contingency branches based on time. We therefore designed our experiments to show that our algorithm works and to help identify the ways in which it can be improved. LPG-*TD* [Gerevini *et al.*, 2004] was the planner that we used for generating seed plans and branches. We choose LPG-*TD* because it can handle the timed initial literals of PDDL2.2 and can optimize for temporal and nontemporal metrics. All experiments were performed on a machine with a 3.0GHz Pentium 4 CPU and 1GB of RAM.

The domain used in the experiments is another version of the domain in Figure 1. The experimental domain has no meal action, but three new actions are added. First, there is a drive action that must occur in order to arrive at *airport1*. Next, actions are added to include the possibility of taking a flight from *airport1* to a new airport, *airport3*, and from *airport3* to *airport2*. The direct flight is more costly than flying through *airport3*. Finally, an action is added to the domain for taking public transportation as the least costly way to get to the hotel from *airport2*. The domain was created in this way to allow many different possible solutions when registering for the conference is the only goal.

The first set of experiments involved plans with no possible parallelism. These experiments were done to compare the runtime as more conditional branches were added to the seed plan. The domain was modified for each run to force different numbers and combinations of branches. The same seed plan was generated every time. Table 1 shows the results of these experiments. In the first run, the seed plan was always safe and thus no branches were created. In successive runs, the number of created branches ranged from 1 to 4. There were 2 different runs that each produced 3 branches. In general, as more branches were created, the runtime increased. However, one of the runs with 3 branches took much longer than the other, and even longer than the run with 4 branches. There are two reasons for this. First, as is clear in the table, generation of the seed plan and branches by LPG-*TD* accounts for nearly all of the runtime. Due to the conditions in this experiment, LPG-*TD* required extra time to create one of the branches. Second, this run contained longer branches than the other run with 3 branches, thus requiring more time for verification time by our algorithm.

| branches created | LPG-*TD* runtime | total runtime |
|---|---|---|
| 0 | 260 | 260.1 |
| 1 | 520 | 527.0 |
| 2 | 780 | 788.5 |
| 3 | 1040 | 1054.0 |
| 3 | 1290 | 1302.6 |
| 4 | 1130 | 1144.9 |

Table 1: Run times in milliseconds of experiments with problems requiring different numbers of conditional branches.

The second set of experiments involved parallel plans. In our algorithm, no dependencies are assumed between actions in the plan. As described previously, dependencies are discovered when the distance graph is built. In this way, our algorithm inherently allows parallel plans. To test this aspect, three new actions were added to the domain. The first was an action for grading exams that had to be completed before arriving at the hotel. The other two actions were for reading papers (one for a long paper and one for a short paper) which had to be completed after grading the exams and before arriving at the hotel. More knowledge was gained by reading the long paper. The problem was then modified by adding two goals to read some paper (either or both) and grade the exams. The metric was modified to rank plans higher when more knowledge was gained while still trying to minimize the money spent. The sequential experiments were re-run with the new modified domain and problem. The addition of the parallel actions caused no significant change in runtime among these tests. But, when the domain was modified so that a branch had to be inserted to read the short paper when the exams took a long time to grade, there was a spike in runtime. The reason for this is that our algorithm does not currently identify parallel paths of execution and treat them separately. In the plan that LPG-*TD* created, grading the exams was the first action, reading the paper was the second action, and the rest of the plan followed. In creating a branch to insert after the grading action, our algorithm had to rediscover the entire rest of the plan, though the grading action only directly affected reading the paper. Our algorithm produces a valid result, but it is inefficient. We plan to research this topic in the future.

## 6 Related work

The main framework of our algorithm is very close to Just-In-Case (JIC) scheduling [Drummond *et al.*, 1994]. The JIC scheduler analyzes a seed schedule, finds possible failure points, and inserts contingency branches so that valuable equipment time is not lost when an experiment fails. Our work extends this framework to multiple planner goals, parallel plans, and nontemporal metrics, but does not consider probability of failure. Presently we insert contingent branches for every action that might not have sufficient time. We plan to improve our algorithm by systematically evaluating and selecting branch insertions points as in [Onder and Pollack, 1999; Dearden *et al.*, 2003]. Dearden et al's [2003] approach involves generating a seed plan and adding contingent plans based on a rich utility metric involving goal values and continuous resources.

There are a number of domain independent planners that can handle durative actions. We used LPG-*TD* because it can optimize based on a nontemporal metric. Other planners include TGP [Smith and Weld, 1999], a planner that uses mutual exclusion reasoning in a temporal context, SAPA [Do and Kambhampati, 2002], a heuristic forward chaining planner; HSP [Haslum and Geffner, 2002] a heuristic planner with time and resources; and CPT an optimal temporal POCL planner based on constraint programming [Vidal and Geffner, 2004].

Vidal and Fargier [1999] present an analysis of three levels of controllability given a plan. The plans analyzed have actions with uncertain durations and temporal constraints but the start times of actions and the durations of assignable intervals have not been assigned yet. A *control sequence* is an assignment of times to time points and durations to assignable intervals such that all the constraints are respected regardless of the actual durations of the uncertain intervals. A control sequence is *strong* if it can be determined prior to execution. A control sequence is *weak* if (parts of) it can only be determined during execution right after some actual action durations have been observed. Finally, a control sequence is *dynamic* if the completion of an action is too late to make an assignment of start times to the remainder of the actions but there exists a time point *t* such that if an actual duration is learned in advance at time *t* safe assignments can be made. In our work we are concerned with generating plans with strong control sequences.

Tsamardinos et al. describe an algorithm for merging existing plans with assignable durations and nontemporal conditional branches [2000]. We plan to extend our algorithm by using their plan merging framework. In particular, we will be generating two plans, one with the minimum duration and one with the maximum duration for each uncertain interval and merging those two plans into a conditional plan such that common actions are executed unconditionally and actions that differ are executed under appropriate contexts.

Tempastic [Younes and Simmons, 2004] is a planner that models continuous time, probabilistic effects, probabilistic exogenous events and both achievement and maintenance goals. It uses a “generate-test-debug” algorithm that generates an initial policy and fixes the policy after analyzing the failure paths. In producing a better plan, the objective is to decrease the probability of failure. Nontemporal resources are not modeled.

Mausam and Weld [2005] describe a planner that can handle actions that are concurrent, durative and probabilistic. They use novel heuristics with sampled real-time dynamic programming in this framework to generate policies that are highly optimal. The quality metric includes makespan but nontemporal resources are not modeled in the planning problem.

Prottle [Little *et al.*, 2005] is a planner that allows concurrent actions that have probabilistic effects and probabilistic effect times. Prottle uses effective planning graph based heuristics to search a probabilistic AND/OR graph consisting of advancement and placement nodes. Prottle’s plan metric includes probability of failure but not makespan. Prottle does not model metric resources.

Finally, we would like to mention work from the field of microarchitecture where the objective is to analyze a bottleneck situation which consists of parallel events and determine which events are worthwhile to optimize so that the total makespan of the bottleneck decreases [Fields *et al.*, 2003]. In our future work we will employ techniques from this work to identify portions of a plan that can be optimized rather than to prepare contingent plans as a response to suboptimal execution times.

## 7 Conclusions and Future Work

We have presented a framework for characterizing and directly dealing with temporal uncertainty. We define temporal uncertainty by assigning actions an interval duration, rather than a single point duration. Our approach is to make an optimistic assumption that all actions complete as quickly as possible. We then generate a plan with inexpensive actions that may become invalid at some point when the optimistic assumption proves wrong. We create more costly contingency plans to be executed only when actions in the inexpensive plan run long enough that an unsafe situation occurs.

Our algorithm is greedy and thus it can return locally optimal solutions which are not globally optimal. In the future, we plan to develop several heuristics to help avoid this problem. One idea is to re-generate the entire plan when a temporally unsafe situation is found. This new plan could be compared to the current plan to determine whether a contingency branch should be added to the current plan or if the new plan should replace the seed plan. In addition to avoiding locally optimal solutions, this approach would generate an appropriate solution in the case that no valid contingency branch existed. The main drawback to this approach is that regenerating the entire plan can be very time consuming. Another way to avoid locally optimal solutions would be to take an MDP based approach where every state in the world contains a time stamp. The naive approach of creating a state representing every possible time point would not be efficient, but improvements could be gained by using states representing intervals of time.

As we continue this work, we plan to extend it in several ways. Our algorithm improves on a strictly conservative approach, but the safe TCPs that it generates may still include missed opportunities. We plan to develop algorithms that can find idle time in a TCP and then insert opportunities as defined by [Fox and Long, 2002]. We would also like to extend our work to be able to handle actions with uncertain effects, including uncertain consumption of nontemporal resources. Finally, we will develop a test bed of problems involving not only our conference domain, but other domains that may benefit from our approach, such as the Rover and Satellite domains.

## Acknowledgements

We thank anonymous reviewers for their helpful comments and discussion. Janae N. Foss’ research was supported by the Harriett G. Jenkins Predoctoral Fellowship Program and a grant from the Michigan Council of Women in Technology.

## References

[Dearden *et al.*, 2003] Richard Dearden, Nicolas Meuleau, Sailesh Ramakrishnan, David Smith, and Richard Washington. Incremental contingency planning. In *ICAPS-03 Workshop on Planning Under Uncertainty*, June 2003.

[Dechter *et al.*, 1991] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[Do and Kambhampati, 2002] Minh B. Do and Subbarao Kambhampati. Planning graph-based heuristics for cost-sensitive temporal planning. In *Proc. 6th Int. Conf. on AI Planning & Scheduling*, 2002.

[Drummond *et al.*, 1994] M. Drummond, J. Bresina, and K. Swanson. Just-incase scheduling. In *Proc. 12th National Conf. on Artificial Intelligence*, pages 1098–1104, 1994.

[Edelkamp and Hoffman, 2004] Stefan Edelkamp and Jörg Hoffman. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Computer Science Department, University of Freiburg, January 2004.

[Fields *et al.*, 2003] Brian A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. Using interaction costs for microarchitecture bottleneck analysis. In *Proc. 36th international Symposium on Microarchitecture (MICRO-36'03)*, 2003.

[Fox and Long, 2002] Maria Fox and Derek Long. Single-trajectory opportunistic planning under uncertainty. In *Proceedings of the 21st Workshop of the UK Planning and Scheduling Special Interest Group*, November 2002.

[Gerevini *et al.*, 2004] Alfonso Gerevini, Alessandro Saetti, Ivan Serina, and Paolo Toninelli. Planning in PDDL2.2 domains with LPG-*TD*. In *International Planning Competition booklet (ICAPS-04)*, 2004.

[Haslum and Geffner, 2002] Patrik Haslum and Hector Geffner. Heuristic planning with time and resources. In *Proc. 6th European Conf. on Planning*, 2002.

[Little *et al.*, 2005] Iain Little, Douglas Aberdeen, and Sylvie Thiebaux. Prottle: A probabilistic temporal planner. In *Proc. 20th National Conf. on Artificial Intelligence (AAAI-05)*, 2005.

[Mausam and Weld, 2005] Mausam and Daniel S. Weld. Concurrent probabilistic temporal planning. In *Proc. 15th International Conf. on Automated Planning and Scheduling (ICAPS-05)*, 2005.

[Onder and Pollack, 1999] Nilufer Onder and Martha E. Pollack. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *Proc. 16th National Conf. on Artificial Intelligence*, pages 577–584, 1999.

[Peot and Smith, 1992] Mark A. Peot and David E. Smith. Conditional nonlinear planning. In *Proc. 1st International Conf. on Artificial Intelligence Planning Systems*, pages 189–197, 1992.

[R-Moreno *et al.*, 2002] M Dolores R-Moreno, Angelo Oddi, Daniel Borrajo, Amedeo Cesta, and Daniel Meziat. Integrating hybrid reasoners for planning and scheduling. In *The Twenty-First Workshop of the UK Planning and Scheduling Special Interest Group*, pages 179–189, 2002.

[Smith and Weld, 1999] David E. Smith and Daniel Weld. Temporal planning with mutual exclusion reasoning. In *Proc. 16th International Joint Conf. on Artificial Intelligence*, 1999.

[Tsamardinos *et al.*, 2000] Ioannis Tsamardinos, Martha E. Pollack, and John F. Horty. Merging plans with quantitative temporal constraints, temporally extended actions, and conditional branches. In *Proc. 5th International Conf. on Artificial Intelligence Planning and Scheduling*, pages 264–272, 2000.

[Vidal and Fargier, 1999] Thierry Vidal and Helene Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 11(1):23–45, 1 1999.

[Vidal and Geffner, 2004] Vincent Vidal and Hector Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. In *Proc. 19th National Conf. on Artificial Intelligence*, pages 570–577, 2004.

[Younes and Simmons, 2004] Hakan L.S. Younes and Reid G. Simmons. Solving generalized semi-markov decision processes using continuous phase-type distributions. In *Proc. 19th National Conf. on Artificial Intelligence (AAAI-04)*, 2004.

# Improving Convergence of LRTA*$(k)$ [*]

**Carlos Hernández** and **Pedro Meseguer**
Institut d'Investigació en Intel.ligència Artificial, CSIC
Campus UAB, 08193 Bellaterra, Spain
{chernan,pedro}@iiia.csic.es

## Abstract

LRTA* is a real-time heuristic search algorithm widely used. In each iteration it updates the heuristic estimate of the current state. In this paper, we present three versions of LRTA*$(k)$, a new LRTA*-based algorithm that is able to update the heuristic estimates of up to $k$ states, not necessarily distinct. Based on bounded propagation, this updating strategy maintains heuristic admissibility, so LRTA*$(k)$ keeps the good theoretical properties of LRTA*. The new algorithm produces better solutions in the first trial and converges faster when compared with other state-of-the-art algorithms on classical benchmarks for real-time search. We provide experimental evidence of the improvement in performance of these versions, at the extra cost of longer planning steps.

## 1 Introduction

LRTA* [Korf, 1990] is a real-time heuristic search algorithm that interleaves planning and action execution in an on-line manner. It improves its performance over successive trials on the same problem instance, by recording better heuristic estimates. This algorithm works on a search space where every state $x$ has a heuristic estimate $h(x)$ of the cost from $x$ to a goal. It is complete under a set of reasonable assumptions. If $h(x)$ is admissible, after a number of trials $h(x)$ converges to their exact values along every optimal path.

In this paper we present three versions of LRTA*$(k)$ [Hernandez and Meseguer, 2005], an algorithm based on LRTA* that updates the heuristic estimate of up to $k$ –not necessarily distinct– states per iteration, following a bounded propagation strategy. This updating maintains heuristic admissibility, so LRTA*$(k)$ converges to exact heuristic values in optimal paths. In fact, LRTA* is a particular case of LRTA*$(k)$ with $k = 1$. Bounded propagation causes significant benefits in the first solution, convergence and solution stability, at the cost of extra computation per planning step. In our experiments on classical real-time benchmarks, with small to medium $k$ the time per planning step has remained reasonable when compared with LRTA*.

The three versions presented are denoted with the subindexes 0, 1 and 2. They differ in the extra condition that a state, selected by bounded propagation, has to satisfy to be considered for updating. In LRTA$^*_0(k)$, a state involved in the propagation process is considered for updating only if it has already been expanded in the current trial. In LRTA$^*_1(k)$, this condition is relaxed to have been expanded in the the current or previous trials. In LRTA$^*_2(k)$, no extra condition is imposed, so any state involved in the propagation is considered for updating. Relaxing the condition for state updating causes to improve convergence, at the extra cost of increasing the duration of the average planning step.

This paper is organized as follows. In Section 2, we summarize the main search concepts used for the LRTA* algorithm. In Section 3, we describe the new algorithm LRTA*$(k)$, and present its three versions. We provide experimental results in Section 4 on four classical real-time benchmarks. Comparison with existing work appears in Section 5. Finally, Section 6 contains some conclusions.

## 2 LRTA*

The state space is defined as $(X, A, c, s, G)$, where $(X, A)$ is a finite graph, $c : A \mapsto [0, \infty)$ is a cost function that associates each arc with a finite cost, $s$ is the start state, and $G \subset X$ is the set of goal states. $X$ is a finite set of states, and $A \subset X \times X - \{(x, x)|x \in X\}$ is a finite set of arcs. Each arc $(v, w)$ represents an action whose execution causes the agent to move from $v$ to $w$. The state space is undirected: for any action $(x, y) \in A$ there exists its inverse $(y, x) \in A$ with the same cost $c(x, y) = c(y, x)$. The successors of a state $x$ are $Succ(x) = \{y|(x, y) \in A\}$. A path $(x_0, x_1, x_2, \ldots)$ is a sequence of states such that every pair $(x_i, x_{i+1}) \in A$. The cost of a path is the sum of costs of the actions in that path. A heuristic function $h : X \mapsto [0, \infty)$ associates with each state $x$ an approximation $h(x)$ of the cost of a path from $x$ to a goal. The exact cost $h^*(x)$ is the minimum cost to go from $x$ to a goal. $h$ is admissible iff $\forall x \in X$, $h(x) \leq h^*(x)$. A path $(x_0, x_1, \ldots, x_n)$ with $h(x_i) = h^*(x_i)$, $0 \leq i \leq n$ is *optimal*.

The LRTA* algorithm works as follows. From the current state $x$, it performs lookahead at depth $d$, and updates $h(x)$ to the max $\{h(x)$, min $[k(x, v) + h(v)]\}$, where $v$ is a frontier state and $k(x, v)$ is the cost of going from $x$ to $v$.

[*] Partially supported by the Spanish REPLI project TIN2002-04470-C03-03.

Then, it moves to a state $y$, successor of $x$, with minimum $c(x, y) + h(y)$. This state become the current state and the process iterates, until finding a goal state. This process is called a trial. If the final heuristic estimates of a trial are used to solve the same problem instance, LRTA* improves its performance. Repeating this strategy, LRTA* eventually converges to optimal paths if $h$ is admissible.

The LRTA* algorithm with lookahead at depth 1 (the case considered in this paper) and converging to optimal paths (with $h$ admissible) appears in Figure 1. Like in [Korf, 1990], we assume the existence of $Succ$ and $h_0$ functions, which when applied to a state $x$ generate its set of successors and its initial heuristic estimate, respectively. Procedure `LRTA*` initializes the heuristic estimate of every state using the function $h_0$, and repeats the execution of `LRTA*-trial` until convergence ($h$ does not change). At this point, an optimal path has been found. Procedure `LRTA*-trial` performs a solving trial on the problem instance. It initializes the current state $x$ with the start $s$, and executes the following loop until finding a goal. First, it performs lookahead from $x$ at depth 1, updating its heuristic estimate accordingly (call to function `LookaheadUpdate1`). Second, it selects state $y$ of $Succ(x)$ with minimum value of $c(x, y) + h(y)$ as next state (breaking ties randomly). Third, it executes an action that passes from $x$ to $y$. At this point, $y$ is the new current state and the loop iterates. Note that the heuristic estimators computed in a trial are used as initial values in the next trial.

Function `LookaheadUpdate1` performs lookahead from $x$ at depth 1, and updates $h(x)$ if it is lower than the minimum cost of moving from $x$ to one of its successors $y$ plus its heuristic estimate $h(y)$. It returns $true$ if $h(x)$ changes, otherwise it returns $false$.

In a state space like the one assumed here (finite, positive costs, finite heuristic estimates) where from every state there is a path to a goal it has been proved that LRTA* is complete. In addition, if $h$ is admissible, over repeated trials the heuristic estimates eventually converge to their exact values along every optimal path [Korf, 1990].

## 3 LRTA*$(k)$

LRTA*$(k)$ is a LRTA*-based algorithm that propagates the changes of heuristic estimates up to $k$ states per iteration. Let $x$ be the current state. After lookahead, LRTA* updates $h(x)$. If it changes, this change is propagated on $Succ(x)$ with the same strategy, lookahead plus update. Let $y \in Succ(x)$ be a state that changes $h(y)$. This change is again propagated on $Succ(y)$ and so on. This process would iterate until no further changes, which could be very long and act on regions quite far from the current state. To make it acceptable for real-time search, we limit the updating capacity to $k$ states per iteration. This allows to make movements in bounded time (up to $k$ updates) [Koenig, 2001][Koenig, 2004] and to limit actions to the vicinity [Shimbo and Ishida, 2003]. We call this strategy *bounded propagation*.

```
procedure LRTA*$(X, A, c, s, G)$
  for each $x \in X$ do $h(x) \leftarrow h_0(x)$;          /* initialization */
  repeat
    LRTA*-trial$(X, A, c, s, G)$;
    until $h$ does not change;

procedure LRTA*-trial$(X, A, c, s, G)$
  $x \leftarrow s$;
  while $x \notin G$ do
    $dummy \leftarrow$ LookaheadUpdate1$(x)$;    /* look+upt */
    $y \leftarrow \text{argmin}_{w \in Succ(x)}[c(x, w) + h(w)]$;   /* next state */
    execute$(a \in A$ such that $a = (x, y))$; /* action exec */
    $x \leftarrow y$;

function LookaheadUpdate1$(x)$: boolean;
  $y \leftarrow \text{argmin}_{v \in Succ(x)}[c(x, v) + h(v)]$;       /* lookahead */
  if $h(x) < c(x, y) + h(y)$ then
    $h(x) \leftarrow c(x, y) + h(y)$;
    return $true$;
  else
    return $false$;
```

Figure 1: The LRTA* algorithm.

If we limit bounded propagation to states already visited, we can use the notion of support to avoid checking states that will not change. State $y$ is the *support* of $h(x)$, written $y = supp(x)$, iff $y = argmin_{v \in Succ(x)}(c(x, v) + h(v))$. If $h(y)$ changes, only those states $x$ successors of $y$ such that $y$ is their support could change. If $z$ is a previously visited state, $z \in Succ(y)$ but $y \neq supp(z)$, no matter $h(y)$ change $h(z)$ will not change because $supp(z)$ is the state with minimum value of $c(z, w) + h(w)$ with $w \in Succ(z)$.

With these ideas, bounded propagation and the use of supports, we have developed three versions of LRTA*$(k)$ that differ in the extra condition that a state, involved in the bounded propagation process, must satisfy to be considered for updating. They are explained in the following.

### 3.1 Original Version

The original version of LRTA*$(k)$ [Hernandez and Meseguer, 2005], from now on LRTA$^*_0(k)$, limits heuristic updating to states previously visited in the current trial. To do this, it keeps the sequence of expanded states in the current trial.

The LRTA$^*_0(k)$ algorithm appears in Figure 2. Procedure `LRTA*(k)` performs heuristic initialization and executes `LRTA*(k)-trial` until convergence ($h$ does not change). Procedure `LRTA*(k)-trial` differs from `LRTA*-trial` in three points: it initializes the support of every state to $null$, it records the sequence of expanded states in $path$, and it executes `LookaheadUpdateK` (that performs bounded propagation) instead of `LookaheadUpdate1`. Procedure `LookaheadUpdateK` performs bounded propagation as follows. It maintains a sequence $Q$ of states candidates to update their heuristic estimates. $Q$ is initialized with the current state. At most, $k$ states will be entered in $Q$. This is controlled by the counter $cont$, initialized to $k - 1$. Then, the following loop is executed until $Q$ contains no states. The first state $v$ in $Q$ is extracted, from which lookahead is performed and it is updated accordingly. If $h(v)$ changes (`LookaheadUpdate1` returns $true$), this is propagated over its successors as follows. Any $w$ successor of $v$ that belongs to $path$ enters $Q$ in the last position, provided that there is still room in $Q$ (the limit of $k$ states has not been exhausted during the current execution of the procedure). If $h(v)$ does

```
procedure LRTA*(k)$(X, A, c, s, G, k)$
  for each $x \in X$ do $h(x) \leftarrow h_0(x)$;
  repeat
    LRTA*(k)-trial$(X, A, c, s, G, k)$;
  until $h$ does not change;

procedure LRTA*(k)-trial$(X, A, c, s, G, k)$
  for each $x \in X$ do $supp(x) \leftarrow null$;
  $x \leftarrow s$;
  $path \leftarrow \langle s \rangle$;
  while $x \notin G$ do
    LookaheadUpdateK$(x, k, path)$;
    $y \leftarrow \text{argmin}_{w \in Succ(x)}[c(x, w) + h(w)]$;
    execute($a \in A$ such that $a = (x, y)$);
    $path \leftarrow$ add-last$(path, y)$;
    $x \leftarrow y$;

procedure LookaheadUpdateK$(x, k, path)$
  $Q \leftarrow \langle x \rangle$;
  $cont \leftarrow k - 1$;
  while $Q \neq \emptyset$ do
    $v \leftarrow$ extract-first$(Q)$;
    if LookaheadUpdate1$(v)$ then
      for each $w \in Succ(v)$ do
        if $w \in path \wedge cont > 0 \wedge v = supp(w)$ then
          $Q \leftarrow$ add-last$(Q, w)$;
          $cont \leftarrow cont - 1$;

function LookaheadUpdate1$(x)$: boolean;
  $y \leftarrow \text{argmin}_{v \in Succ(x)}[c(x, v) + h(v)]$;
  $supp(x) \leftarrow y$;
  if $h(x) < c(x, y) + h(y)$ then
    $h(x) \leftarrow c(x, y) + h(y)$;
    return $true$;
  else
    return $false$;
```

Figure 2: LRTA$^*_0(k)$: Original LRTA*$(k)$ algorithm.

not change, the loop iterates processing the next state of $Q$. After bounded propagation $h$ remains admissible (Lemma 2 of [Edelkamp and Eckerle, 1997]). Therefore, convergence of LRTA*$(k)$ to optimal paths is guaranteed, because Theorem 3 of [Korf, 1990] is also valid for LRTA*$(k)$.

### 3.2 First Version

The next version of LRTA*$(k)$, that we call LRTA$^*_1(k)$, limits heuristic updating to states already visited in the current or previous trials. To do this, it keeps the sequence of expanded states of the current or previous trials. We also maintain the table of supports between trials, because they are valid supports for visited states.

This version increases the effect of propagation after the first trial. From the second trial on, the agent have a higher possibility of making updates, because there are states in $path$ from the first step.

The LRTA$^*_1(k)$ algorithm appears in Figure 3. Differences with LRTA$^*_0(k)$ are located in procedure LRTA*(k), which initializes the table of supports and $path$ at the very beginning, instead of being initialized at each trial (as was made by procedure LRTA*(k)-trial in the original version).

```
procedure LRTA*(k)$(X, A, c, s, G, k)$
  for each $x \in X$ do
    $h(x) \leftarrow h_0(x)$;
    $supp(x) \leftarrow null$;
  $path \leftarrow \langle s \rangle$;
  repeat
    LRTA*(k)-trial$(X, A, c, s, G, k)$;
  until $h$ does not change;

procedure LRTA*(k)-trial$(X, A, c, s, G, k)$
  $x \leftarrow s$;
  while $x \notin G$ do
    LookaheadUpdateK$(x, k, path)$;
    $y \leftarrow \text{argmin}_{w \in Succ(x)}[c(x, w) + h(w)]$;
    execute($a \in A$ such that $a = (x, y)$);
    $path \leftarrow$ add-last$(path, y)$;
    $x \leftarrow y$;
```

Figure 3: LRTA$^*_1(k)$: First version of LRTA*$(k)$. Only procedures that change with respect to LRTA$^*_0(k)$ are shown.

### 3.3 Second Version

In the second version of LRTA*$(k)$, we give up the condition of updating on previously expanded states only. If $x$ is the current state and $h(x)$ changes, this change can be propagated to any state $y \in Succ(x)$. We call this second version LRTA$^*_2(k)$, and the algorithm appears in Figure 4.

Variable $path$ is no longer needed. Supports are defined for states previously visited. Now, since any state (visited or not) can be updated, the use of supports has to be limited to visited states. This is done in the procedure LookaheadUpdateK, which differentiates between successors with support (vis-

```
procedure LRTA*(k)$(X, A, c, s, G, k)$
  for each $x \in X$ do
    $h(x) \leftarrow h_0(x)$;
    $supp(x) \leftarrow null$;
  repeat
    LRTA*(k)-trial$(X, A, c, s, G, k)$;
  until $h$ does not change;

procedure LRTA*(k)-trial$(X, A, c, s, G, k)$
  $x \leftarrow s$;
  while $x \notin G$ do
    LookaheadUpdateK$(x, k, path)$;
    $y \leftarrow \text{argmin}_{w \in Succ(x)}[c(x, w) + h(w)]$;
    execute($a \in A$ such that $a = (x, y)$);
    $x \leftarrow y$;

procedure LookaheadUpdateK$(x, k, path)$
  $Q \leftarrow \langle x \rangle$;
  $cont \leftarrow k - 1$;
  while $Q \neq \emptyset$ do
    $v \leftarrow$ extract-first$(Q)$;
    if LookaheadUpdate1$(v)$ then
      for each $w \in Succ(v)$ do
        if $cont > 0 \wedge (supp(w) = v \vee supp(w) = null)$ then
            $Q \leftarrow$ add-last$(Q, w)$;
            $cont \leftarrow cont - 1$;
```

Figure 4: LRTA$^*_2(k)$: Second version of LRTA*$(k)$. Only procedures that change with respect to LRTA$^*_0(k)$ are shown.

ited) and without (not visited). A visited successor is entered in $Q$ if it is support of the state currently processed. A non visited successor is always entered in $Q$.

## 4 Experimental Results

In this Section we compare the performance of the three LRTA$^*(k)$ versions in the first trial, convergence and stability, for different values of $k$. The experimental analysis is centered in the comparison of the three versions, although we include results for RTA* and FALCONS for reference purposes [1]. As benchmarks we use four-connected grids where an agent can move one cell north, south, east or west, plus the popular 8-puzzle.

We use the following grids as benchmarks:

1. Grid35. Grids of size $301 \times 301$ with a 35% of obstacles placed randomly. In this type of grid heuristics tend to be only slightly misleading.
2. Grid70. Grids of size $301 \times 301$ with a 70% obstacles placed randomly. In this type of grid heuristics could be misleading.
3. Maze. Acyclic mazes of size $181 \times 181$ whose corridor structure was generated with depth-first search. In this type of grid heuristics could be very misleading.

In each case, results are averaged over 1000 different instances. In grids of size $301 \times 301$ the start and goal state are chosen randomly with the restriction that there is a path from the start state to the goal state. In mazes, the start state is (0,0), and the goal state is (180,180). In 8-puzzle, the initial state was chosen randomly. As initial heuristic between two states in four-connected grids we use the Manhattan distance. As initial heuristic in 8-puzzle we use the Manhattan distance defined as the sum, for all tiles, of their horizontal and vertical distances from their respective goal positions.

Results for Grid35, Grid70, Maze and 8-puzzle appear in Table 1, Table 2, Table 3 and Table 4, respectively. The results are presented in terms of solution cost ($\times 10^3$) , number of expanded states or memory ($\times 10^3$) and time per step ($\times 10^{-6}$ seconds), for the first trial, convergence (with the number of trials $\times 10^3$ to converge), and stability indexes [Shimbo and Ishida, 2003].

Regarding the first trial, there is no difference between LRTA$^*_0(k)$ and LRTA$^*_1(k)$, since they perform the same execution (differences between them appear from the second trial on). Comparing with LRTA$^*_2(k)$, in terms of cost this version improves over the previous ones for high $k$ ($k = 500$ in Grid35, $k = \infty$ in Grid70, $k = \infty$ in 8-puzzle), although it exhibits worse performance for small $k$. LRTA$^*_2(k)$ always requires more memory and longer planning steps than LRTA$^*_0(k)$/LRTA$^*_1(k)$.

Regarding convergence, in the grid benchmarks we observe that LRTA$^*_1(k)$ (almost) always improves over LRTA$^*_0(k)$ in number of trials. This improvement is minor with small $k$, and becomes substantial with large $k$. The same effect can be observed in the total solution cost. This improvement does not always causes a higher memory usage, but it requires longer planning steps (moderate increments for small $k$, substantial increments for large $k$). In the 8-puzzle, LRTA$^*_1(k)$ moderately improves over LRTA$^*_0(k)$ in terms of trials and cost, requiring a slightly longer planning step. Considering LRTA$^*_2(k)$, both gridworlds and 8-puzzle, it get worse than LRTA$^*_1(k)$ for small $k$ and only for large $k$ it achieves some improvement. LRTA$^*_2(k)$ always requires more memory and the planning step becomes longer (very long with high $k$).

Regarding solution stability, we computed the indices IAE, ISE, ITAE, ITSE, and SOD [Shimbo and Ishida, 2003]. IAE provides the sum of the error in the convergence. ISE provides the square of the sum of the error in the convergence, it penalizes large overshoots. ITAE and ITSE are two time-weighted versions of IAE and ISE, that impose large penalties on sustained errors. SOD sums up the difference in solution costs between two consecutive trials when the solution worsens. If SOD is equal to 0 convergence is monotonic.

In Grid35, LRTA$^*_1(k)$ shows better stability indexes than LRTA$^*_0(k)$ and LRTA$^*_2(k)$. In Grid70, LRTA$^*_1(k)$ is better than LRTA$^*_0(k)$ and LRTA$^*_2(k)$ only in SOD index for all values of $k$. For the other indexes there is no clear winner. Something similar happens in Maze, where there is no clear winner (except for $k = \infty$, where the winner is LRTA$^*_2(k)$). In 8-puzzle, results are very close. LRTA$^*_2(k)$ has the best results by a small margin.

In summary, we observe that for the four benchmarks tested, LRTA$^*_1(k)$ improves consistently over LRTA$^*_0(k)$, at the extra cost of more memory (in some cases) and a moderate-to-substantial increment in the planning step time. LRTA$^*_2(k)$ performance tends to decrease with small $k$, getting some moderate gains with a very high $k$, but it requires much longer planning steps. We consider that its benefits are small compared with the extra cost in planning time. Therefore, LRTA$^*_1(k)$ seems to be the algorithm of choice for these benchmarks. Other domains may require further experimentation.

## 5 Related Work

In his seminal work, Korf proposed LRTA* and RTA* (an algorithm with a different updating strategy, able to find better solutions in the first trial but without converging to optimal routes) [Korf, 1990]. After that, several approaches have been made to to improve LRTA* on the quality of the first solution, convergence and stability. LCM [Pemberton and Korf, 1992] keeps expanded states locally consistent (a state $x$ is locally consistent iff $h(x) = min_{v \in Succ(x)}[c(x,v) + h(v)]$), by propagating changes in the heuristic estimates. HLRTA* [Thorpe, 1994] is a hybrid between RTA* and LRTA*. It finds better solutions than LRTA* in the first trial and converges to optimal paths. As RTA*, it avoids the ping-pong effect [Edelkamp and Eckerle, 1997] but requires more memory. The weighted and bounded versions of LRTA* [Shimbo and Ishida, 2003] speed up convergence and improve solution stability, but sacrifice optimality. FALCONS[Furcy and Koenig, 2000] accelerates convergence and keeps optimality

[1] In [Hernandez and Meseguer, 2005], it can be found an analysis of LRTA$^*(k)$ performance compared with RTA* and FALCONS.

| Grid35 | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\text{LRTA}_0^*(k)$ | | | | | | | | | | | | |
| $k$ | **Cost%** | **Memory%** | **Time/Step%** | **Cost%** | **Trials%** | **Memory%** | **Time/Step%** | **IAE** | **ISE** | **ITAE** | **ITSE** | **SOD** |
| | 68.7=100% | 4.8=100% | 0.34=100% | 6493.8=100% | 2.5=100% | 44.9=100% | 0.36=100% | $\times 10^6$ | $\times 10^9$ | $\times 10^8$ | $\times 10^{11}$ | $\times 10^5$ |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 5.2 | 56.5 | 38.1 | 130.9 | 15.9 |
| 6 | 24% | 72% | 202% | 48% | 63% | 89% | 163% | 2.3 | 10.6 | 11.7 | 26.3 | 6.8 |
| 15 | 17% | 76% | 263% | 36% | 48% | 100% | 189% | 1.7 | 7.6 | 6.5 | 14.2 | 5.1 |
| 500 | 15% | 120% | 579% | 26% | 42% | 133% | 216% | 1.2 | 4.3 | 3.9 | 6.5 | 3.6 |
| $\infty$ | 14% | 126% | 782% | 25% | 42% | 136% | 223% | 1.1 | 3.7 | 3.8 | 6.2 | 3.4 |
| $\text{LRTA}_1^*(k)$ | | | | | | | | | | | | |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 5.2 | 56.5 | 38.1 | 130.9 | 15.9 |
| 6 | 24% | 72% | 202% | 46% | 67% | 100% | 182% | 2.2 | 9.3 | 11.7 | 24.5 | 6.5 |
| 15 | 17% | 76% | 263% | 28% | 41% | 100% | 231% | 1.3 | 5.1 | 4.4 | 8.8 | 3.9 |
| 500 | 15% | 120% | 579% | 5% | 6% | 100% | 685% | 0.3 | 1.3 | 0.1 | 0.3 | 0.8 |
| $\infty$ | 14% | 126% | 782% | 1.3% | 1.2% | 93% | 2220% | 0.07 | 0.5 | 0.006 | 0.02 | 0.2 |
| $\text{LRTA}_2^*(k)$ | | | | | | | | | | | | |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 5.2 | 56.5 | 38.1 | 130.9 | 15.9 |
| 6 | 30% | 112% | 191% | 52% | 76% | 153% | 170% | 2.4 | 10.7 | 14.6 | 30.5 | 7.3 |
| 15 | 20% | 117% | 256% | 32% | 47% | 154% | 222% | 1.5 | 6.5 | 5.7 | 11.8 | 4.5 |
| 500 | 9% | 181% | 894% | 6% | 8% | 153% | 736% | 1.1 | 3.7 | 3.8 | 6.2 | 3.4 |
| $\infty$ | 7% | 206% | 1762% | 0.7% | 0.9% | 145% | 4306% | 0.3 | 1.1 | 0.2 | 0.4 | 0.9 |
| RTA* | 45% | 66% | 91% | - | - | - | - | - | - | - | - | - |
| FALCONS | 1402% | 183% | 126% | 62% | 26% | 245% | 122% | 3.7 | 3898.1 | 6.3 | 88.3 | 10.6 |

Table 1: Grid35 results for the first trial (left), convergence (middle) and stability (right) for $\text{LRTA}_0^*(k)$, $\text{LRTA}_1^*(k)$ and $\text{LRTA}_2^*(k)$. Average over 1000 instances.

| Grid70 | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\text{LRTA}_0^*(k)$ | | | | | | | | | | | | |
| $k$ | **Cost%** | **Memory%** | **Time/Step%** | **Cost%** | **Trials%** | **Memory%** | **Time/Step%** | **IAE** | **ISE** | **ITAE** | **ITSE** | **SOD** |
| | 146.3=100% | 1.4=100% | 0.34=100% | 790.0=100% | 0.3=100% | 1.6=100% | 0.37=100% | $\times 10^3$ | $\times 10^6$ | $\times 10^4$ | $\times 10^7$ | $\times 10^2$ |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 509.1 | 24227.7 | 2029.2 | 9145.2 | 1174.8 |
| 6 | 34% | 99% | 182% | 44% | 57% | 100% | 148% | 188.4 | 2931.4 | 702.6 | 1265.5 | 592.0 |
| 15 | 18% | 100% | 240% | 27% | 37% | 103% | 179% | 112.9 | 950.8 | 301.7 | 501.4 | 379.7 |
| 500 | 4% | 104% | 769% | 10% | 17% | 107% | 302% | 26.5 | 75.2 | 25.0 | 26.8 | 65.3 |
| $\infty$ | 2% | 108% | 5236% | 2% | 5% | 111% | 906% | 4.0 | 5.9 | 2.7 | 1.1 | 5.8 |
| $\text{LRTA}_1^*(k)$ | | | | | | | | | | | | |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 509.1 | 24227.7 | 2029.2 | 9145.2 | 1174.8 |
| 6 | 34% | 99% | 182% | 45% | 56% | 100% | 158% | 200.0 | 3110.0 | 600.0 | 1520.0 | 533.0 |
| 15 | 18% | 100% | 240% | 26% | 33% | 100% | 195% | 100.0 | 1110.0 | 231.0 | 501.0 | 327.0 |
| 500 | 4% | 104% | 769% | 4% | 6% | 100% | 557% | 20.0 | 64.0 | 8.1 | 16.4 | 51.2 |
| $\infty$ | 2% | 108% | 5236% | 0.8% | 1.2% | 100% | 2715% | 2.0 | 4.7 | 0.3 | 0.5 | 0.2 |
| $\text{LRTA}_2^*(k)$ | | | | | | | | | | | | |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 509.1 | 24227.7 | 2029.2 | 9145.2 | 1174.8 |
| 6 | 42% | 178% | 184% | 53% | 67% | 185% | 157% | 231.1 | 4480.0 | 791.6 | 2070.5 | 618.0 |
| 15 | 25% | 178% | 247% | 32% | 41% | 187% | 199% | 139.5 | 1670.8 | 333.2 | 732.1 | 383.8 |
| 500 | 4% | 182% | 882% | 5% | 7% | 187% | 646% | 22.4 | 88.8 | 11.4 | 23.4 | 64.7 |
| $\infty$ | 1.1% | 175% | 4279% | 0.7% | 1.2% | 186% | 3666% | 1.6 | 1.7 | 0.2 | 0.2 | 0.4 |
| RTA* | 29% | 101% | 97% | - | - | - | - | - | - | - | - | - |
| FALCONS | 53% | 103% | 105% | 81% | 55% | 104% | 103% | 480.9 | 17268.5 | 4088.3 | 46665.2 | 1244.5 |

Table 2: Grid70 results for the first trial (left), convergence (middle) and stability (right) for $\text{LRTA}_0^*(k)$, $\text{LRTA}_1^*(k)$ and $\text{LRTA}_2^*(k)$. Average over 1000 instances.

| Maze | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\text{LRTA}_0^*(k)$ | | | | | | | | | | | | |
| $k$ | **Cost%** | **Memory%** | **Time/Step%** | **Cost%** | **Trials%** | **Memory%** | **Time/Step%** | **IAE** | **ISE** | **ITAE** | **ITSE** | **SOD** |
| | 588.0=100% | 8.2=100% | 0.35=100% | 27767.1=100% | 1.5=100% | 13.8=100% | 0.35=100% | $\times 10^5$ | $\times 10^9$ | $\times 10^6$ | $\times 10^{10}$ | $\times 10^4$ |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 221.6 | 5338.3 | 6952.1 | 120297.1 | 1331.1 |
| 6 | 29% | 92% | 186% | 39% | 48% | 120% | 169% | 82.0 | 403.8 | 2109.0 | 4490.6 | 412.4 |
| 15 | 16% | 89% | 252% | 23% | 24% | 128% | 232% | 50.7 | 200.3 | 659.2 | 1242.9 | 188.8 |
| 500 | 4% | 89% | 1116% | 5% | 7% | 120% | 663% | 9.6 | 28.4 | 34.8 | 66.6 | 48.6 |
| $\infty$ | 2% | 90% | 8513% | 2% | 6% | 116% | 1792% | 1.5 | 1.6 | 4.2 | 1.6 | 7.0 |
| $\text{LRTA}_1^*(k)$ | | | | | | | | | | | | |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 221.6 | 5338.3 | 6952.1 | 120297.1 | 1331.1 |
| 6 | 29% | 92% | 186% | 41% | 47% | 107% | 180% | 87.3 | 1100.0 | 1580.0 | 15000.0 | 531.0 |
| 15 | 16% | 89% | 252% | 23% | 25% | 111% | 237% | 50.5 | 485.0 | 556.0 | 4010.0 | 301.8 |
| 500 | 4% | 89% | 1116% | 4% | 3% | 112% | 881% | 8.0 | 43.1 | 14.3 | 57.6 | 42.6 |
| $\infty$ | 2% | 90% | 8513% | 0.1% | 0.2% | 112% | 20541% | 0.3 | 0.4 | 0.05 | 0.08 | 1.1 |
| $\text{LRTA}_2^*(k)$ | | | | | | | | | | | | |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 221.6 | 5338.3 | 6952.1 | 120297.1 | 1331.1 |
| 6 | 53% | 198% | 190% | 52% | 65% | 199% | 176% | 107.0 | 1460.0 | 2720.0 | 27600.0 | 671.0 |
| 15 | 36% | 201% | 259% | 31% | 35% | 199% | 238% | 66.8 | 736.0 | 1010.0 | 8260.0 | 402.0 |
| 500 | 9% | 211% | 1082% | 5% | 5% | 200% | 998% | 10.4 | 60.7 | 24.4 | 106.0 | 54.3 |
| $\infty$ | 2% | 198% | 13662% | 0.1% | 0.2% | 212% | 28549% | 0.2 | 0.3 | 0.05 | 0.06 | 0.8 |
| RTA* | 6% | 88% | 96% | - | - | - | - | - | - | - | - | - |
| FALCONS | 15% | 129% | 106% | 78% | 50% | 134% | 105% | 189.4 | 1136.2 | 7072.6 | 43695.6 | 680.8 |

Table 3: Maze results for the first trial (left), convergence (middle) and stability (right) for $\text{LRTA}_0^*(k)$, $\text{LRTA}_1^*(k)$ and $\text{LRTA}_2^*(k)$. Average over 1000 instances.

| 8-puzzle | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\text{LRTA}_0^*(k)$ | | | | | | | | | | | | | |
| $k$ | **Cost%** | **Memory%** | **Time/Step%** | **Cost%** | **Trials%** | **Memory%** | **Time/Step%** | **IAE** | **ISE** | **ITAE** | **ITSE** | **SOD** |
| | 0.361=100% | 0.208=100% | 0.57=100% | 87.1=100% | 0.272=100% | 33.8=100% | 0.53=100% | $\times 10^4$ | $\times 10^7$ | $\times 10^7$ | $\times 10^9$ | $\times 10^4$ |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 8.0 | 4.7 | 1.8 | 9.2 | 4.3 |
| 4 | 61% | 93% | 101% | 55% | 81% | 84% | 104% | 4.3 | 1.6 | 0.8 | 2.7 | 2.4 |
| 16 | 66% | 102% | 102% | 43% | 62% | 71% | 105% | 3.3 | 1.3 | 0.5 | 1.7 | 1.8 |
| 32 | 65% | 101% | 105% | 43% | 62% | 71% | 105% | 3.3 | 1.3 | 0.5 | 1.6 | 1.7 |
| 128 | 63% | 98% | 104% | 43% | 62% | 72% | 105% | 3.3 | 1.3 | 0.5 | 1.6 | 1.7 |
| $\infty$ | 68% | 106% | 102% | 43% | 62% | 72% | 105% | 3.3 | 1.3 | 0.5 | 1.7 | 1.7 |
| $\text{LRTA}_1^*(k)$ | | | | | | | | | | | | |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 8.0 | 4.7 | 1.8 | 9.2 | 4.3 |
| 4 | 61% | 93% | 101% | 57% | 85% | 89% | 104% | 4.4 | 1.7 | 0.9 | 2.9 | 2.5 |
| 16 | 66% | 102% | 102% | 39% | 54% | 77% | 107% | 3.1 | 1.3 | 0.4 | 1.5 | 1.7 |
| 32 | 65% | 101% | 105% | 38% | 51% | 76% | 108% | 3.0 | 1.3 | 0.4 | 1.4 | 1.6 |
| 128 | 63% | 98% | 104% | 38% | 50% | 75% | 109% | 3.0 | 1.2 | 0.4 | 1.4 | 1.5 |
| $\infty$ | 68% | 106% | 102% | 38% | 50% | 75% | 109% | 3.0 | 1.2 | 0.3 | 1.4 | 1.5 |
| $\text{LRTA}_2^*(k)$ | | | | | | | | | | | | |
| 1 (LRTA*) | 100% | 100% | 100% | 100% | 100% | 100 | 100% | 8.0 | 4.7 | 1.8 | 9.2 | 4.3 |
| 4 | 78% | 127% | 104% | 58% | 86% | 92% | 105% | 4.4 | 1.7 | 0.9 | 2.8 | 2.5 |
| 16 | 62% | 193% | 115% | 36% | 62% | 103% | 113% | 2.7 | 0.9 | 0.4 | 1.1 | 1.5 |
| 32 | 63% | 266% | 116% | 30% | 55% | 111% | 118% | 2.3 | 0.7 | 0.3 | 0.7 | 1.2 |
| 128 | 62% | 325% | 123% | 26% | 49% | 124% | 126% | 1.9 | 0.6 | 0.2 | 0.5 | 1.0 |
| $\infty$ | 60% | 317% | 123% | 26% | 48% | 127% | 128% | 1.9 | 0.6 | 0.2 | 0.5 | 1.0 |
| RTA* | 63% | 97% | 99% | - | - | - | - | - | - | - | - | - |
| FALCONS | 250% | 258% | 102% | 41% | 30% | 48% | 103% | 3.4 | 3.4 | 2.8 | 1.9 | 1.6 |

Table 4: 8-puzzle results for the first trial (left), convergence (middle) and stability (right) for $\text{LRTA}_0^*(k)$, $\text{LRTA}_1^*(k)$ and $\text{LRTA}_2^*(k)$. Average over 1000 instances.

by using $g(x) + h(x)$ as heuristic function, where $g(x)$ is the cost from the start state to $x$. FALCONS improves convergence but it may perform a large amount of exploration in earlier trials. eFALCONS [Furcy and Koenig, 2001] is a hybrid between HLRTA* and FALCONS. It converges as FALCONS, performing a smaller amount of actions in earlier trials, although it may be greater than LRTA*. A new version of LRTA* [Koenig, 2004] improves convergence by increasing lookahead depth, causing to increase the planning time per step.$\gamma-$Trap [Bulitko, 2004] uses adaptive lookahead depth and offers control on the exploration vs. exploitation trade-off, but sacrifices optimality. Other approaches consider search with moving target [Ishida and Korf, 1991] and cooperative agents [Knight, 1993], [Goldenberg *et al.*, 2003].

The idea that motivated LRTA*$(k)$, bounded propagation of changes in heuristic estimates, is clearly related with the LCM algorithm, which performs an unbounded propagation of those changes. In fact, LCM can be seen as an $\text{LRTA}_0^*(k = \infty)$ without using supports. In addition, LCM is able to switch to a shortest path approach in case unbounded propagation exceeds some limit of updates (see [Pemberton and Korf, 1992] for further details).

## 6 Conclusions

LRTA*$(k)$ is a real-time search algorithm that converges to optimal routes. Based on LRTA*, it performs bounded propagation of heuristic changes up to $k$ states, which causes longer planning steps. Experimentally, LRTA*$(k)$ shows a substantial performance improvement with respect to LRTA* and FALCONS, in terms of first trial, convergence and solution stability. Improvements depend on the $k$ value: the higher $k$, the better results at the cost of longer planning steps.

We have presented three versions of LRTA*$(k)$, that differ in the extra condition for a state to be considered for updating. $\text{LRTA}_0^*(k)$ limits heuristic updating to states previously visited in the current trial, $\text{LRTA}_1^*(k)$ to states previously visited in the current or previous trials, and $\text{LRTA}_2^*(k)$ requires no extra condition. Experimentally, we show that $\text{LRTA}_1^*(k)$ solves benchmarks better than $\text{LRTA}_0^*(k)$, at the cost of longer planning steps. When comparing $\text{LRTA}_2^*(k)$ against $\text{LRTA}_1^*(k)$, the former does not solve benchmarks better than the later, although $\text{LRTA}_2^*(k)$ always uses more memory and time per planning step. We think these versions could be very useful for applications with different requirements on the time available between execution of consecutive actions.

## Acknowledgements

We thank the anonymous reviewers, whose comments greatly improved the paper. They pointed out the LCM algorithm [Pemberton and Korf, 1992], initially overlooked by us.

## References

[Bulitko, 2004] V. Bulitko. Learning for adaptive real-time search. *The Computing Research Rep. (CoRR): cs.DC/0407017*, 2004.

[Edelkamp and Eckerle, 1997] S. Edelkamp and J. Eckerle. New strategies in learning real time heuristic search. In *Proc. AAAI Workshop on On-Line Search*, pages 30–35, 1997.

[Furcy and Koenig, 2000] D. Furcy and S. Koenig. Speeding up the convergence of real-time search. In *Proc. AAAI*, 2000.

[Furcy and Koenig, 2001] D. Furcy and S. Koenig. Combining two fast-learning real-time search algorithms yields even faster learning. In *Proc. 6th European Conference on Planning*, 2001.

[Goldenberg *et al.*, 2003] M. Goldenberg, A. Kovarksy, X. Wu, and J. Schaeffer. Multiple agents moving target search. In *Proc. 18th IJCAI*, 2003.

[Hernandez and Meseguer, 2005] C. Hernandez and P. Meseguer. Lrta*(k). In *Proc. IJCAI*, 2005.

[Ishida and Korf, 1991] T. Ishida and R. E. Korf. Moving target search. In *Proc 12th IJCAI*, 1991.

[Knight, 1993] K. Knight. Are many reactive agents better than a few deliberative ones? In *Proc. 13th IJCAI*, 1993.

[Koenig, 2001] S. Koenig. Agent-centered search. *Artificial Intelligence Magazine*, 22(4):109–131, 2001.

[Koenig, 2004] S. Koenig. A comparison of fast search methods for real-time situated agents. In *Proc. 3rd AAMAS*, 2004.

[Korf, 1990] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[Pemberton and Korf, 1992] J. Pemberton and R. E. Korf. Making locally optimal decisions on graphs with cycles. Technical Report 920004, Computer Science Dep. UCLA, 1992.

[Shimbo and Ishida, 2003] M. Shimbo and T. Ishida. Controlling the learning process of real-time heuristic search. *Artificial Intelligence*, 146(1):1–41, 2003.

[Thorpe, 1994] P. E. Thorpe. A hybrid learning real-time search algorithm. Master's thesis, Computer Science Dep., UCLA, 1994.

# Learning Task Allocation via Multi-level Policy Gradient Algorithm with Dynamic Learning Rate

**Sherief Abdallah** and **Victor Lesser**
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003
{shario,lesser}@cs.umass.edu

## Abstract

Task allocation is the process of assigning tasks to appropriate resources. To achieve scalability, it is common to use a network of agents (also called mediators) that handles task allocation. This work proposes a novel multi-level policy gradient algorithm to solve the local decision problem at each mediator agent. The higher level policy stochastically chooses a task decomposition. The lower level policy assigns subtasks to neighboring agents also stochastically. Agents learn autonomously, cooperatively, and concurrently to increase system performance. No state information is used except for the task being allocated. Furthermore, the algorithm dynamically adjusts the learning rate, to speed up convergence, using the ratio of action values. Experimental results show how our proposed solution outperforms other deterministic approaches by balancing the load over resources and converging faster to better policies.

## 1 Introduction

Task allocation is the process of assigning tasks to appropriate resources. The problem appears in many real applications like the Grid, web services, sensor nets and other domains[Czajkowski and et al, 2001]. Consider the web services as an example. In that domain there are servers distributed over the web. Each server provides a set of services for applications. Users may appear anywhere in the web asking for a composition of services (also called a task) that requires more than one server. Any server can work on more than one task at a time. However, the cost of executing a task increases in proportion to the total number of tasks being serviced by the server. Since users usually do not know where servers are, a network of agents (also called mediators) that know about different servers is used. Such agents take requests from users and reply to users with the appropriate set of servers.

This work illustrates how agents in such a network can learn to work cooperatively in order to optimize the task allocation problem. In particular, this work proposes a novel multi-level policy gradient algorithm to optimize the local decision of each agent in the network. The higher level policy stochastically chooses a task decomposition. The lower level policy stochastically assigns subtasks to neighboring agents. Agents learn autonomously, cooperatively, and concurrently to minimize the cost of executing tasks. The algorithm does not use any state information except the type of the task being allocated and estimates of the cost for assigning task types to neighbors. Furthermore, to speed up convergence, our proposed algorithm dynamically changes the learning rate in proportion to the cost of choosing an action (whether this action is choosing a decomposition or assigning a task to a neighbor).

Two factors make the problem both interesting and challenging: the limited local view of each agent and the need for load balancing. In large distributed systems, having a global view of the system's state is impossible from practical point of view. Agents usually rely on their limited local view and use communication to augment this view. This is a trade-off between optimality and scalability. In our system, the only a *priori* knowledge known by an agent (as will be described shortly) is the addresses of its direct neighbors. Furthermore, agents do not communicate their states, but rely solely on the statistical outcomes of interacting with neighboring agents. In other words, agent $A$'s knowledge about its neighbor agent $B$ is summarized via a statistical average of previous outcomes when $A$ tried assigning a task to $B$.

What makes load balancing needed in many real systems is the nonlinear increase of task execution cost with respect to the increase in load. Cost here is a signal of the system's performance. For example, cost may increase due to an increase in task waiting time to indicate a reduction in users' satisfaction. Therefore, in real systems, it is almost always better to divide the load as fairly as possible among servers and resources. The algorithm presented in this paper aims at balancing the load over servers/resources. Experimental results show how our algorithm significantly outperforms deterministic approaches that ignore load balancing.

The paper is organized as follows. The rest of this section presents a motivating example. Then a formal problem definition is presented, followed by a description of local agent decision problem. Next a description of our algorithm that optimizes the local agent decision is given. Experimental results are then presented and discussed, showing how our algorithm outperforms other deterministic approaches. Then a discussion of related work is given. We finally conclude and lay out our future directions.

### 1.1 Motivating Example

To get a better insight into the complexity of the decision making of each agent, consider the example in Figure 1. This system consists of three agents, $MA$, $MD$, and $MF$. Each agent is connected to two resources. There are two main types of resources, $A$ and $B$. Resource $A_i$ is of type $A$ and can undertake only task type $TA$. Similarly, resource $B_i$ is of type $B$ and can only undertake task type $TB$. Resources $A_f$ and $B_f$ are of types $A$ and $B$ respectively but they are fast resources that need half the time of other resources to finish their tasks. Task $TAB$ is a more complex task that has three alternative decompositions: $\{TA, TA\}$, $\{TB, TB\}$ and $\{TA, TB\}$. However, only agent $MD$ knows how to decompose task $TAB$.

Suppose agent $MA$ receives many tasks of type $TA$. If $MA$ always chooses $A1$ to assign $TA$ to (i.e. deterministic policy), then $A1$ will be overloaded and its cost rapidly increases. After several trials, $MA$ will not see $A1$ as appealing and will switch its policy to another neighbor. As the other neighbor gets overloaded $MA$ will switch again and so on. This means that $MA$ will not converge on a deterministic policy and will keep oscillating after spikes of high costs due to overloading. One would expect a stochastic policy, where $MA$ chooses each neighbor with a certain probability, would perform better. Similarly, suppose agent $MD$ receives a task of type $TAB$. $MD$ then needs to choose among the three possible decompositions of $TAB$. Each decomposition imposes certain load patterns on the system. For example, always choosing the decomposition $\{TA, TB\}$ means there will be equal load on both resource types $A$ and $B$.

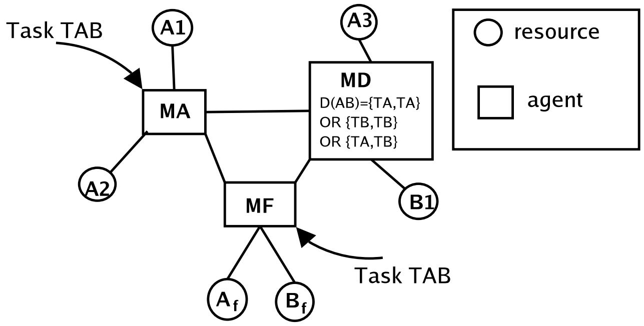


Figure 1: A network of agents that are responsible for assigning resources to incoming tasks.

## 2 Problem Definition

Let $T = \{T_1, ..., T_{|T|}\}$ be the set of task types. Different instances of tasks types appear randomly at different agents. Each task instance $I_j$ is defined by an arrival time $t_{I_j}$, a task type $T_{I_j} \in T$, and a payoff $O_{I_j}$. A decomposition function $D(T_i) = \{d_1, ..., d_{|D(T_i)|}\}$ associates with each task a set of decompositions, where $d_i \subseteq T$ (hence $D(T_i)$ provides alternative ways to do task $T_i$). The system has a set of resources $R = \{R_1, ..., R_{|R|}\}$. A resource $R_i$ can undertake a set of task types $H_{R_i} \subseteq T$. $\forall T_j \in H_{R_i} : F_{R_i}(T_j) > 0$ is the time resource $R_i$ needs to finish a task of type $T_j$. The cost of executing a task $T_j$ at resource $R_i$ at time $t$ is $C^t(T_j, R_i)$. The cost is time-dependent because it depends on the total load on resource $R_i$ at that time. The goal is to optimize the allocation of tasks to appropriate resources such that net profit (which is payoff reduced by total cost) over period of time $\Delta$ is maximized. More formally, the global system goal is to maximize the objective function $\Gamma$ defined as follows.

$$\Gamma = \sum_{I:t_I \in \Delta} O_I - \sum_{\langle T_i, R_j \rangle \in A} C^{t_I}(T_i, R_j)$$

where $A = \{a_1, ..., a_{|A|}\}$ is a set of task-resource assignments, where $a_i = \langle T_i, R_i \rangle$. However, because there is not any centralized entity that has a global view of the whole system, evaluating and optimizing $\Gamma$ is practically impossible. Instead, one needs a local objective function $\Gamma_{M_x}$ that each agent $M_x$ attempts to optimize. Let $M = \{M_1, ..., M_{|M|}\}$ be the set of agents interconnecting the set of resources $R$. Each agent $M_x$ has a set of neighbors $N(M_x) \subseteq M \cup R$. Each agent knows of a set of decompositions $D_{M_x}$, where $D_{M_x}(T_j) \subseteq D(T_j)$. The goal of each agent $M_x$ is to allocate incoming task instance $I = \langle t_I, T_I, O_I \rangle$ to neighboring agents such that $\Gamma_{M_x}(I)$ is maximized, where

$$\Gamma_{M_x}(I) = O_I - \sum_{\langle T_i, n_j \rangle \in A_{M_x}(I,d)} C^{t_I}(T_i, n_j)$$

where $A_{M_x}(I, d) = \{a_k : a_k = \langle T_i, n_j \rangle\}$ is a set of task-neighbor assignments, where $n_j \in N(M_x)$ and $T_i \in d \in D_{M_x}(T_I)$. $C^{t_I}(T_i, n_j)$ is the cost of assigning task $T_i$ to neighbor $n_j$. In other words, agent $M_i$ needs to find both a decomposition $d$ and an assignment of neighbors to each of the subtask types in $d$ such that the total *estimated* cost of executing $I$ is maximized (note that cost is negative). The cost $C^{t_I}(T_i, n_j)$ is only an estimation because it depends on how agent $n_j$ will conduct the allocation of $T_i$. For example, if $n_j$ is still learning then it is likely that the cost will be higher than the real cost (e.g. because $n_j$ is allocating $T_i$ poorly). As agents interact with each other, one would hope that the local agent policies converge to good (if not optimal) collective policy. Therefore, the local objective function at each agent $\Gamma_{M_i}$ only approximates the global objective function $\Gamma$. However, as the results in this paper show, using our algorithm agents successfully converge and learn cooperate in allocating tasks. The following section presents our algorithm

## 3 Multi-level policy gradient algorithm

An agent, in a task allocation framework, makes its decision in a two-step process. First, it chooses a decomposition from the set of alternative decompositions. Then for each subtask in the chosen decomposition the agent chooses one of its neighbors to assign. Formally, each agent needs to learn two policies: $\pi_{high}(T_i, d_j)$ and $\pi_{low}(T_i, n_j)$. $\pi_{high}(T_i, d_j)$ is the probability of choosing decomposition $d_j \in D(T_i)$, while $\pi_{low}(T_i, n_j)$ is the probability of choosing neighbor $n_j$ to assign to task $T_i$. Any of the two policies (or both) can be deterministic (e.g. $\forall T_i \exists n_j : \pi(T_i, n_j) = 1$). However, one would expect deterministic policies to be suboptimal as they can not balance the load as well as stochastic policies.

While $\pi_{low}$ could have been conditioned on the chosen composition (i.e. $\pi_{low}^{d_k}(T_i, n_j)$, where $d_k$ is the chosen decomposition by $\pi_{high}$) we opted to make both $\pi_{high}$ and $\pi_{low}$ independent. This speeds up learning, because a single $\pi_{low}$ is shared across decompositions and across tasks, but is not always optimal. For example, consider again the scenario in Figure 1. Let agent $MA$ receives task $TAB$ and assume $MA$ can decompose $TAB$ to $\{TA, TA\}$. Now the cost of assigning one task $TA$ to $A1$ is not independent of the decomposition. The cost depends on how the other task is assigned (if both are assigned to the same agent then the cost should be higher). Nevertheless, in most cases this is a valid approximation as verified by our results.

Agents communicate with each other using messages. There are only two types of messages: REQUEST and RESPONSE. A agent $M_{sender}$ sends a REQUEST message to agent $M_{receiver}$ asking it to accomplish certain task. $M_{receiver}$ estimates the cost for accomplishing the requested task (as will be described shortly) and sends a RESPONSE message, with the estimated cost, back to $M_{sender}$. Therefore, the operation of each agent is driven by received messages (i.e. event driven) and is divided into two algorithms for processing each message type. Algorithm 1 is where decision making occurs (deciding how to decompose a task and assign subtasks) while Algorithm 2 is where learning takes place.

**Algorithm 1**: Process REQUEST message

```
Input: REQUEST from $M_{sender}$ to $M_{receiver}$ to do task $T_i$
begin
    Choose a decomposition $d^*$ uniformly at random
    proportional to $\pi_{high}(T_i, d_j), \forall d_j \in D(T_i)$.
    for each task $T_k \in d^*$ choose a neighbor $n_l$ uniformly at
    random proportional to $\pi_{low}(T_k, n_l)$,
    $\forall n_l \in N(M_{receiver})$. Let $A = \{a_1, ..., a_{|d^*|}\}$ be the
    chosen set of assignments for each subtask of $d^*$, where
    $a_k = \langle T_{a_k}, n_{a_k} \rangle$.
    Send a RESPONSE message to $M_{sender}$ with the
    estimated cost of $A$, $C_{T_i} = \sum_{\langle T_i, n_j \rangle \in A} C(T_i, n_j)$.
    Send REQUEST messages to neighbors according to $A$.
end
```

## 3.1 Learning

Learning a stochastic policy is usually slower and more difficult than learning a deterministic policy (Q-learning [Sutton and Barto, 1999] is a well known and understood learning algorithm for deterministic policies). Learning a stochastic policy usually involves some sort of policy gradient algorithms as described in Algorithm 2. The main unknown variable for each agent is the cost of assigning a certain task type to a neighbor. This negative value can be learned using a simple update equation derived from Q-routing [Boyan and Littman, 1994]: $C(T_i, n_j) \leftarrow (1 - \alpha)C(T_i, n_j) + \alpha C^{new}(T_i, n_j)$. The equation merges previous cost estimate, $C(T_i, n_j)$ , with a newly received cost estimate, $C^{new}(T_i, n_j)$, using a weight parameter $\alpha$.

**Algorithm 2**: Process RESPONSE message

```
Input: RESPONSE from neighbor $n_j$ regarding task $T_i$ with
       estimated cost $C$
begin
    Let $n^* = argmax_{n_j} C(T_i, n_j)$.
    update the cost $C(T_i, n_j) \leftarrow (1 - \alpha)C(T_i, n_j) + \alpha C$.
    update policy (either deterministically or stochastically as
    described shortly)
end
```

Updating policies $\pi_{low}$ and $\pi_{high}$ can be done either deterministically using Q-routing-based [Boyan and Littman, 1994] approach or stochastically using policy gradient approach. Algorithm 3 shows the deterministic approach while Algorithm 4 shows the policy gradient approach. Experimental results compares both extremes and hybrids of them.

**Algorithm 3**: Deterministic Policy Update

```
Input: task $T_i$
begin
    $\forall n_j : \pi_{low}(T_i, n_j) \leftarrow 1$ iff $n_j = argmax_k C_{n_k}(T_i)$
    otherwise $\pi_{low}(T_i, n_j) \leftarrow 0$
    $\forall T_l, d_j$ s.t. $T_i \in d_j$ and $d_j \in D(T_l) : \pi_{high}(T_l, d_j) \leftarrow 1$
    iff $d_j = argmax_k \sum_{T_u \in d_k} max_m C_{n_m}(T_u)$
    otherwise $\pi_{high}(T_l, d_j) \leftarrow 0$
end
```

**Algorithm 4**: Policy Gradient Update

```
Input: task $T_i$ and neighbor $n_j$
begin
    $\pi_{low}(T_i, n_j) \leftarrow \pi_{low}(T_i, n_j) + \delta$ iff
    $n_j = argmax_k C_{n_k}(T_i)$
    otherwise $\pi_{low}(T_i, n_j) \leftarrow \pi(T_i, n_j) - \delta$
    normalize $\pi_{low}$ s.t. $\sum_{n_j} \pi(T_i, n_j) = 1$
    $\forall T_k, d_l$ s.t. $d_l \in D(T_k)$ and $T_i \in d_l$ :
    $\pi_{high}(T_k, d_l) \leftarrow \pi_{high}(T_k, d_l) + \delta$ if $d_l$ is the best
    decomposition for $T_k$
    otherwise $\pi_{high}(T_k, d_l) \leftarrow \pi(T_k, d_l) - \delta$
    normalize $\pi_{high}$ s.t. $\sum_{d_j} \pi(T_i, d_j) = 1$
end
```

The policy gradient algorithm above uses a fixed learning rate $\delta$. The smaller $\delta$ is the more careful our algorithm explores the policy space, and hence the more likely it will converge to an optimal policy. However, the smaller the $\delta$ is the slower the convergence. In this work we propose using dynamic learning rates that are derived from learned costs. The use of different learning rate of an agent depending on the agent's performance has been proposed before [Michael Bowling, 2002]. However, previous work only used two fixed values of learning rates. We propose taking advantage of the consistency of the cost estimates (all are non positives) and scales $\delta$ accordingly. In particular, line 4.7 is modified to "otherwise $\pi_{high}(T_k, d_l) \leftarrow \pi(T_k, d_l) - \delta \frac{C(T_k,d_l)}{max_{d_u} C(T_k,d_u)}$", where $C(T_k, d_u)$ is the maximum cost of allocating $T_k$ if decomposition $d_u \in T_k$ is chosen; i.e. $C(T_k, d_u) = \sum_{T_i \in d_u} max_{n_j} C(T_i, n_j)$. To prevent spikes in learning rate, especially in the beginning of learning, the learning rate is not allowed to surpass a threshold $\delta_{max}$.

### 3.2 Cycles

Like any routing algorithm, it is possible to have cyclic policies. For example, two neighboring agents may send the same task back and forth between each other. Such a policy is undesirable as it wastes system resources without getting any real work done. This problem has two aspects. The first is detecting such a cycle. The second is choosing an appropriate reinforcement signal to penalize such behavior.

There are two known methods to detect cycles. The first method assigns a unique identifier for each task. Each agent then keeps track of task identifiers it had seen. A cycle is detected once a task identifier is seen twice. Two problems make this approach unappealing: ensuring the uniqueness of task identifiers across the distributed network and deciding for how long to keep task identifiers. A simpler yet approximate approach is to use the *age* of a task to detect a cycle. If a task has been floating in the system for too long, then it is likely that there is a cycle. What makes this approach approximate is defining the maximum age. Optimally, maximum age should be the diameter of the network. However, in an open and dynamic system, it is unlikely that any agent would know the diameter of the network. We use the second approach in our experiments.

Once a cycle is detected at an agent, the system faces the credit assignment problem: determining who is/are responsible and penalizing them. Several factors make this problem difficult: use of stochastic policies, partial observability, and using task age for detecting cycles. All these factors add uncertainty to determining who is/are responsible for the cycle. For example, the agent that received an old task may not even be part of the cycle. The work in [Tao *et al.*, 2001] used a global penalty signal (i.e. all agents are penalized once a cycle is detected). Their approach does not scale to a large open system. Our work on the other hand uses a local penalty: the agent who received a too old task sends a high negative penalty to the sender of that task. Experimental results show the effectiveness of this approach in conjunction with our learning algorithm.

## 4 Experimental Results

The first part of our results evaluate the performance using the example scenario in Figure 1. This helps in getting better understanding of how our approach works. The second part evaluates the scalability of the approach using the scenario in Figure 2. Both parts aim at evaluating the benefit of both multi-leveled policies and the dynamic learning rate.

For the small scenario in Figure 1, in each time step a task of type $TAB$ appear at agent $MF$ with probability 0.5 and at agent $MA$ with probability 0.5. Agent $MD$, the only agent who knows how to decompose $TAB$, does not receive any task directly. The cost of any task at a resource $R$ is $-10 \times load(R)^2$, where $load(R)$ is the number of tasks currently being serviced at resource $R$. When a resource fails to accomplish a task (e.g. when a resource of type $A$ is assigned a task of type $TB$), a penalty of -10000 is imposed as a cost. A task also fails if it reaches age 10 time units. The cost of communicating a task to a neighbor is -1. Tasks takes 5 time units to execute on resources of type $A$ or $B$ and only 3 time units to execute on either $A_f$ or $B_f$.

Figure 3 compares the performance of our algorithm for three settings of the learning rate $\delta$: dynamic between 0.0001 and 0.01, static at 0.01, and static at 0.0001. The horizontal axis is the time steps while the vertical axis is the absolute sum of incurred costs per 100 time steps, averaged over 10 simulation runs (lower is better). The static-at-0.0001 is too slow and it did not converge even after 10000 times steps. As expected, a larger static learning rate (0.01) leads to faster convergence. Using a dynamic learning rate strikes a balance by converging to a much better policy than static-at-0.01 (less than 25% of its cost) in much less time than the static-at-0.0001. Although there might be a static learning rate that achieves performance similar to that of the dynamic rate, it is much harder to fine tune the learning rate to a fixed value than to specify the range of the dynamic rate (we used $\delta = 0.0001$ and $\delta_{max} = 0.01$).

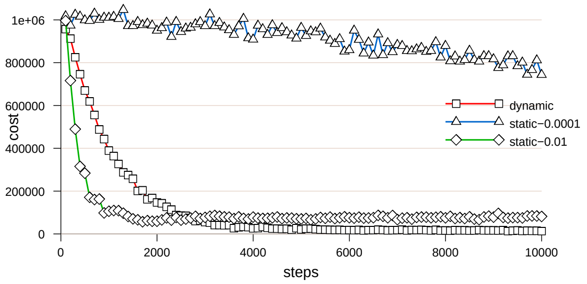


Figure 3: The effect of the dynamic learning rate.

Figure 4 compares the performance of our algorithm using four settings of the policies $\pi_{low}$ and $\pi_{high}$: both are deterministic (*deterministic*), only $\pi_{low}$ is stochastic (*low*), only $\pi_{high}$ is stochastic (*high*), and both are stochastic (*two-level*). As expected, *two-level* is the slowest to converge but achieves the lowest steady cost (about 80% of the second lowest steady cost, *low*). On the other hand, and to our surprise, *high* converges faster than *deterministic* (and achieves lower steady cost than *deterministic*, which is expected). The reason is that even without any learning, $\pi_{high}$ selects a decomposition uniformly at random. This slightly balances the load without paying the price of slow convergence due to learning a stochastic $\pi_{low}$.

Figure 5 illustrates the evolution of stochastic policies in agents $MD$ and $MA$ during a simulation run. The horizontal axis represents time steps. The vertical axis represents policies, i.e. the total 1.0 probability divided over actions (an

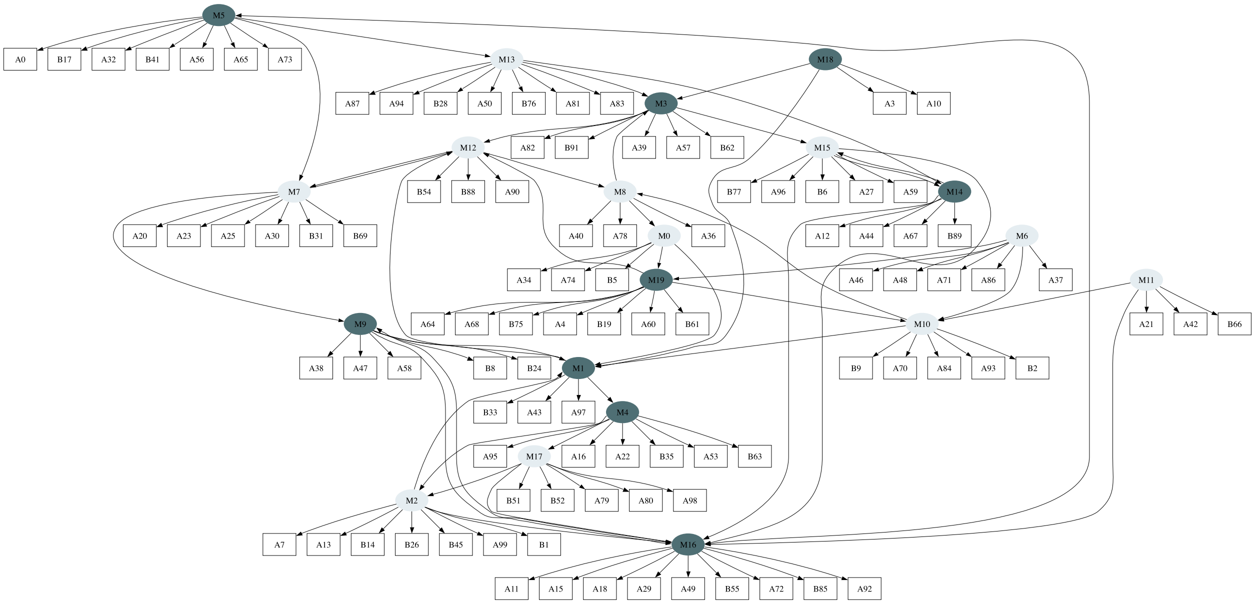


Figure 2: A large scale network of 100 resources and 20 agents.

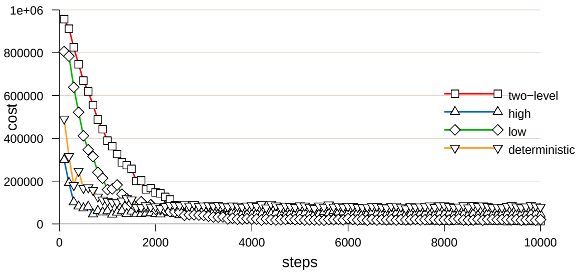


Figure 4: The effect of two level stochastic policies on performance.

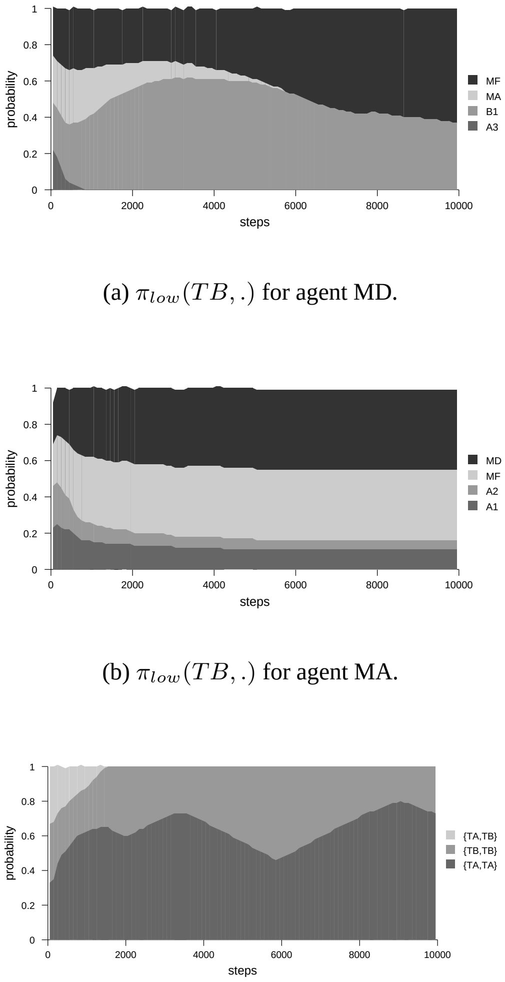
action is a neighbor in case of $\pi_{low}$ and a decomposition in case of $\pi_{high}$). Figure 5(a) shows $\pi_{low}$ of task $TB$ at agent $MD$. There are four possible assignments of $TB$, to each neighbor of $MD$. The probability of assigning $A3$, which is a resource of type $A$, quickly drops to zero as expected. Also since $MA$ is not directly controlling any resources of type $B$, the probability of $MD$ choosing $MA$ also drops to zero but after a while (about 6000 steps). The reasons are cycles and indirect links. Initially $MD$ may send a request for a task of type $TB$ to $MA$ who in turn either sends it to $MF$ or back to $MD$. However, using the simple maximum task age mechanism, eventually $MD$ learns to stop sending tasks of type $TB$ to $MA$. In the end, $MD$ only chooses among two assignments for $TB$: $B1$ and $MF$, with more probability of choosing $MF$. This what one would expect to balance the load: faster resources get more tasks.

Figure 5(b) shows $\pi_{low}(TB, .)$ for agent $MA$. After step 6000 we see the policy almost fixed. This is because $MA$ is not receiving any tasks of type $TB$ from agent $MD$, therefore it stopped learning about it. Figure 5(c) shows how $MD$ learns $\pi_{high}$ for different decompositions of task $TB$. Agent $MD$ quickly learns to drop decomposition $\{TA, TB\}$. The reason is that this decomposition requires equal numbers of resource types $A$ and $B$, while the system contains 4 $A$ resources and only 2 $B$ resources. $MD$ converges to an intuitive policy that produces more $TA$ tasks than $TB$ tasks.

The second part of the results show the scalability of our approach using the system in Figure 2. This system consists of 100 resources (rectangles) and 20 agents (ellipses). With probability 0.67 the resource is of type $A$, otherwise it of type $B$. Also with probability 0.67 the resource is normal, otherwise it is fast. Each agent has two neighboring agents picked randomly from the set of agents. Each resource is connected randomly to one of the agents. At each time step, tasks of type $TAB$ appear at 11 agents (light gray) with probability 0.5. The other 9 agents (dark gray) know how to decompose tasks of type $TAB$. Other parameters are the same as the small scenario. Therefore, the average number of $TAB$ tasks per 100 time steps is $0.5 \times 11 \times 100 = 550$, which requires (after decomposition) 1100 resources. A lower bound on the average cost, assuming perfect knowledge and perfect distribution of load, is 11000. The highest average cost (if all tasks allocated to the same resource) is Figures 6 and 7 show the performance of the different approaches in the larger system. We can see significant savings of our approach compared to the other approaches.

## 5 Related Work

In [Hannah and Mouaddib, 2002], a mediator serially allocates tasks to agents. That work used a Markov Decision Process (MDP) model where actions are agent-task assignments and learned a deterministic policy. This differs from our work where all subtasks are allocated concurrently and



(a) $\pi_{low}(TB,.)$ for agent MD.

(b) $\pi_{low}(TB,.)$ for agent MA.

(c) $\pi_{high}(TAB,.)$ for agent MD.

Figure 5: Policies of different agents.

using two-level stochastic policy. That work also assumed the set of tasks were *fixed* and arrived in fixed order, while we assume tasks arrive stochastically in time and location. They also assumed agents with homogeneous capabilities, while our model supports heterogeneous agents.

The work in [Dolgov and Durfee, 2004] modeled the resource allocation problem as a constrained MDP, or CMDP. A CMDP is an MDP augmented with a set of (resource) constraints. The set of actions were assumed fixed and the policy was serial and deterministic. They also used an offline algorithm which solved the problem assuming the transition probabilities are known. We use an on-line algorithm without sharing state information among agents.

Task allocation can be viewed as a more complex and more general form of packet routing. As in routing, each agent acts as a router, trying to route the packet through the least costly path. Packets impose little load on the nodes (resources) as opposed to tasks, which raises the issue of load balancing. Also task allocation involves alternative decompositions while packets are routed as non-decomposable units. Most of the previous work in packet routing [Boyan and Littman, 1994; Kumar and Miikkulainen, 1998] maps the routing problem to a set of local decision problems for each agent. The work used reinforcement learning techniques to learn a deterministic policy for each router. The goal was to minimize average packet delay. Experimental results showed the effectiveness of the approach. More recently, a policy gradient approach was used to solve the packet routing problem [Tao *et al.*, 2001]. However, the work ignored the load on the nodes and only focused on the capacity of links. Their policy gradient also used a fixed learning rate, unlike the algorithm presented here.

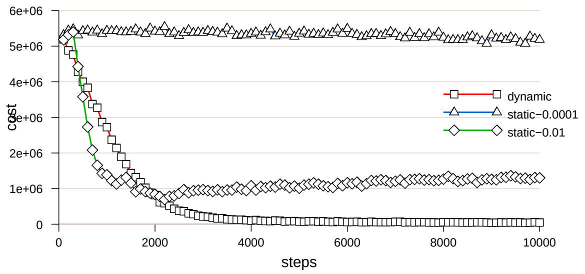


Figure 6: The effect of dynamic learning rate in the large system scenario.

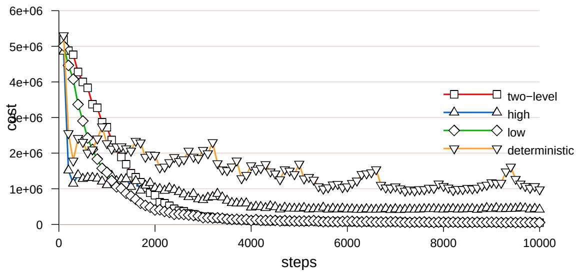


Figure 7: The effect of two level policies in the large system scenario.

## 6 Conclusion and Future Work

This paper presents a novel algorithm that allows agents in a network to learn cooperatively how to allocate a task. The algorithm learns two-level stochastic policies using policy gradient. The high level policy selects a decomposition for an incoming task while the low level policy assigns a neighboring agent to each task in the selected decomposition. Experimental results show the benefit of introducing each of these levels with more than four times saving in cost as compared to deterministic approaches. Our algorithm also dynamically adjusts the learning rate. Experimental results show how using a dynamic learning rate significantly speeds up convergence while outperforming learners with fixed learning rate.

An interesting issue that was not covered in the paper is how to set up the network connections, i.e. the neighborhood of each agent $N$. Optimally, the network should reduce communication overhead by adapting to task arrival patterns. For example, an agent that receives many tasks asking for resource $R_x$ should be connected as closely as possible to resources of that type. A related issue is how the system would perform in the face of changes in the network (e.g. an agent or a resource leaving the system or another agent or a resource entering.) Furthermore, this work models resource failures implicitly using penalties. An explicit model of failure probability may allow agents to learn better policies (e.g. preferring an agent with high probability of failure if its cost is cheap and the task payoff is low, or vice versa).

## References

[Boyan and Littman, 1994] Justin A. Boyan and Michael L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 671–678. Morgan Kaufmann Publishers, Inc., 1994.

[Czajkowski and et al, 2001] K. Czajkowski and et al. Grid information services for distributed resource sharing. *Proceedings of the 10th IEEE Symp On High Performance Distributed Computing*, 2001.

[Dolgov and Durfee, 2004] Dmitri Dolgov and Edmund Durfee. Optimal resource allocation and policy formulation in loosely-coupled markov decision processes. In *In Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling.*, 2004.

[Hannah and Mouaddib, 2002] Hosam Hannah and Abdel-Illah Mouaddib. Task selection problem under uncertainty as decision-making. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, 2002.

[Kumar and Miikkulainen, 1998] Shailesh Kumar and Risto Miikkulainen. Confidence-based q-routing: An on-line adaptive network routing algorithm. In *Proceedings of Artificial Neural Networks in Engineering*, 1998.

[Michael Bowling, 2002] Manuela Veloso Michael Bowling. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215–250, 2002.

[Sutton and Barto, 1999] R Sutton and A Barto. *Reinforcment Learning: An Introduction*. MIT Press, 1999.

[Tao *et al.*, 2001] Nigel Tao, Jonathan Baxter, and Lex Weaver. A multi-agent, policy-gradient approach to network routing. In *Proc. 18th International Conf. on Machine Learning*, pages 553–560. Morgan Kaufmann, San Francisco, CA, 2001.

# An Algorithm Better than AO*?

**Blai Bonet**
Departamento de Computación
Universidad Simón Bolívar
Caracas, Venezuela
bonet@ldc.usb.ve

**Héctor Geffner**
ICREA & Universitat Pompeu Fabra
Paseo de Circunvalación, 8
Barcelona, Spain
hector.geffner@upf.edu

## Abstract

Recently there has been a renewed interest in AO* as planning problems involving uncertainty and feedback can be naturally formulated as AND/OR graphs. In this work, we carry out what is probably the first detailed empirical evaluation of AO* in relation to other AND/OR search algorithms. We compare AO* with two other methods: the well-known Value Iteration (VI) algorithm, and a new algorithm, Learning in Depth-First Search (LDFS). We consider instances from four domains, use three different heuristic functions, and focus on the optimization of cost in the worst case (Max AND/OR graphs). Roughly we find that while AO* does better than VI in the presence of informed heuristics, VI does better than recent extensions of AO* in the presence of cycles in the AND/OR graph. At the same time, LDFS and its variant Bounded LDFS, which can be regarded as extensions of IDA*, are almost never slower than either AO* or VI, and in many cases, are orders-of-magnitude faster.

## 1 Introduction

A* and AO* are the two classical heuristic best-first algorithms for searching OR and AND/OR graphs [Hart *et al.*, 1968; Martelli and Montanari, 1973; Pearl, 1983]. The A* algorithm is taught in every AI class, and has been studied thoroughly both theoretically and empirically. The AO* algorithm, on the other hand, has found less uses in AI, and while prominent in early AI texts [Nilsson, 1980] it has disappeared from current ones [Russell and Norvig, 1994]. In the last few years, however, there has been a renewed interest in the AO* algorithm in planning research where problems involving uncertainty and feedback can be formulated as search problems over AND/OR graphs [Bonet and Geffner, 2000].

In this work, we carry out what is probably the first in-depth empirical evaluation of AO* in relation with other AND/OR graph search algorithms. We compare AO* with an old but general algorithm, *Value Iteration* [Bellman, 1957; Bertsekas, 1995], and a new algorithm, *Learning in Depth-First Search*, and its variant Bounded LDFS [Bonet and Geffner, 2005]. While VI performs a sequence of Bellman updates over all states in parallel until convergence, LDFS performs selective Bellman updates on top of successive depth-first searches, very much as Learning RTA* [Korf, 1990] and RTDP [Barto *et al.*, 1995] perform Bellman updates on top of successive greedy (real-time) searches.

In the absence of accepted benchmarks for evaluating AND/OR graph search algorithms, we introduce four parametric domains, and consider a large number of instances, some involving millions of states. In all cases we focus on the computation of solutions with minimum cost in the worst case using three different and general admissible heuristic functions. We find roughly that while AO* does better than VI in the presence of informed heuristics, LDFS, with or without heuristics, tends to do better than both.

AO* is limited to handling AND/OR graphs without cycles. The difficulties arising from cycles can be illustrated by means of a simple graph with two states and two actions: an action $a$ with cost $5$ maps the initial state $s_0$ non-deterministically into either a goal state $s_G$ or $s_0$ itself, and a second action $b$ with cost $10$ maps $s_0$ deterministically into $s_G$. Clearly, the problem has cost $10$ and $b$ is the only (optimal) solution, yet the simple cost revision step in AO* does not yield this result. Thus, for domains where such cycles appear, we evaluate a recent variant of AO*, $\text{CFC}_{rev^*}$, introduced in [Jimenéz and Torras, 2000] that is not affected by this problem. We could have used LAO* as well [Hansen and Zilberstein, 2001], but this would be an overkill as LAO* is designed to minimize expected cost in probabilistic AND/OR graphs (MDPs) where *solutions themselves* can be cyclic, something that cannot occur in Additive or Max AND/OR graphs. Further algorithms for cyclic graphs are discussed in [Mahanti *et al.*, 2003]. LDFS has no limitations of this type; unlike AO*, it is not affected by the presence of cycles in the graph, and unlike Value Iteration, it is not affected either by the presence of dead-ends in the state space if the problem is solvable.

The paper is organized as follows: we consider first the models, then the algorithms, the experimental set up and the results, and close with a brief discussion.

## 2 Models

We consider AND/OR graphs that arise from non-deterministic state models as those used in planning with non-determinism and full observability, where there are

1. a discrete and finite state space $S$,
2. an initial state $s_0 \in S$,
3. a non-empty set of terminal states $S_T \subseteq S$,
4. actions $A(s) \subseteq A$ applicable in each non-terminal state,

5. a function mapping non-terminal states $s$ and actions $a \in A(s)$ into *sets* of states $F(a, s) \subseteq S$,
6. action costs $c(a, s)$ for non-terminal states $s$, and
7. terminal costs $c_T(s)$ for terminal states.

Models where the states are only *partially observable*, can be described in similar terms, replacing states by *sets of states* or *belief states* [Bonet and Geffner, 2000].

We assume that both $A(s)$ and $F(a, s)$ are non-empty, that action costs $c(a, s)$ are all positive, and terminal costs $c_T(s)$ are non-negative. When terminal costs are all zero, terminal states are called *goals*.

The mapping from non-deterministic state models to AND/OR graphs is immediate: non-terminal states $s$ become OR nodes, connected to the AND nodes $< s, a >$ for each $a \in A(s)$, whose children are the states $s' \in F(a, s)$. The inverse mapping is also direct.

The solutions to this and various other state models can be expressed in terms of the so-called Bellman equation that characterizes the *optimal cost function* [Bellman, 1957; Bertsekas, 1995]:

$$V(s) \stackrel{\text{def}}{=} \begin{cases} c_T(s) & \text{if } s \text{ terminal} \\ \min_{a \in A(s)} Q_V(a, s) & \text{otherwise} \end{cases} \quad (1)$$

where $Q_V(a, s)$ is an abbreviation of the cost-to-go, which for Max and Additive AND/OR graphs takes the form:

$$Q_V(a, s) \ : \begin{cases} c(a, s) + \max_{s' \in F(a,s)} V(s') & \text{(Max)} \\ c(a, s) + \sum_{s' \in F(a,s)} V(s') & \text{(Add)} \end{cases} \quad (2)$$

Other models can be handled in this way by choosing other forms for $Q_V(a, s)$. For example, for MDPs, it is the weighted sum $c(a, s) + \sum_{s' \in F(a,s)} V(s')P_a(s'|s)$ where $P_a(s'|s)$ is the probability of going from $s$ to $s'$ given $a$.

In the absence of dead-ends, there is a unique (optimal) value function $V^*(s)$ that solves the Bellman equation, and the optimal solutions can be expressed in terms of the policies $\pi$ that are *greedy* with respect to $V^*(s)$. A policy $\pi$ is a function mapping states $s \in S$ into actions $a \in A(s)$, and a policy $\pi_V$ is greedy with respect to a value function $V(s)$, or simply greedy in $V$, iff $\pi_V$ is the best policy assuming that the cost-to-go is given by $V(s)$; i.e.

$$\pi_V(s) \ = \ \operatorname*{argmin}_{a \in A(s)} Q_V(a, s)\,. \quad (3)$$

Since the initial state $s_0$ is known, it is actually sufficient to consider *closed (partial) policies* $\pi$ that prescribe the actions to do in all (non-terminal) states reachable from $s_0$ and $\pi$. Any closed policy $\pi$ relative to a state $s$ has a cost $V^\pi(s)$ that expresses the cost of solving the problem starting from $s$. The costs $V^\pi(s)$ are given by the solution of (1) but with the operator $\min_{a \in A(s)}$ removed and the action $a$ replaced by $\pi(s)$. These costs are well-defined when the resulting equations have a solution over the subset of states reachable from $s_0$ and $\pi$. For Max and Additive AND/OR graphs, this happens when $\pi$ is *acyclic*; else $V^\pi(s_0) = \infty$. When $\pi$ is acyclic, the costs $V^\pi(s_0)$ can be defined recursively starting with the terminal states $s'$ for which $V^\pi(s') = c_T(s')$, and up to the non-terminal states $s$ reachable from $s_0$ and $\pi$ for which $V^\pi(s) = Q_{V^\pi}(\pi(s), s)$. In all cases, we are interested in computing a solution $\pi$ that minimizes $V^\pi(s_0)$. The resulting value is the optimal cost of the problem $V^*(s_0)$.

# 3 Algorithms

We consider three algorithms for computing such optimal solutions for AND/OR graphs: Value Iteration, AO*, and Learning in Depth-First Search.

## 3.1 Value Iteration

Value iteration is a simple and quite effective algorithm that computes the fixed point $V^*(s)$ of Bellman equation by plugging an estimate value function $V_i(s)$ in the right-hand side and obtaining a new estimate $V_{i+1}(s)$ on the left-hand side, iterating until $V_i(s) = V_{i+1}(s)$ for all $s \in S$ [Bellman, 1957]. In our setting, this convergence is guaranteed provided that there are no dead-end states, i.e., states $s$ for which $V^*(s) = \infty$. Often convergence is accelerated if the same value function vector $V(s)$ is used on both left and right. In such a case, in each iteration, the states values are *updated* sequentially from first to last as:

$$V(s) \ := \ \min_{a \in A(s)} Q_V(a, s)\,. \quad (4)$$

The iterations continue until $V$ satisfies the Bellman equation, and hence $V = V^*$. Any policy $\pi$ greedy in $V^*$ provides then an optimal solution to the problem. VI can deal with a variety of models and is very easy to implement.

## 3.2 AO*

AO* is a best-first algorithm for solving acyclic AND/OR graphs [Martelli and Montanari, 1973; Nilsson, 1980; Pearl, 1983]. Starting with a partial graph $G$ containing only the initial state $s_0$, two operations are performed iteratively: first, a best partial policy over $G$ is constructed and a non-terminal tip state $s$ reachable with this policy is expanded; second, the value function and best policy over the updated graph are incrementally recomputed. This process continues until the best partial policy is complete. The second step, called the *cost revision step*, exploits the acyclicity of the AND/OR graph for recomputing the optimal costs and policy over the partial graph $G$ *in a single pass*, unlike Value Iteration (yet see [Hansen and Zilberstein, 2001]). In this computation, the states outside $G$ are regarded as terminal states with costs given by their heuristic values. When the AND/OR graph contains cycles, however, this basic cost-revision operation is not adequate. In this paper, we use the AO* variant developed in [Jiménez and Torras, 2000], called CFC$_{rev^*}$, which is based in the cost revision operation from [Chakrabarti, 1994] and is able to handle cycles.

Unlike VI, AO* can solve AND/OR graphs without having to consider the entire state space, and exploits lower bounds for focusing the search. Still, expanding the partial graph one state at a time, and recomputing the best policy over the graph after each step, imposes an overhead that, as we will see, does not always appear to pay off.

## 3.3 Learning DFS

LDFS is an algorithm akin to IDA* with transposition tables which applies to a variety of models [Bonet and Geffner, 2005]. While IDA* consists of a sequence of DFS iterations that backtrack upon encountering states with costs exceeding a given bound, LDFS consists of a sequence of DFS iterations that backtrack upon encountering states that are *inconsistent*: namely states $s$ whose values are not consistent with the values of its children; i.e. $V(s) \neq \min_{a \in A(s)} Q_V(a, s)$. The

```
LDFS-DRIVER($s_0$)
begin
    repeat $solved$ := LDFS($s_0$) until $solved$
    return $(V, \pi)$
end

LDFS($s$)
begin
    if $s$ is SOLVED or terminal then
        if $s$ is terminal then $V(s) := c_T(s)$
        Mark $s$ as SOLVED
        return $true$

    $flag := false$
    foreach $a \in A(s)$ do
        if $Q_V(a, s) > V(s)$ then continue
        $flag := true$
        foreach $s' \in F(a, s)$ do
            $flag :=$ LDFS($s'$) & $[Q_V(a, s) \leq V(s)]$
            if $\neg flag$ then break
        if $flag$ then break

    if $flag$ then
        $\pi(s) := a$
        Mark $s$ as SOLVED
    else
        $V(s) := \min_{a \in A(s)} Q_V(a, s)$

    return $flag$
end
```

Algorithm 1: Learning DFS

expression $Q_V(a, s)$ encodes the type of model: OR graphs, Additive or Max AND/OR graphs, MDPs, etc. Upon encountering such inconsistent states, LDFS updates their values (making them consistent) and backtracks, updating along the way ancestor states as well. In addition, when the DFS beneath a state $s$ does not find an inconsistent state (a condition kept by $flag$ in Fig. 1), $s$ is labeled as *solved* and is not expanded again. The DFS iterations terminate when the initial state $s_0$ is solved. Provided the initial value function is admissible and monotonic (i.e., $V(s) \leq \min_{a \in A(s)} Q_V(a, s)$ for all $s$), LDFS returns an optimal policy if one exists. The code for LDFS is quite simple and similar to IDA* [Reinefeld and Marsland, 1994]; see Fig. 1.

Bounded LDFS, shown in Fig. 2, is a slight variation of LDFS that accommodates an explicit $bound$ parameter for focusing the search further on paths that are 'critical' in the presence of Max rather than Additive models. For Game Trees, Bounded LDFS reduces to the state-of-the-art MTD$(-\infty)$ algorithm: an iterative alpha-beta search procedure with null windows and memory [Plaat *et al.*, 1996]. The code in Fig. 2, unlike the code in [Bonet and Geffner, 2005] is for general Max AND/OR graphs and not only trees, and replaces the boolean SOLVED$(s)$ tag in LDFS by a numerical tag $U(s)$ that stands for an *upper bound*; i.e., $U(s) \geq V^*(s) \geq V(s)$. This change is needed because Bounded LDFS, unlike LDFS, minimizes $V^\pi(s_0)$ but not necessarily $V^\pi(s)$ for all states $s$ reachable from $s_0$ and $\pi$ (in Additive models, the first condition implies the second). Thus, while the SOLVED$(s)$ tag in LDFS means that an optimal policy for $s$ has been found, the $U(s)$ tag in Bounded LDFS means only that a policy $\pi$ with cost $V^\pi(s) = U(s)$ has been found. Bounded LDFS ends however when the lower and upper bounds for $s_0$ coincide. The upper bounds $U(s)$ are initialized to $\infty$. The code in Fig. 2 is for Max AND/OR graphs; for Additive graphs, the term $\sum_{s''} V(s'')$ needs to be subtracted from the right-hand side of line $nb := bound - c(a, s)$ for $s''$ in $F(a, s)$ and $s'' \neq s'$. The resulting procedure however is equivalent to LDFS.

```
B-LDFS-DRIVER($s_0$)
begin
    repeat B-LDFS($s_0, V(s_0)$) until $V(s_0) \geq U(s_0)$
    return $(V, \pi)$
end

B-LDFS($s, bound$)
begin
    if $s$ is terminal or $V(s) \geq bound$ then
        if $s$ is terminal then $V(s) := U(s) := c_T(s)$
        return $true$

    $flag := false$
    foreach $a \in A(s)$ do
        if $Q_V(a, s) > bound$ then continue
        $flag := true$
        foreach $s' \in F(a, s)$ do
            $nb := bound - c(a, s)$
            $flag :=$ B-LDFS($s', nb$) & $[Q_V(a, s) \leq bound]$
            if $\neg flag$ then break
        if $flag$ then break

    if $flag$ then
        $\pi(s) := a$
        $U(s) := bound$
    else
        $V(s) := \min_{a \in A(s)} Q_V(a, s)$

    return $flag$
end
```

Algorithm 2: Bounded LDFS for Max AND/OR Graphs

## 4 Experiments

We implemented all algorithms in C++. Our AO* code is a careful implementation of the algorithm in [Nilsson, 1980], while our CFC$_{rev^*}$ code is a modification of the code obtained from the authors [Jimenéz and Torras, 2000] that makes it roughly an order-of-magnitude faster.

For all algorithms we initialize the values of the terminal states to their true values $V(s) = c_T(s)$ and non-terminals to some *heuristic values* $h(s)$ where $h$ is an admissible and monotone heuristic function. We consider three such heuristics: the first, the non-informative $h = 0$, and then two functions $h_1$ and $h_2$ that stand for the value functions that result from performing $n$ iterations of value iteration, and an equivalent number of 'random' state updates respectively,[1] starting with $V(s) = 0$ at non-terminals. In all the experiments, we set $n$ to $N_{vi}/2$ where $N_{vi}$ is the number of iterations that value iteration takes to converge. These heuristics are informative but expensive to compute, yet we use them for as-

[1] More precisely, the random updates are done by looping over the states $s \in S$, selecting and updating states $s$ with probability $1/2$ til $n \times |S|$ updates are made.

| problem | $\lvert S\rvert$ | $V^*$ | $N_{\text{VI}}$ | $\lvert A\rvert$ | $\lvert F\rvert$ | $\lvert\pi^*\rvert$ |
|---|---|---|---|---|---|---|
| coins-10 | 43 | 3 | 2 | 172 | 3 | 9 |
| coins-60 | 1,018 | 5 | 2 | 315K | 3 | 12 |
| mts-5 | 625 | 17 | 14 | 4 | 4 | 156 |
| mts-35 | $1,5$M | 573 | 322 | 4 | 4 | 220K |
| mts-40 | $2,5$M | 684 | – | 4 | 4 | 304K |
| diag-60-10 | 29,738 | 6 | 8 | 10 | 2 | 119 |
| diag-60-28 | $>$ 15M | 6 | – | 28 | 2 | 119 |
| rules-5000 | 5,000 | 156 | 158 | 50 | 50 | 4,917 |
| rules-20000 | 20,000 | 592 | 594 | 50 | 50 | 19,889 |

Table 1: Data for smallest and largest instances: number of (reachable) belief states, optimal cost, number of iterations taken by VI, max branching in OR nodes ($|A|$) and AND nodes ($|F|$), and size of optimal solution ($M = 10^6$; $K = 10^3$).

sessing how well the various algorithms are able to exploit heuristic information. The times for computing the heuristics are common to all algorithms and are not included in the runtimes.

We are interested in *minimizing cost in the worst case* (Max AND/OR graphs). Some relevant features of the instances considered are summarized in Table 1. A brief description of the domains follows.

**Coins:**
There are $N$ coins including a counterfeit coin that is either lighter or heavier than the others, and a 2-pan balance. A strategy is needed for identifying the counterfeit coin, and whether it is heavier or lighter than the others [Pearl, 1983]. We experiment with $N = 10, 20, \ldots, 60$. In order to reduce symmetries we use the representation from [Fuxi *et al.*, 2003] where a (belief) state is a tuple of non-negative integers $(s, ls, hs, u)$ that add up to $N$ and stand for the number of coins that are known to be of standard weight ($s$), standard or lighter weight ($ls$), standard or heavier weight ($hs$), and completely unknown weight ($u$). See [Fuxi *et al.*, 2003] for details.

**Diagnosis:**
There are $N$ binary tests for finding out the true state of a system among $M$ different states [Pattipati and Alexandridis, 1990]. An instance is described by a binary matrix $T$ of size $M \times N$ such that $T_{ij} = 1$ iff test $j$ is positive when the state is $i$. The goal is to obtain a strategy for identifying the true state. The search space consists of all non-empty subsets of states, and the actions are the tests. Solvable instances can be generated by requiring that no two rows in $T$ are equal, and $N > \log_2(M)$ [Garey, 1972]. We performed two classes of experiments: a first class with $N$ fixed to 10 and $M$ varying in $\{10, 20, \ldots, 60\}$, and a second class with $M$ fixed to 60 and $N$ varying in $\{10, 12, \ldots, 28\}$. In each case, we report average runtimes and standard deviations over 5 random instances.

**Rules:**
We consider the derivation of atoms in acyclic rule systems with $N$ atoms, and at most $R$ rules per atom, and $M$ atoms per rule body. In the experiments $R = M = 50$ and $N$ is in $\{5000, 10000, \ldots, 20000\}$. For each value of $N$, we report average times and standard deviations over 5 random solvable instances.

**Moving Target Search:**
A predator must catch a prey that moves non-deterministically to a non-blocked adjacent cell in a given random maze of size $N \times N$. At each time, the predator and prey move one position. Initially, the predator is in the upper left position and the prey in the bottom right position. The task is to obtain an optimal strategy for catching the prey. In [Ishida and Korf, 1995], a similar problem is considered in a real-time setting where the predator moves 'faster' than the prey, and no optimality requirements are made. Solvable instances are generated by ensuring that the undirected graph underlying the maze is connected and loop free. Such loop-free mazes can be generated by performing random Depth-First traversals of the $N \times N$ empty grid, inserting 'walls' when loops are encountered. We consider $N = 15, 20, \ldots, 40$, and in each case report average times and standard deviations over 5 random instances. Since the resulting AND/OR graphs involve cycles, the algorithm CFC$_{rev^*}$ is used instead of AO*.

## 4.1 Results

The results of the experiments are shown in Fig. 2, along with a detailed explanation of the data. Each square depicts the runtimes in seconds for a given domain and heuristic in a logarithmic scale. The figure also includes data from another learning algorithm, a Labeled version of Min-Max LRTA* [Koenig, 2001]. Min-Max LRTA* is an extension of Korf's LRTA* [Korf, 1990] and, at the same time, the Min-Max variant of RTDP [Barto *et al.*, 1995]. Labeled RTDP and Labeled Min-Max LRTA* are extensions of RTDP and Min-Max LRTA* [Bonet and Geffner, 2003] that speed up convergence and provide a crisp termination condition by keeping track of the states that are solved.

The domains from top to bottom are COINS, DIAGNOSIS 1 and 2, RULES, and MTS, while the heuristics from left to right are $h = 0$, $h_1$, and $h_2$. As mentioned above, MTS involves cycles, and thus, CFC$_{rev^*}$ is used instead of AO*. Thus leaving this domain aside for a moment, we can see that with the two (informed) heuristics, AO* does better than VI in almost all cases, with the exception of COINS with $h_1$ where VI beats all algorithms by a small margin. Indeed, as it can be seen in Table 1, VI happens to solve COINS in very few iterations (this actually has to do with a topological sort done in our implementation of VI for finding first the states that are reachable). In DIAGNOSIS and in COINS with $h_1$, AO* runs one or more orders of magnitude faster than VI. With $h = 0$, the results are mixed, with VI doing better, and in certain cases (DIAGNOSIS) much better. Adding now LDFS to the picture, we see that it is never worse than either AO* or VI, except in COINS with $h = 0$ and $h_2$, and RULES with $h = 0$ where it is slower than VI and AO* respectively by a small factor (in the latter case 2). In most cases, however, LDFS runs faster than both AO* and VI for the different heuristics, in several of them by one or more orders of magnitude. Bounded LDFS in turn does never worse than LDFS, and in a few cases, including DIAGNOSIS with $h = 0$, runs an order of magnitude faster. In MTS, a problem which involves cycles in the AND/OR graph, AO* cannot be used, CFC$_{rev^*}$ solves only the smallest problem, and VI solves all but the largest problem, an order of magnitude slower than LDFS, which in turn is slower than Bounded LDFS. Finally, Min-Max LRTA* is never worse than AO*, performs similar to LDFS and Bounded LDFS except in DIAGNOSIS and COINS where Bounded LDFS dominates all algorithms, and in RULES where Min-Max LRTA* dominates all algorithms.

The difference in performance between VI and the other algorithms for $h \neq 0$ suggests that the latter make better use of the initial heuristic values. At the same time, the difference between LDFS and AO* suggests that often the overhead involved in expanding the partial graph one state at a time, and recomputing the best policy over the graph after each step, does not always pay off.[2] LDFS makes use of the heuristic information but makes no such (best-first) commitments. Last, the difference in performance between LDFS and Bounded LDFS can be traced to a theoretical property mentioned above and discussed in further detail in [Bonet and Geffner, 2005]: while LDFS (and AO* and VI) compute policies $\pi$ that are optimal over all the states reachable from $s_0$ and $\pi$, Bounded LDFS computes policies $\pi$ that are optimal only where needed; i.e. in $s_0$. For OR and Additive AND/OR graphs, the latter notion implies the former, but for Max models does not. Bounded LDFS (and Game Tree algorithms) exploits this distinction, while LDFS, AO*, and Value Iteration do not.

## 5 Discussion

We have carried an empirical evaluation of AND/OR search algorithms over a wide variety of instances, using three heuristics, and focusing in the optimization of cost in the worst case (Max AND/OR graphs). Over these examples and with these heuristics, the studied algorithms rank from fastest to slowest as Bounded LDFS, LDFS, AO*, and VI, with some small variations.The results for Min-Max LRTA* show that its performance is similar to LDFS but inferior to Bounded LDFS except in RULES.

We have considered the solution of Max AND/OR graphs as it relates well to problems in planning where one aims to minimize cost in the worst case. Additive AND/OR graphs, on the other hand, do not provide a meaningful cost criteria for the problems considered, as in the presence of common subproblems they count repeated solution subgraphs multiple times. The semantics of Max AND/OR graphs does not have this problem. Still we have done preliminary tests under the Additive semantics to find out whether the results change substantially or not. Interestingly, in some domains like diagnosis, the results do not change much, but in others, like RULES they do,[3] making indeed AO* way better than LDFS and VI, and suggesting, perhaps not surprisingly, that the effective solution of Additive and Max AND/OR graphs may require different ideas in each case. In any case, by making the various problems and source codes available, we hope to encourage the necessary experimentation that has been lacking so far in the area.

## Acknowledgements

We thank P. Jimenez and C. Torras for making the code of $\text{CFC}_{rev^*}$ available, and A. Frangi for the use of the Hermes Computing Resource at the Aragon Inst. of Engr. Research (I3A), U. of Zaragoza. H. Geffner is partially supported by grant TIC2002-04470-C03-02 from MCyT/Spain, and B. Bonet by grant DI-CAI-001-04 USB/Venezuela.

[2] A similar observation appears in [Hansen and Zilberstein, 2001].

[3] Note that due to the common subproblems, the algorithms would *not* minimize the number of rules in the derivations.

## References

[Barto *et al.*, 1995] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[Bellman, 1957] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Bertsekas, 1995] D. Bertsekas. *Dynamic Programming and Optimal Control, (2 Vols)*. Athena Scientific, 1995.

[Bonet and Geffner, 2000] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In S. Chien, S. Kambhampati, and C. Knoblock, editors, *Proc. 6th International Conf. on Artificial Intelligence Planning and Scheduling*, pages 52–61, Breckenridge, CO, 2000. AAAI Press.

[Bonet and Geffner, 2003] B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In E. Giunchiglia, N. Muscettola, and D. Nau, editors, *Proc. 13th International Conf. on Automated Planning and Scheduling*, pages 12–21, Trento, Italy, 2003. AAAI Press.

[Bonet and Geffner, 2005] B. Bonet and H. Geffner. Learning in DFS: A unified approach to heuristic search in deterministic, non-deterministic, probabilistic, and game tree settings. 2005.

[Chakrabarti, 1994] P. P. Chakrabarti. Algorithms for searching explicit AND/OR graphs and their applications to problem reduction search. *Artificial Intelligence*, 65(2):329–345, 1994.

[Fuxi *et al.*, 2003] Z. Fuxi, T. Ming, and H. Yanxiang. A solution to billiard balls puzzle using AO* algorithm and its application to product development. In V. Palade, R. Howlett, and L. Jain, editors, *Proc. 7th International Conf. on Knowledge-Based Intelligent Information & Engineering Systems*, pages 1015–1022. Springer, 2003.

[Garey, 1972] M. Garey. Optimal binary identification procedures. *SIAM Journal on Applied Mathematics*, 23(2):173–186, 1972.

[Hansen and Zilberstein, 2001] E. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129:35–62, 2001.

[Hart *et al.*, 1968] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4:100–107, 1968.

[Ishida and Korf, 1995] T. Ishida and R. Korf. Moving-target search: A real-time search for changing goals. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 17:609–619, 1995.

[Jimenéz and Torras, 2000] P. Jimenéz and C. Torras. An efficient algorithm for searching implicit AND/OR graphs with cycles. *Artificial Intelligence*, 124:1–30, 2000.

[Koenig, 2001] S. Koenig. Minimax real-time heuristic search. *Artificial Intelligence*, 129:165–197, 2001.

[Korf, 1990] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2–3):189–211, 1990.

[Mahanti *et al.*, 2003] A. Mahanti, S. Ghose, and S. K. Sadhukhan. A framework for searching AND/OR graphs with cycles. *CoRR*, cs.AI/0305001, 2003.

[Martelli and Montanari, 1973] A. Martelli and U. Montanari. Additive AND/OR graphs. In N. Nilsson, editor, *Proc. 3rd International Joint Conf. on Artificial Intelligence*, pages 1–11, Palo Alto, CA, 1973. William Kaufmann.

[Nilsson, 1980] N. Nilsson. *Principles of Artificial Intelligence*. Tioga, 1980.

[Pattipati and Alexandridis, 1990] K. Pattipati and M. Alexandridis. Applications of heuristic search and information theory to sequential fault diagnosis. *IEEE Trans. System, Man and Cybernetics*, 20:872–887, 1990.

[Pearl, 1983] J. Pearl. *Heuristics*. Morgan Kaufmann, 1983.

[Plaat *et al.*, 1996] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1-2):255–293, 1996.

[Reinefeld and Marsland, 1994] A. Reinefeld and T. Marsland. Enhanced iterative-deepening search. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[Russell and Norvig, 1994] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1994.

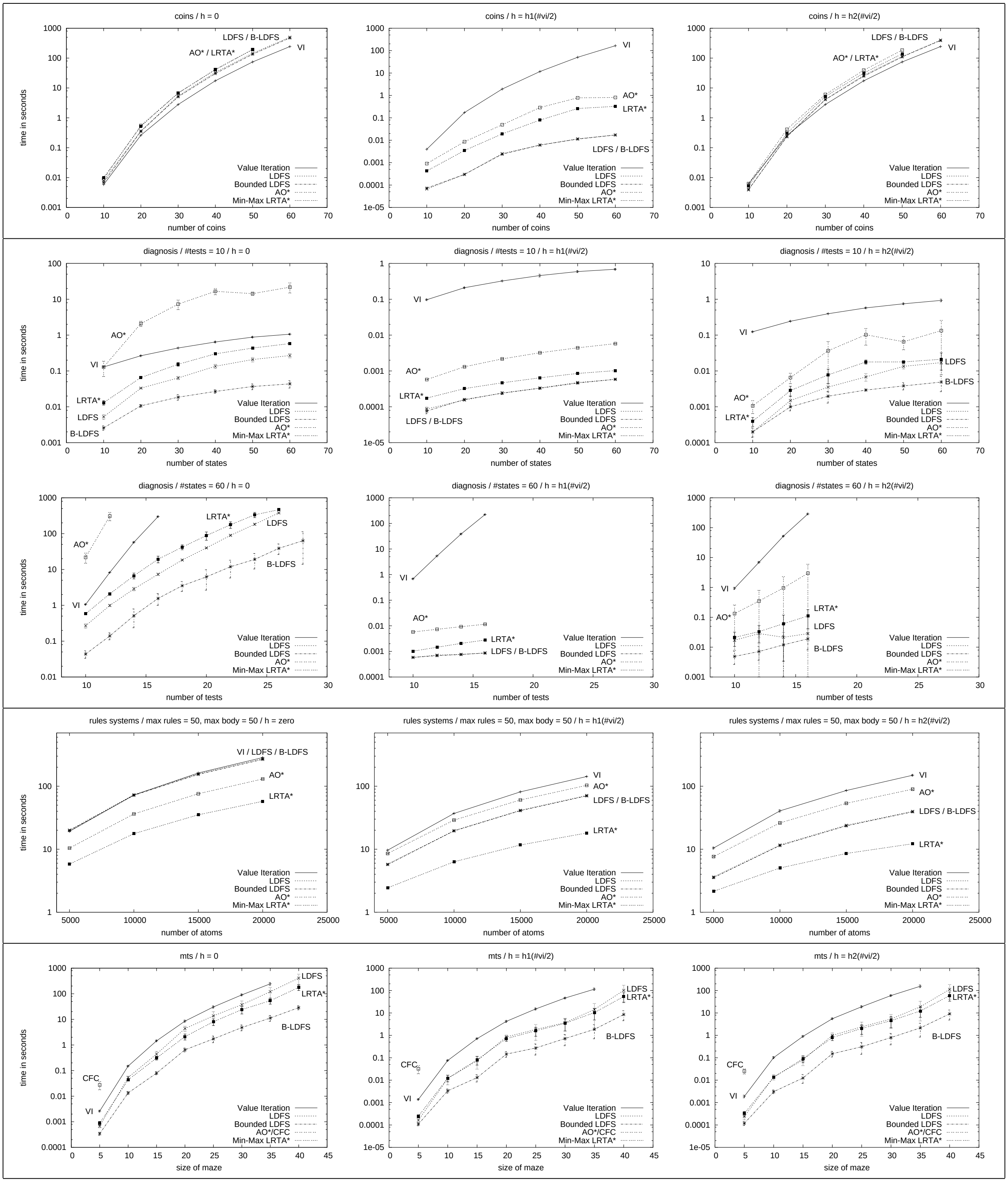


Table 2: Experiments: each square depicts runtimes in seconds for problems with a given domain and heuristic. The domains are from top to bottom: COINS, DIAGNOSIS 1 and 2, RULES, and MTS, and the heuristics from left to right: $h = 0$, $h_1$, and $h_2$. In the first diagnosis domain, the number of states is increased, while in the second, the number of tests. Problems with more than 16 tests are not solved for $h_1$ and $h_2$ as these heuristics could not be computed beyond that point. Such problems are solved by LDFS and Bounded LDFS with $h = 0$. All runtimes are shown in logarithmic scales, yet the range of scales vary.

# Hypothetical Planning

**Tamara Babaian**
CIS Department, Bentley College,
Waltham, MA 02452
tbabaian@bentley.edu

## Abstract

We present a novel method for interleaving planning with execution, called iterative deepening in hypotheticals. The method consists of performing an iterative deepening search in the space of partially ordered hypothetical plans. Hypothetical plans are partial plans in which the achievement of an otherwise unachievable goal may be conditioned on certain outcomes of sensing. This approach has been implemented within the PSIPLAN-S framework and used in a collaborative bibliography assistant, called Writer's Aid.

## 1 Introduction and Motivation

A planning agent operating in a real world must often deal with domains in which only incomplete information about the domain is available and furthermore, the complete information can never be acquired due to the large number of domain individuals. In such environments, using sensing actions judiciously and effectively to discover information that is relevant, but yet unknown, becomes critical.

Given correct, but incomplete description of the initial situation, a solution plan that *provably* achieves the goal may not exist. However, it may possible to construct the solution by interleaving the process of planning with execution of some information gathering steps. The method that we describe in this paper, called *hypothetical planning*, provides a mechanism for enacting such interleaved planning with execution. It is formulated using entailment and reasoning about knowledge and ignorance and guarantees non-redundancy of information gathering in that sensing actions are carried out only when the critical information is missing.

Hypothetical plans hypothesize on the value of an unknown subgoal; by verifying a hypothesis via execution of a sensing action, the planner eventually reduces the incompleteness of the knowledge so that a solution plan is found or the goal is proven to be unsatisfiable. For example, having no information on the location of a paper, the planner may adopt a hypothesis that the paper is available from a certain collection, and verify the information by querying the collection. Hypothetical plans in addition to causal links between a subgoal and an entailing it effect, contain *hypothetical links*, which link knowledge effects to domain subgoals. The idea is as follows: if neither $p$, nor $\neg p$ is known to be true prior to $S$, and there is a sensing action $a_s$ whose effect entails knowing the truth value of $p$, then by executing $a_s$ the planner may find out that $p$ is true. A hypothetical plan leads to a solution plan, if after verifying the hypothesis, the plan can be successfully completed, which is not guaranteed even when the hypothesis is confirmed by an observation.

An alternative approach to planning with incomplete information is conditional planning, i.e. creating branching plans based on the possible outcomes of a sensing action. Applied to the above scenario, conditional planning would involve planning ahead for each of the two possible outcomes of checking if the paper is available from the searched collection. However, in the environments with a high degree of incompleteness, planning ahead for every contingency is computationally prohibitive, especially when a sensing action involves information on multiple atoms.

Furthermore, predicting all possible outcomes of sensing in a meaningful way becomes impossible when the sensing action may discover new objects. In such situations, the agent needs to proceed with execution and then complete the plan given the observation. For example, consider the goal of removing all fragile objects from a room. Given no prior information on the contents of the room, it is impossible to predict which objects are in it, if any of them are fragile, and therefore need to be removed. Thus, it does not make sense to create plans for removing any objects until the information on the contents of the room becomes available.

Suppose that the agent operating in the room can perform the sensing action of identifying all objects in the room, and another one, determining if the object is marked as fragile. Hypothetical planning would hypothesize that by using the first action the agent may discover that no objects are inside the room, thus yielding the goal of having no fragile objects satisfied. If however, upon executing the first action some objects are found inside the room, the agent now has a choice of either creating a plan to move all discovered objects out, or first identifying which are fragile and only removing those marked as fragile. The first solution can be obtained without any further information gathering, the second solution again requires hypothetical planning and execution.

An approach to interleaving planning with execution performed by XII [Golden *et al.*, 1994] and PUCCINI [Golden, 1998] planners (both based on the approach used in IPEM [Ambros-Ingerson and Steel, 1988]) is the other alternative to hypothetical planning. This method treats execution as one of the nondeterministic choices within the planning algorithm. In hypothetical planning execution is triggered by the need, thus it is more tightly constrained, and used only when necessary. The hypothetical planner's behavior is thus not dependent on the model of nondeterminism in the planner implementation and is better suited for an application in which the time of response is critical and sensing operations may take considerable time or are otherwise costly.

Hypothetical planning has been implemented in a partial order planning [Russell and Norvig, 1995] algorithm called PSIPOP-SE and used at the core of a collaborative bibliography assistant, called Writer's Aid [Babaian *et al.*, 2002]. PSIPOP-SE extends a sound and complete open world planner PSIPOP[Babaian and Schmolze, 2000] to planning with sensing, knowledge goals and interleaved execution. It uses PSIPLAN-S representation for reasoning and planning with incomplete information, sensing and knwoeldge goals.

When the set set of agents sensing actions is rich, the use of hypothetical plans may considerably expand the search space. To limit the search space, PSIPOP-SE explores the search space gradually increasing the maximum allowed plan number of hypotheses made in supporting a subgoal. This parameter is called the *hypothetical level of a plan.* Hypothetical level of a simple plan is 0. An example of a plan with hypothetical level two is a plan that hypothesizes that a paper is available from the author's homepage, and then, having no information about the author's homepage, hypothesizes that the url for the homepage can be found from a known index. Verification of each hypothesis reduces the uncertainty, therefore the size of subspace of hypothetical plans on each consecutive level is reduced, while the lower-level hypothetical subspace is explored. In our experiments Writer's Aid was unable to explore the entire space of plans of hypothetical level up to 2 at once due to the large size of this space, but was successful at exploring subspaces gradually, starting from maximum hypothetical level of 0. We call this approach **iterative deepening in hypotheticals**

The rest of the paper is organized as follows. An overview of the PSIPLAN representation is presented in the next section. The definition of a hypothetical link and the partial order planning algorithm interleaving planning with execution PSIPOP-SE are presented in Section 3.

## 2 Overview of PSIPLAN

PSIPLAN assumes infinite number of domain constants, and no other function symbols. PSIPLAN propositions include **ground domain atoms**, **domain $\psi$-forms** and **knowledge $\psi$-forms**. The general form of a $\psi$-form is

$$[Q(\vec{x}) \text{ except } \{\sigma_1, \ldots, \sigma_n\}],$$

and it represents a possibly infinite set of ground propositions that are obtained by instantiating the formula $Q(\vec{x})$ called the *main form*, with all possible ground assignments on the variables in $\vec{x}$, except for the instances specified by the substitutions $\sigma_i$ called the *exceptions*. Each $\sigma_i$ is a substitution on a subset of variables of $\vec{x}$. The main form $Q(\vec{x})$ of a *domain* $\psi$-form is a disjunction of negated literals. In *knowledge* $\psi$-forms $Q(\vec{x})$ has a form $KW(P(\vec{x}))$, where $P(\vec{x})$ is a disjunction of negated literals. All variables in $\vec{x}$ are implicitly universally quantified.

The combination of **domain atoms and $\psi$-forms** is necessary to describe situations as the following one, in which the agent knows that

> *The only bibliographies preferred by Ed are the digital library of the ACM, and maybe the ResearchIndex database.*

In PSIPLAN-S the example statement above is expressed by stating that

1. *ACM's digital library is a preferred bibliography*, which is represented by a ground atom:
$$a = PrefBib(ACM) \tag{1}$$
2. *Nothing is a preferred bibliography except for ACM and the ResearchIndex*, which is represented by the $\psi$-form:
$$\psi = [\neg PrefBib(b) \text{ except } \{\{b = ACM\}, \{b = RI\}\}] \tag{2}$$
Thus, $\psi$ denotes all ground instances of the formula $\neg PrefBib(b)$ minus two exceptions: $\neg PrefBib(ACM)$ and $\neg PrefBib(RI)$ and is equivalent to the universally quantified predicate calculus formula $\forall b. \neg PrefBib(b) \vee (b = ACM) \vee (b = RI)$

Formally, we define the set of ground propositions represented by a $\psi$-form as follows

1. $\phi([Q(\vec{x})]) = \{Q(\vec{x})\sigma \mid Q(\vec{x})\sigma \text{ is ground }\}$
2. $\phi([Q(\vec{x}) \text{ except } \{\sigma_1, \ldots, \sigma_n\}]) =$ $\phi([Q(\vec{x})]) - \phi([Q(\vec{x})\sigma_1]) - \ldots - \phi([Q(\vec{x})\sigma_n])$

Note that assuming infinite number of individual domain objects, a finite set of PSIPLAN-S domain propositions can represent an infinite number of ground negated clauses without the knowledge of all domain objects by the virtue of implicit universal quantification in $\psi$-forms. However, it can represent only finite *"positive knowledge"*, i. e. finite number of atoms.

The algorithms for reasoning with $\psi$-formsare not presented in this paper (see [Babaian, 2000]), however, we note that these computations are carried out by manipulations on the main form and exceptions of the $\psi$-forms without expanding the $\psi$-form into the corresponding set of ground propositions.

**Knowledge $\psi$-forms** similarly to domain $\psi$-forms, represent a conjunction of all ground instances of the main form, however each ground instance in this case is a *knowledge proposition.* Knowledge propositions have form $KW(p)$, where $p$ is a ground clause and represent *knowing p or knowing* not $p$, i.e. that the value of a domain clause $p$ is known without committing to a particular value. For example, $KW(PrefBib(ACM))$ represents knowing-whether $ACM$ is a preferred bibliography. Note that $KW(p)$ is semantically equivalent to $KW(\neg p)$. However, in the main form of a $\psi$-form the $KW$-fied formula is always a negated clause, as in the knowledge $\psi$-form below that represents knowing the set of all preferred bibliographies.

$$\tilde{\psi} = [KW(\neg PrefBib(b))] \tag{3}$$

Knowledge propositions in PSIPLAN-S are used to reason about knowledge and ignorance, represent information goals and results of sensing actions. For example, posted as a goal, $\tilde{\psi}$ requires knowing the value of each ground instance of $PrefBib(b)$, or in other words, knowing the set of preferred bibliographies. The effect of checking if $RI$ is a preferred bibliography, is a knowledge proposition $KW(PrefBib(RI))$. A negated kw-proposition $\neg KW(p)$ represents ignorance about $p$.

### Semantics

A *world state* is a truth assignment on domain atoms. $w(q)$ denotes that $q$ is *true* in the world state $w$. Let $\mathcal{W}$ denote the set of all world states.

To define a model we use *k-states* of Baral and Son [Baral and Son, 2001]. A k-state is a pair $(w, W)$, where $w$ denotes a world state from $\mathcal{W}$, and $W$ denotes a a set of world states. A k-state represents a knowledge state of an agent who actually being in the world state $w$ thinks it can be in any of the world states of $W$.

A set of *models* is denoted by $\alpha$ and defined below. We are assuming that the agent's knowledge is *correct*, hence we require that for any k-state $(w, W)$ in a model $w \in W$. In what follows, $c$ represents a ground negated domain clause and $q$ represents a ground domain proposition, i.e. domain atom or a ground negated clause.

1. $\alpha(q) = \{(w, W) \mid w \in W \wedge \forall w' \in W. w'(q)\}$
2. $\alpha(KW(c)) = \{(w, W) \mid w \in W \wedge ([\forall w' \in W. w'(c)] \vee [\forall w' \in W. w'(\neg c)])\}$
3. $\alpha(\neg KW(c)) = \{(w, W) \mid w \in W \wedge [\exists w' \in W. w'(c)] \wedge [\exists w'' \in W. w''(\neg c)]\}$.
4. $\alpha(\{q_1, \ldots, q_k\}) = \cap_{i=1}^{k} \alpha(q_k)$.

A set of ground propositions $q_1, \ldots, q_k$ *k-entails* (or, for brevity, entails) another ground proposition $q$, denoted $q_1, \ldots, q_k \models_k q$ if $\alpha(\{q_1, \ldots, q_k\}) \subseteq \alpha(q)$.

Note that according to this semantics the k-entailment of ground domain propositions is equivalent to the ordinary entailment. Furthermore,

$$q \models_k KW(q), \text{ and,} q \models_k KW(\neg q).$$

A set of models of a PSIPLAN-S proposition is defined as the set of models of the set of ground propositions it represents.

**Definition 1** For a PSIPLAN-S proposition $p$,$\alpha(p)$ is defined as the set of models $\alpha(\phi(p))$.

**Definition 2** For a set of PSIPLAN-S propositions $p_1, \ldots p_m, p$

$$p_1, \ldots p_m \models_k p \text{ if and only if } \alpha(\{p_1, \ldots, p_m\}) \subseteq \alpha(p)$$

## 2.1 $\psi$-form Entailment

While we do not have the space to present the details of the algorithms for computing entailment in PSIPLAN, we state several key properties underlying those algorithms, and illustrate them with examples. The property critical for the efficiency of $\psi$-form reasoning is formulated in Theorem 1 below: given a set of $\psi$-forms $\Psi = \{\psi_1, \ldots, \psi_n\}$, $\Psi \models_k \psi$ only if there is a $\psi$-form $\psi_i \in \Psi$ that *nearly entails* $\psi$, i.e. main part of $\psi_i$ entails the main part of $\psi$, or $[\mathcal{M}(\psi_i)] \models_k [\mathcal{M}(\psi)]$.

**Theorem 1** Given a set of $\psi$-forms $\Psi = \{\psi_1, \ldots, \psi_n\}$ and a $\psi$-form $\psi$, $\Psi \models_k \psi$ only if there is a $\psi$-form $\psi_i$ in $\Psi$ such that $[\mathcal{M}(\psi_i)] \models_k [\mathcal{M}(\psi)]$.

**E-Difference** For any two sets of ground propositions $A$ and $B$, *e-difference* is defined as follows.

$$B \dot{-} A = \{b \mid b \in B \;\wedge\; A \not\models_k b\}$$

As $\psi$-forms are compact representations of sets of ground propositions, we extend the e-difference operation to $\psi$-forms. The following example illustrates the e-difference operation.

**Example 1** Let $\psi$ denote $[Kn(\neg In(R, z))$ except $\{\{z = A\}, \{z = B\}\}]$, which represents that there are no items in room R except for possibly $A$ and $B$. Further, let $\tilde{\psi}$ denote $[KW(\neg In(R, x) \vee \neg Fragile(x))]$, which can represent a goal of knowing for all objects (x) if they are inside room R and also fragile. $\psi$ entails *most* of $\tilde{\psi}$, indeed, since $\neg In(R, z)$ is *true* for all values of $z$ except possibly $A$ and $B$, then so is the disjunction inside the $\tilde{\psi}$'s $KW$ clause. Thus, the only parts of $\tilde{\psi}$ that are not entailed by $\psi$ are

$$\tilde{\psi}_1 = [KW(\neg In(R, A) \vee \neg Fragile(A))]$$
$$\tilde{\psi}_2 = [KW(\neg In(R, B) \vee \neg Fragile(B))]$$

and therefore $\tilde{\psi} \dot{-} \psi = \{\tilde{\psi}_1, \tilde{\psi}_2\}$

The e-difference operator plays a key role in computing entailment. The next Theorem describes the necessary and sufficient conditions for entailment of a domain or a knowledge $\psi$-form by a set of domain atoms and $\psi$-forms.

We call a set $s$ of domain propositions *saturated*, when there are no possible resolutions between a ground atom $a$ and a ground negated clause $\neg a \vee \neg a_1 \vee \neg a_n$, represented by some $\psi$-form in $s$. A saturated equivalent of such a set can be computed in polynomial time in the number of propositions ($\psi$-forms and atoms) in $s$.

**Theorem 2** Let $s = A \cup \Psi$ be a consistent saturated set of domain atoms ($A$) and $\psi$-forms ($\Psi$), and $\psi$ is any $\psi$-form (either domain or knowledge). $s \models_k \psi$ if and only if

1. there exist $a_1, \ldots, a_n \in A$, such that $\psi = [KW(\neg a_1 \vee \ldots \vee \neg a_n)]$, or
2. there exists $\psi \in \Psi$, such that $[\mathcal{M}(\psi_k)] \models_k [\mathcal{M}(\psi)]$, and, furthermore, $s - \psi \models_k (\psi \dot{-} \psi_k)$

### PSIPLAN-S SOK

SOK (State Of Knowledge) database is a consistent set of PSIPLAN-S domain atoms or psiforms. It represents the knowledge available to the system in the following way:

1. a domain proposition $p$ is true in the world, if and only if $SOK \models_k p$,
2. furthermore, we make a Closed Know-Whether Assumption (**CKWA**) and assume that if $SOK \not\models_k KW(p)$ then the truth value of $p$ is not known, i.e. $\neg KW(p)$

The set of possible worlds corresponding to this representation consists of all world states in which everything known to the agent is true, and only things known to the agent are guaranteed to be true. Such representation is sound and complete, due to soundness and completeness of reasoning about domain and knowledge propositions from a set of domain propositions in PSIPLAN. Importantly, the inference procedures also run in polynomial time and are fast, which bears directly on the speed of planning with PSIPLAN-S. PSIPLAN-S thus ensures precise and fast reasoning about knowledge and ignorance.

### PSIPLAN-S Actions and SOK update

PSIPLAN-S distinguishes two types of actions: *domain actions* that change the world (e.g., an action of downloading a paper from a url), and *sensing actions* that do not change the world but only return information about it (e.g., querying a bibliography).

Each domain action has a list of *preconditions*, $\mathcal{P}$, and an encoding of the effects of the action as a set of literals, called the *assert list*, $\mathcal{A}$. The propositions in $\mathcal{P}$ can include literals and quantified $\psi$-forms, where the term *quantified* is used informally to denote a $\psi$-form that uses at least one variable, and thus represents infinite number of ground instances. We assume that an action is deterministic and can change the truth-value of only a *finite* number of atoms, thus assert list contains literals only, and no quantified $\psi$-forms.

To update SOK $s$ after executing a domain action $a_d$ all propositions whose truth value[1] could have been changed must be removed from $s$ – these are all propositions entailed by the *negation* of some effect of $a_d$. The propositions entailed by effects of $a_d$ are also removed, and then the effects of $a$ are added to the new SOK. The agent's SOK after executing a domain action $a_d$ in the SOK $s$ is computed by function *update*$(s, a_d)$ below.

$$update(s, a_d) = ((s \dot{-} \mathcal{A}^-(a_d)) \dot{-} \mathcal{A}(a_d)) \cup \mathcal{A}(a_d), \qquad (4)$$

where $\mathcal{A}^-(a_d)$ denotes the set of propositions obtained by negating each proposition in $a_d$'s, assert list $\mathcal{A}(a_d)$.

Sensing actions also have preconditions. Effects of the sensing are given by its *knowledge list*, denoted $\mathcal{K}$. The propositions in $\mathcal{K}$ are kw-$\psi$-forms. After a sensing action is executed, it returns an *observation list* of kn-propositions corresponding to the information that was learned, denoted $\Delta$.

**Download(?p, ?s, ?u)**
$\mathcal{P}: HasPaper(?u, ?s, ?p)$
$\mathcal{A}: Got(?p)$

**QueryBib(?b, ?kwd)**
$\mathcal{P}: PrefBib(?b)$
$\mathcal{K}$: $[KW(\neg Rel(p, ?kwd) \vee \neg InCollection(p, ?b))]$

Figure 1: Example of Writer's Aid's domain and sensing actions. The variable p is implicitly universally quantified. Other variables are action schema parameters

Figure 1 provides examples of two PSIPLAN-S actions. *Download(?p, ?s, ?u)* is an action of downloading paper $?p$ from url $?u$ of source $?s$.*QueryBib(?b, ?kwd)* is a sensing action that identifies all papers, which according to bibliography $?b$ are related to keyword $?kwd$. The effect of this action is encoded in the knowledge list that contains a quantified $\psi$-form, and states that as a result of this action the set of all papers in collection of bibliography $?b$ that are related to keyword $?kwd$ will be identified.

For example, suppose after executing sensing action $a = QueryBib(ACM, XII)$ with effect $[KW(\neg Rel(p, ?kwd) \vee \neg InCollection(p, ?b))]$ papers *Paper$_1$* and *Paper$_2$* were found as the only ones related to keyword XII, i.e. $\Delta(a)$ consists of the following propositions:

$$\begin{array}{l} [\neg Rel(p, XII) \vee \neg InCollection(p, ACM) \\ except\{p = Paper_1\}, \{p = Paper_2\}] \\ Rel(Paper_1, XII), InCollection(Paper_1, ACM) \\ Rel(Paper_2, XII), InCollection(Paper_2, ACM) \end{array} \qquad (5)$$

After the execution of a sensing action $a_s$, the set of observed propositions, denoted below by $\Delta(a_s)$ is added to the SOK, i.e.

$$update(s, a_s) = s \cup \Delta(a_s) \qquad (6)$$

After propositions from $\Delta(a)$ are added to the SOK, all possible resolutions from SOK propositions are computed and added to the new SOK – this is a necessary step that guarantees soundness and completeness of domain goal inference in PSIPOP-SE.

[1] true or false

## 3 Planning with Hypotheticals

We assume the reader's familiarity with Partial Order Planning (POP) [Russell and Norvig, 1995]. PSIPOP-SE is a partial order planner that builds on its predecessors: a sound and complete open world partial order planning algorithm PSIPOP [Babaian and Schmolze, 2000] and PSIPOP-S[Babaian, 2000], which is an extension of PSIPOP to planning with sensing and knowledge goals. All three algorithms are based on PSIPLAN-S representation and calculus. PSIPOP-SE extends PSIPOP-S to planning with execution.

A **hypothetical link** is a link between an effect of a sensing action and a domain subgoal, when the truth value of the subgoal proposition is unknown and it is possible that the result of sensing will reveal that the subgoal is true. To define hypothetical links formally, we first need to define the $kwfy()$ operation for PSIPLAN-S domain propositions. Intuitively, the purpose of $kwfy(p)$ is to reflect the existing knowledge regarding all ground propositions represented by $p$. $kwfy(p)$ defines the smallest PSIPLAN-S knowledge proposition implied by $p$.

**Definition 3** $kwfy(p)$ **operator.**

- For a domain atom $a$, $kwfy(a) = [KW(\neg a)]$.
- For a domain $\psi$-form $[P(\vec{x})$ except $\{\sigma_1, \ldots, \sigma_n\}]$,
  $$kwfy(\psi) = [KW(P(\vec{x})) \text{ except } \{\sigma_1, \ldots, \sigma_n\}].$$

A hypothetical link is created between an effect $k$ of a sensing step $S_s$ and a (domain) precondition $p$ on step $S_p$ if and only if

1. $k \models_k kwfy(p)$, i.e. the effect of sensing will result in knowing the truth value of every ground propositions denoted by $p$, and
2. $kwfy(p)$ does not hold immediately prior to step $S_p$, i.e. the values of at least some ground propositions denoted by $p$ are not known prior to $S_p$.

Hypothetical links are similar in spirit to Golden's *observational* links [Golden, 1998], but observational links to $p$ do not require agent's ignorance regarding $p$ and are formulated using conditional effects rather than knowledge propositions.

In the example, illustrated in Figure 2, the planner attempts to find support to a precondition to the *Download* action. The precondition $HasPaper(P, ?s, ?u)$ requires that paper $P$ be available for download from some source $?s$ at some url $?u$. Suppose, that neither the agent's current state of knowledge nor its domain actions can bring about the achievement of the goal, however there is a sensing action $QuerySourceForPaper(P, ?s)$ with effect $k = [KW(\neg HasPaper(P, ?s, u))]$. This effect entails $kwfy(HasPaper(P, ?s, ?u))$, which equals $[KW(\neg HasPaper(P, ?s, ?u))]$ . Note that here, as everywhere else, variables $?u, ?s$ are implicitly existentially quantified and treated as Skolem constants, while $u$ in the knowledge effect $k$ is the $\psi$-form's universally quantified variable. To ensure that the sensing would not be redundant, the planner first tries to prove that given the current partial plan, the value of $HasPaper(P, ?s, ?u)$, is not already known, by calling procedure $VerifyIgnorance()$.

Procedure $VerifyIgnorance()$is passed a partial plan and a domain subgoal $p$ on step $S_p$, and tries to find support to the goal $p$ without adding any new actions. When it fails to find support for $kwfy(p)$, by the CKWA we can assume value of $p$ is not known, and the procedure returns true. Otherwise, it returns false.

```
VerifyIgnorance($plan, p, S_p$)
  if exist effects $e_1, \ldots, e_n$ of steps in $plan$
  possibly before $S_p$, such that $e_1, \ldots, e_n \models_k kwfy(p)$
     return false
  else return true
```

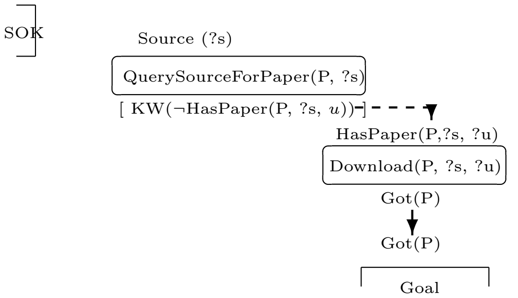


Figure 2: A depiction of a hypothetical plan. (Steps are represented by boxes containing action operator's name and parameters. – with dashed arrows.

The maximum number of consecutive hypotheses made in supporting any subgoal in a plan is called the **hypothetical level** of a plan Hypothetical level of a regular (also here called *simple*) partial order plan is 0. The space of hypothetical plans is explored gradually, by limiting the maximum allowed hypothetical level of a plan to avoid too much hypothesizing.

PSIPOP-SE **algorithm** is outlined in Figure 3. Note that this formulation is generalized and leaves out many details of PSIPLAN-S reasoning and associated goal satisfaction and threat resolution techniques, which can be found in [Babaian and Schmolze, 2000; Babaian, 2000], in order to focus on the details of planning with hypotheticals. PSIPOP-SE is a nondeterministic algorithm that is passed the initial plan encoding just the current SOK and goal state as its initial and goal steps, and an additional parameter `maxHL` denoting the maximum hypothetical level of explored plans. The following fields are added to the standard plan structure to support hypothetical planning: `hlevel` denotes the hypothetical level of the plan, `suspendedGoals` denotes a list of sets of goals, planning for which is suspended until the sensing step(s) are executed.

PSIPOP-SE starts by calling procedure POPH. POPH is searching for a way of supporting an open goal of a partially ordered plan that is passed in as a parameter, and simultaneously explores the hypothetical support for the goal. Hypothetical plans are created by procedure *FindHypPlans*, which nondeterministically chooses a sensing step - source of the hypothetical link to the goal in consideration, suspending the rest of the plan's open goals, and setting the set of plan's goals to the precondition of the added sensing step. The hypothetical plans returned by *FindHypPlans* are not expanded further unless the search for a simple solution plan results in failure.

If POPH returns with a failure, in other words, a simple plan that achieves a goal does not exist, PSIPOP-SE nondeterministically chooses a hypothetical plan from HPlans. The picked hypothetical plan has as its list of open goals the preconditions of the earliest source of the first in order hypothetical link, and the rest of the plan's open preconditions as its suspended goals. These subgoals were suspended because unless the target condition of the hypothetical link is found to be true, it does not make sense to continue planning to satisfy the rest of subgoals of the plan.

To enable execution of the first in order information gathering action, PSIPOP-SE calls procedure HPOP, which searches for a (partial) plan that makes the sensing step-source of the hypothetical link executable from the initial state. If such completion is found, HPOP executes the plan up until the source of the hypothetical link, otherwise, the next hypothetical plan is explored.

Upon execution of each action SOK is updated according to equations (4) and (6) in procedure *UpdateAfterExecution*. The executed plan is updated as well: the executed steps are removed, links originating in the executed steps are now drawn from the initial step denoting the SOK, previously suspended goals are restored and the planner continues to work towards completing the plan.

It is possible that due to the executed portion of the plan some sensing acts may have become redundant, as previously unknown propositions became known. To avoid redundant information gathering, procedure *HPOP* verifies that the sensing is necessary by calling *VerifyIgnorance*, when picking the next hypothetical plan to expand. Note also that some causal links may be invalidated because the truth value of a proposition was reversed by the executed actions. This would not happen to the executed current plan, but it may affect other hypothetical plans in HPlans. Thus, *HPOP* may discard some invalid links originating from the initial step (SOK) that are no longer valid, adding their target conditions to the plan's goals.

## 4 Conclusions and Future Work

We have presented a novel method for interleaving planning with execution, which enables information gathering to be used in support of planning goals. The method has been implemented within a partial order planner, however, its formulation is based on the general concepts of entailment, reasoning about knowledge and ignorance, which could make the method applicable to other planning and acting frameworks.

```
PSIPOP-SE (init-plan, maxHL)
HPlans = $\emptyset$ // hypothetical plans
if POPH(init-plan, HPlans, maxHL) fails
      Choose a plan ph from HPlans
      remove ph from HPlans
      HPOP(ph, maxHL, HPlans)

POPH(plan, maxHL, HPlans)
if (plan.goals = $\emptyset$) return plan
else plan' = copy(plan)
   Choose a goal g from plan'.goals
   if FindSupport(plan',g) fails or
   ResolveThreats(plan',g) fails)
      result =$\emptyset$
   HPlans=HPlans$\cup$FindHypPlans(plan',maxHL, g)
   if result =$\emptyset$ then fail
   else POPH(plan', maxHL, HPlans)

FindHypPlans(plan, maxHL, g)
// where g denotes a precondition $p$ on step $S_p$
if (plan.hlevel<maxHL) and $VerifyIgnorance$(p, S$_p$) =
true
   Choose a sensing operator S$_s$ with effect $k$
   such that $k \models_k kwfy(p)$. If found $S_s$:
      planh = copy(plan)
      add hypoth. link $S_s --> S_p$ to planh.links
      planh.hlevel = planh.hlevel + 1
      push (planh.goals) to planh.suspendedGoals
      planh.goals = $\mathcal{P}(S_s)$
      return planh

HPOP(planh, maxHL, HPlans)
-- Complete and execute a hypothetical $planh$
Remove invalid causal links with source in the SOK
from planh.links,
add their goals to plan.goals
Find the earliest step $S_s$ - source of hypothetical
link in planh.
Suppose it is linked to precondition $p$ of $S_p$
if VerifyIgnorance(p, S$_p$) = true and ph.goals$\neq \emptyset$
      // find an executable completion of ph, phe
      phe = POPH(planh, maxHL, HPlans))
   else phe = ph;
   Execute phe up to and including $S_s$
   UpdateAfterExecution (phe)
   if (phe.hlevel > 0)
   // remaining plan still has hypothetical links
      HPOP(phe, maxHL, HPlans)
   else POPH(phe, maxHL, HPlans)

UpdateAfterExecution (ph)
For each executed step S in ph
   SOK = update(SOK,S) // equations (4,6)
   Replace S with SOK in all causal links
   originating from S to the rest of plan
ph.hlevel = ph.hlevel - 1
ph.goals = pop a list from ph.suspendedGoals
```

Figure 3: Nondeterministic algorithm PSIPOP-SE.

Future research needs to focus on fully exploring the properties of hypothetical planning on problems from a variety of domains, generalizing the hypothetical planning approach to planning in domains with irreversible actions, and examining formal issues related to soundness and completeness of the search for hypothetical plans in PSIPOP-SE.

## References

[Ambros-Ingerson and Steel, 1988] Jose A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 83–88, St. Paul, Minnesota, 21–26 August 1988. Morgan Kaufmann.

[Babaian and Schmolze, 2000] T. Babaian and J. Schmolze. Psiplan: open world planning with $\psi$-forms. In *Proceedings of AIPS'00*, pages 292–300, 2000.

[Babaian *et al.*, 2002] Tamara Babaian, Barbara J. Grosz, and Stuart M. Shieber. A writer's collaborative assistant. In *Proc. of IUI'02*, pages 7–14. ACM Press, January 2002.

[Babaian, 2000] Tamara Babaian. *Knowledge Representation and Open World Planning Using $\psi$-forms.* PhD thesis, Tufts University, 2000.

[Baral and Son, 2001] Chitta Baral and Tran Cao Son. Formalizing sensing actions – a transition function based approach. *Artificial Intelligence*, 125, 2001.

[Golden *et al.*, 1994] K. Golden, O. Etzioni, and D. Weld. Omnipotence without omniscience: Efficient sensor management for planning. In *Proceedings of AAAI-94*, 1994.

[Golden, 1998] Keith Golden. Leap before you look: Information gathering in the puccini planner. In *Proceedings of AIPS'98*. AAAI Press, June 1998.

[Russell and Norvig, 1995] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, 1995.

# Risk-directed Exploration in Reinforcement Learning

**Edith L.M. Law** and **Melanie Coggan** and **Doina Precup** and **Bohdana Ratitch**

McGill University
School of Computer Science
3480 University Street
Montreal, Quebec, Canada
H3A 2A7

## Abstract

Learning agents that have to act autonomously and learn without an explicit teacher are faced with an important dilemma. On one hand, they need to explore their environment in order to gather information. On the other hand, exploration can result in action choices with catastrophic consequences. This is an important issue for agents that learn in a realistic setting, rather than in simulation. Hence, it is important to assess the risk of actions and be able to learn quickly how to avoid "bad" outcomes. We present a heuristic approach for defining risk for reinforcement learning agents, and an algorithm for incorporating this risk notion into exploration. Unlike other existing work, our method still allows the agent to learn optimal action values. Our heuristic allows the definition of agents that are either risk-averse or risk-seeking. In preliminary experiments, risk-based RL agents compare favorably with agents using other undirected and directed exploration methods.

## 1 Introduction

Intelligent agents are increasingly used for tasks that humans would not or could not perform. For example, robots may be deployed to collect data on an unexplored planet, clean up toxic waste at a disaster zone, and recover sunken objects from the deep sea. Artificial intelligence is used in medicine to make diagnosis, recommend short term and long-term treatment strategies, or dynamically control biomedical devices. In these safety critical systems, it is crucial for autonomous agents to be as conservative as possible, in order to minimize the risk of damaging expensive robotic machinery in the former case, and of undermining patient well-being in the latter. Common to these applications is the fact that agents are required to operate autonomously in environments that are unknown, uncertain and changing. Second, bounding the overall risk of the system does not suffice, if the system is to be used in real time. In a potentially hazardous environment, a single wrong choice of action may lead to fatal and irrecoverable consequences.

Reinforcement learning (RL) provides a framework for learning in stochastic, dynamic environments whose model is unknown a priori or incomplete. RL agents learn about the environment by selecting actions that are informative, a process known as *exploration*. At the same time, they strive to behave optimally, by selecting the best known action in any given state (*exploitation*). One of the key issues facing RL agents in practical applications is how to balance exploration and exploitation [Bulitko, 2004]. Most of the existing exploration methods are aimed at ensuring that the agent can gather enough information about the environment. However, this process can result in "catastrophic" outcomes, which are not explicitly considered by most existing exploration methods. A standard assumption is that learning takes place in simulation, and hence many reincarnations of the agent are possible. However, this is not always an option.

Two main approaches to handling risk in exploration have been proposed in prior RL research. One approach is to formulate the control problem that the agent is trying to solve in such a way that risk is not an issue. Research along this line, e.g. [Milan, 1996; Singh *et al.*, 1994] is based on formulating actions that are not risky, based on prior knowledge about the environment. As second line of research is based on transforming the action values that are being learned such that risk is taken into account. Existing approaches include solving a Markov Decision Problem (MDP) subject to constraints on the variance of the returns [Sato and Kobayashi, 2000] or on the frequency of entering a fatal state [Geibel, 2001], and distorting the action values [Heger, 1994; Gaskett, 2003; Neuneier and Mihatsch, 2002]. There are several reasons why it is not desirable to induce risk-averse behavior by transforming the action values. First, if the action values are updated based on a conservative criterion, the policy may be overly pessimistic. Second, the distortion of the action values means that the true long-term utilities of actions are not computed accurately anymore. Hence, it is harder to understand what kind of approximation errors the agent will produce.

In this paper we present an alternative approach, which does not distort the action values. We adapt a risk measure defined in economics for one-shot decision making to MDPs. We present a straightforward directed exploration algorithm which uses the risk measure, together with the estimated action values, in order to pick actions. We illustrate the way in which this approach can be used to generate *risk-sensitive* behavior (rather than just conservative risk avoidance). In preliminary experiments on gridworld domains, this risk measure compares favorably with other undirected and directed exploration methods.

## 2 The notion of risk

The notion of risk is crucial in economics and decision theory. Economists typically distinguish between decisions under risk and decisions under uncertainty. In decision making under risk, an agent is faced with a set of actions, whose effects are unknown but can be represented in terms of a probability distribution over outcomes. In decision making under uncertainty, no assumptions about the existence of a probability distribution over outcomes can be made. In the context of decision making under risk, actions are essentially equivalent to lotteries, where a lottery $l_i$ is a set of outcomes $o_i$, each of which occurs with probability $p_i$ and is associated with a specific reward $r_i$. The decision problem is analogous to the problem of determining one's preference amongst a set of lotteries at each time step.

We are focusing on sequential decision problems formally represented as MDPs [Bellman, 1957]. An MDP consists of a tuple $\{S,A,T,R\}$, where $S$ is a discrete finite set of states in the environment, $A$ is a discrete finite set of available or permissible actions within the environment, $T$ is a set of matrices consisting of the probabilities of transitioning between states and $R$ is a set of matrices containing the expected rewards associated with transitions. More precisely,

$$T^a_{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a) \ \forall t$$

and

$$R^a_{ss'} = E[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'] \ \forall t$$

The goal of the decision problem is to find a way of selecting actions, called a *policy* which maximizes the long term expected reward:

$$E[\sum_{t=0}^{T'} \gamma^t r_t]$$

where $T'$ is the number of decision epochs, in a finite horizon problem, or $\infty$ for the infinite horizon problem and $\gamma$ is a discounting factor which is used to weigh less the rewards obtained later in the future. RL algorithms compute a good, sometimes optimal policy when the MDP model, specified by the matrices $T$ and $R$, is unknown. In this case, the agent must explore unknown actions in order to gain information about the state space. Risk arises naturally in RL, like in one-step decision making, due to the stochasticity of the environment. In some environments, actions can potentially have "catastrophic" consequences, i.e. destroying the agent in some way. Hence, despite the information they may reveal, actions may *not* always be worth taking. It is important to ask how much risk the agent should tolerate in order to gather information.

## 3 Risk-directed Exploration

Intuitively, an action could be deemed risky under two circumstances: the action may lead to a negative event (one much worse that the "average" event expected); or, the action may have a lot of stochasticity. The second interpretation is based on observations about the behavior of animals and people, who tend to prefer determinism. The risk measure adopted in this paper, which is a variant based on the definition proposed by [Yang and Qiu, 2005], incorporates these two intuitions about risk. Given a state, we define the measure of risk for a particular action $a$ as the weighted sum of the entropy and normalized expected reward of that action:

$$Risk(s,a) = wH(s,a) - (1-w)\frac{E[R^a_{ss'}]}{\max_{a \in A_s} |E[R^a_{ss'}]|}$$

where:

$$H(s,a) = -T^a_{ss'} \log T^a_{ss'} \text{ and } E[R^a_{ss'}] = \sum_{s'} T^a_{ss'} R^a_{ss'}$$

The definition consists of an entropy term, describing the stochasticity of the outcomes of a given action in a state, and a normalized expected reward term, describing how much worse this action is, in terms of immediate consequences, than the best action in this state. These two terms are weighted using a parameter $w$. The risk measure of an action is combined linearly with the action value to form the risk-adjusted utility of an action:

$$U_r(s,a) = p * (1 - Risk(s,a)) + (1-p) * Q(s,a) \quad (1)$$

The first term measures the safety value of an action, while the second term measures the long-term utility of that action. The parameter $p$ provides a way to interpolate between paying attention to the long-term utility of an action, and paying attention to safety.

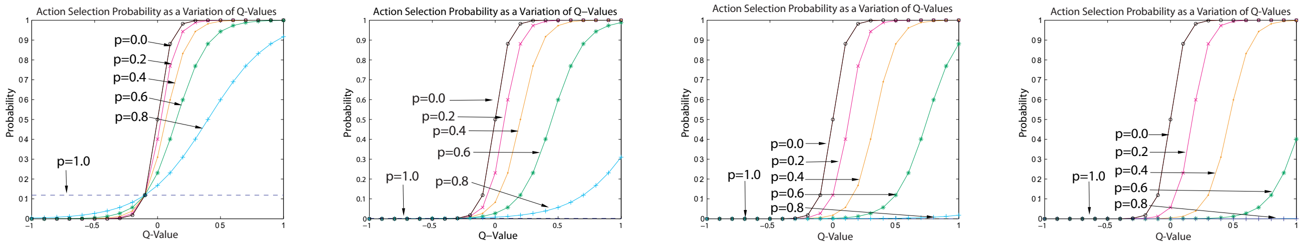
We use the risk-adjusted utility of an action as a substitute for action values in a Boltzmann distribution:

$$\pi(s,a) = \frac{e^{\frac{U_r(s,a)}{\tau}}}{\sum_{b=1}^{n} e^{\frac{U_r(s,b)}{\tau}}} \quad (2)$$

where $\tau$ is the temperature parameter. This exploration strategy can be naturally incorporated in value-based RL algorithms, such as Sarsa or Q-learning. The corresponding version of Sarsa, taking into account risk, is given in Figure 1.

```
Initialize $Q(s,a), n(s), n(s,a) \ \forall s \ \forall a$
Repeat (for each episode):
  Initialize $s_t$
  $n(s_t) \leftarrow n(s_t) + 1$
  Choose $a_t$ from $s_t$ using (2)
  Repeat (for each step of episode):
    Execute action $a_t$, observe $r_{t+1}$ and $s_{t+1}$
    $n(s_t, a_t) \leftarrow n(s_t, a_t) + 1$
    Choose $a_{t+1}$ from $s_{t+1}$ using (2)
    $Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) +$
      $\alpha[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$
    Update $Risk(s_t, a_t)$ and $U_r(s_t, a_t)$
    $s_t \leftarrow s_{t+1}$; $a_t \leftarrow a_{t+1}$
  Until $s_t$ is terminal
```

**Figure 1:** Risk-directed exploration in Sarsa



**(a)** $\delta = 0.1$ **(b)** $\delta = 0.3$ **(c)** $\delta = 0.5$ **(d)** $\delta = 0.7$

**Figure 3:** The probability of selecting action $a_1$ when the risk values of the two actions differ by varying amounts $\delta$

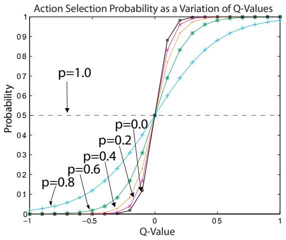


**Figure 2:** Different trends in the probability of selecting action $a_1$ for risk-directed exploration with different $p$-values

Note that computing the risk-adjusted utility requires a model of the environment. Such a model can be given, or it can be learned based on samples, at the same time as the value function. We explored both alternatives in our experiments.

Consider two actions $a_1$ and $a_2$ that have the same risk value. Figure 2 shows that for the standard Boltzmann, $a_1$ has increasingly higher probability of being selected when its action value surpasses that of $a_2$ which is constant at 0, and lower probability of being selected when its action value falls below 0. The probability curve is sigmoid-shaped. A similar trend is observed for the Boltzmann probability that uses risk-adjusted utilities, although the curves become flatter as the $p$-value increases, i.e. as risk term is increasingly overweighted. At $p = 1.0$, the risk measure completely dominates the *risk-adjusted utility* value. Since the two actions have the same risk, they are picked with equal probability at all times. Note that in practice one would always expect $p < 1$.

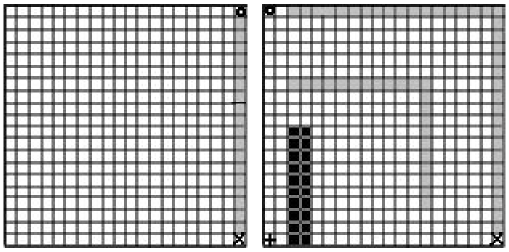
For the intermediate $p$-values, there are two observations. As the $p$-value increases, the probability of selecting $a_1$ is lowered. It requires a greater difference in the predicted value in order for $a_1$ to be preferred. This is analogous to risk-averse behavior. However, it is also true that unless its value is very bad, $a_1$ still has some probability of being chosen. This is due to the fact that the *safety* term in the risk-adjusted utilities can be positive even for risky actions. As a result, it can potentially raises the risk-adjusted utilities of both good and bad actions, causing the bad actions to be selected more often than desirable. Ideally, the Boltzmann probability function should be transformed such that it is concave for positive utilities and convex for negative utilities. This type of transformation may have interesting connections with the Prospect theory [Kahneman and Tversky, 1979] in psychology, which states that people are risk-averse when prospect are framed in terms of gain, and risk-seeking when prospects are framed in terms of losses, implying the existence of a concave and convex utility function for gains and losses respectively.

What if $a_1$ has a different value of risk than $a_2$? The probability of selecting $a_1$ when the risk value of $a_1$ is higher than $a_2$ by 0.1, 0.3, 0.5, 0.7 are plotted in Figure 3. The value of $a_2$ is 0 at all times, while the value of $a_1$ changes from -1 to 1. Within each plot in Figure 3, the general trend induced by intermediate $p$-values is still observed, i.e. the higher the $p$-values, the flatter the curve. In addition, the more the risk value of $a_1$ increases, the higher the predicted value has to be in order for $a_1$ to be selected. Furthermore, the greater the $p$-value, the more drastically the action selection probability is depressed as the difference of the risk values between $a_1$ and $a_2$ becomes larger.

In short, the parameter $p$ controls the relative risk aversion of the agent. As the value of the parameter $p$ increases, the Boltzmann action selection rule selects the action with higher risk with exceedingly lower probability (Figure 2, 3).

## 4 Experimental Results

An environment typically used in reinforcement learning to evaluate the sensitivity of algorithms to risk is the cliff world. Two versions for this environment are presented in Figure 4. In these environments, the objective of the agent is to travel from the start to the goal state without falling off the cliff.



**(a)** Close-by-cliff World **(b)** Cake-or-cheese World

**Figure 4:** Environments

In both worlds, the state is the exact position of the agent. In a grid world with 20 by 20 tiles, the total number of states is 400. The terminal states include the location where the

| METHOD | A | B | C |
|---|---|---|---|
| SOFTMAX | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ |
| SOFTMAX, $TD(\lambda)$ | — | — | $\alpha = 0.25, \tau = 0.01$ |
| RECENCY-BASED | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.05$ |
| COUNTER-BASED | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.05$ |
| RISK-BASED, P=0.2 | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ |
| RISK-BASED, P=0.4 | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ |
| RISK-BASED, P=0.6 | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.1, \tau = 0.01$ |
| RISK-BASED, P=0.8 | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.25, \tau = 0.01$ |

**Table 1:** Parameter Settings for close-by-cliff world. A: 1-step risk, fixed model; B: 1-step risk, learned model; C: 2-step risk, learned model

| METHOD | A | B | C |
|---|---|---|---|
| SOFTMAX | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ |
| SOFTMAX, $TD(\lambda)$ | — | — | $\alpha = 0.5, \tau = 0.05$ |
| RECENCY-BASED | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.05$ | $\alpha = 0.1, \tau = 0.05$ |
| COUNTER-BASED | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.05$ |
| RISK-BASED, P=0.2 | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.25, \tau = 0.01$ |
| RISK-BASED, P=0.4 | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.25, \tau = 0.01$ |
| RISK-BASED, P=0.6 | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.25, \tau = 0.01$ |
| RISK-BASED, P=0.8 | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ | $\alpha = 0.5, \tau = 0.01$ |

**Table 2:** Parameter Settings for cake-or-cheese world. A: 1-step risk, fixed model; B: 1-step risk, learned model; C: 2-step risk, learned model

*goal* (cheese) is found, and the location of the cliffs. The agents are allowed four actions, i.e. $A$={up, left, right and down}. However, due to the constraints of the boundaries of the grid, the set of permissible actions $A_s$ in each state may be smaller than $A$. With probability 0.8 that the agent will enter a state as intended, and with probability 0.2 that it will slip into the neighboring cells of the intended destination. Finally, the reward for reaching the goal is +1, the penalties for falling off a cliff -1, and the reward for all other states is 0. In the cake-or-cheese world, the reward for reaching the cheese is +1, while the reward for reaching the cake is +0.01. In testing this environment, it would be interesting to observe whether the agent chooses the short path to the small goal, longer but less risky path to the large goal, or the shorter but more risky path to the large goal. The agent has a maximum of 4000 time steps to complete each training trial, and 400 time steps to complete each testing trial.

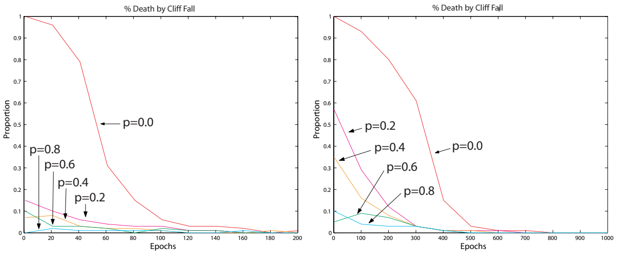
In our experiments, the *performance* of the algorithm is characterized by five measures: (a) the training score in terms of cumulative discounted reward (b) the testing score in terms of cumulative discounted reward (c) % of termination by cliff fall during learning (d) % of termination by reaching the goal during learning (e) the life span during training, in terms of the number of time steps elapsed until termination.

Three sets of experiments are run for each environment: 1-step risk using fixed model, 1-step risk using learned model and 2-step risk using learned model. The experiment is run over 200 episodes for the close-by-cliff world and 1000 episdoes for the cake-and-cheese world and all results are averaged over 20 runs. $w$ is set to be 0.5. The constant used in the counter-based method is 400 for both environments, and that for the recency-based method is 400 for the close-by-cliff world, and 4 for the cake-or-cheese world. $\alpha$ and $\tau$ are optimized for each algorithm for each environment and experimental scenario, as shown in table 1 and table 2.

## 4.1 Effect of varying $p$

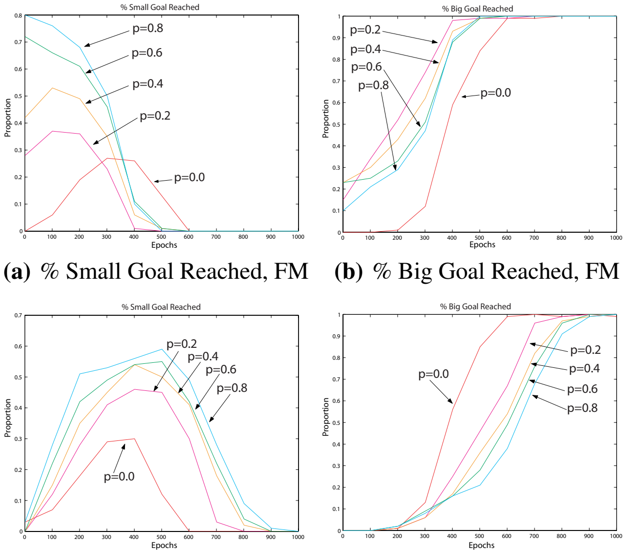
Varying the parameter $p$ produces an interesting range of risk-averse behavior. As shown in Figure 5, when using a fixed model, the higher the $p$-value, the lower the percentage of death during training.

In the cake-or-cheese world, the higher the $p$-value, the more the agent prefers the small goal, especially during the beginning of learning (figure 6). A similar trend can be observed for both the fixed and learned model.



**(a)** Close-by-cliff World **(b)** Cake-or-cheese World

**Figure 5:** Percentage of cliff fall using fixed model



**(a)** % Small Goal Reached, FM **(b)** % Big Goal Reached, FM

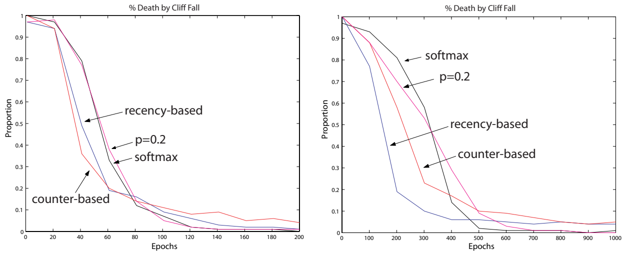
**(c)** % Small Goal Reached, LM **(d)** % Big Goal Reached, LM

**Figure 6:** Percentage of termination at small goal versus large goal in cake-and-cheese world. FM=Fixed Model, LM=Learned Model

## 4.2 Learned model with look-ahead versus without look-ahead

We tested the performance and behavior of the algorithm using a learned model. For one-step risk, risk-directed exploration failed to produce better online learning performance than other directed methods (figure 7).

One remedy is to incorporate more lookahead in the learning, by introducing two-step risk.



**(a)** Close-by-cliff World **(b)** Cake-or-cheese World

**Figure 7:** Percentage of cliff fall using learned model (without lookahead)

$$Risk^{II}(s,a) = \lambda H^{II}(s,a) - (1-\lambda)\frac{E^{II}[R^a_{ss'}]}{\max_{a\in A}|E^{II}[R^a_{ss'}]|}$$

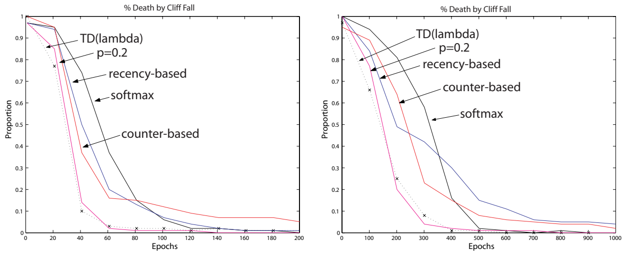
where

$$H^{II}(s,a) = H(s,a) + \textstyle\sum_{s'} T^{a^*}_{ss'} H(s',a^*)$$

$$E^{II}[R^a_{ss'}] = \frac{\sum_{s'} T^a_{ss'}[R^a_{ss'} + \sum_{s''} T^{a^*}_{s's''} R^{a^*}_{s's''}]}{\max_a E^{II}[R^a_{ss'}]}$$

and $a^*$ is chosen to maximize $Q(s',a')$

As shown in Figure 9, using two-step risk results in more distinction between the life span of the agent under different values of $p$, and generally longer life span than using one-step risk. Based on the percentage of cliff fall during training, the performance of risk-directed exploration using two-step risk is comparable to that of $TD(\lambda)$ where $\lambda = 0.7$ (figure 8). However, under $TD(\lambda)$, the agent has much shorter life span and prefers the small goal significantly less than if it were to adopt risk-directed exploration during training (figure 10). Comparing the percentage of termination at the small goal versus the big goal (figure 10) with the equivalent results for one-step risk (figure 6(c) and 6(d)) implies that the effects of risk aversion is much more exaggerated when lookahead is incorporated.

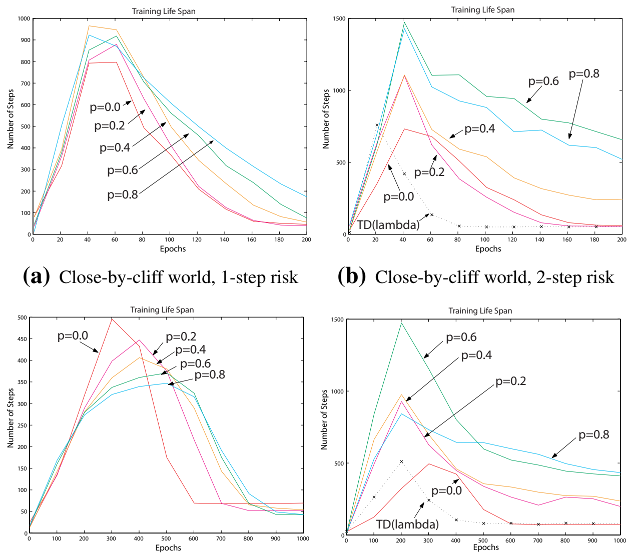


**(a)** Close-by-cliff World **(b)** Cake-or-cheese World

**Figure 8:** Percentage of cliff fall using learned model (with lookahead)

## 5 Discussion and Future Work

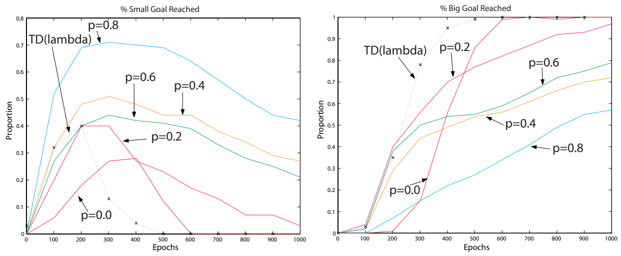
The risk-directed exploration method presented here offers a simple and intuitive solution for ensuring survival during learning by risk avoidance. The mechanism of risk avoidance is achieved by learning the risk values of actions during learning, based on which the probability of selecting that action is adjusted.



**(a)** Close-by-cliff world, 1-step risk **(b)** Close-by-cliff world, 2-step risk

**(c)** Cake-or-cheese world, 1-step risk **(d)** Cake-or-cheese world, 2-step risk

**Figure 9:** Life span using learned model



**(a)** % Small Goal Reached **(b)** % Big Goal Reached

**Figure 10:** Percentage of termination at small goal versus large goal in cake-and-cheese world using learned model (with lookahead)

One criticism of this method may be that by visiting only states that are less risky, the agent does not sample widely enough to have an accurate picture of the environment. As a result, learning an optimal policy will be slower. Our standpoint is that if self-preservation is one of the criteria of an efficient exploration method, this sacrifice is acceptable.

Similarly, the claim that risk aversion is useful for survival is likely to provoke disagreement. One may argue that risk aversion is useful in certain situation, but it can produce pathological behavior in others. Imagine a cliff world environment where the cliff divide the space between the agent and cheese. The risk-directed exploration method will select actions such that the agent remains in the safer region of the environment, never approaching the cheese.

Reflection on the limitation of risk aversion suggests that it may be beneficial for the agent to be risk-averse at certain times, but risk-seeking at other times depending on the current context. In fact, risk sensitivity in decision making has been widely observed in the study of animal foraging behaviour. In one experiment, the yellow-eyed junco birds were presented with a choice between a feeding station that provides a constant supply of three seeds and a second feeding station that provides either no seeds or six seeds with equal probability. It is found that the birds' preferences for the two foraging options depended on the temperature. At normal temperature (19°C), the birds are on a positive energy budget, i.e. the average reward of three seeds is sufficient to maintain the energy level above a critical threshold. It is observed that the birds prefer the constant foraging option that provides three seeds, i.e. they are risk-averse. At low temperature (1°C), where the average reward of three seeds can no longer compensate for the energy expenditure, a reversal in the preference is observed. The birds were risk-seeking, preferring the variable foraging option that has some probability of providing enough seeds to bring the energy level above the critical threshold [Caraco *et al.*, 1990]. This example illustrates that risk attitude does not remain static, but adapts continuously to the environment in favour of actions that maximize the probability of survival. This switch between risk-seeking and risk-averse behaviour is also observed when the source of hazard is not resource depletion, but predation [Milinski and Heller, 1978].

These observations of animal foraging behaviour have interesting implication for decision making in uncertain environments. First, these evidence support the fact that a measure of risk, instead of expected utility, can be potentially useful for the valuation of a prospect. Second, the ability to adjust risk attitude dependent on the context seems to have a clear advantage in ensuring survival, and empirically shown to exist even in human decision making [March and Shapira, 1992]. In addition, risk sensitivity may be useful also for modelling a wide range of rich emotion, behaviour and personality in agents.

The risk-directed exploration method presented in this paper can be easily extended to provide a framework in which the risk attitude is dynamically alternated during the learning phase based on the current context. This can be done by adjusting the parameter $p$ subject to some predetermined schedule of decay, or according to some other constraints. In this paper, we focus on understanding the behaviour and performance of the risk-directed exploration method for a fixed level of $p$. Hence, the appropriate mechanisms for dynamically controlling the risk attitude remains an open research question.

Another interesting enhancement would be the use of a multi-step risk measure. An adjustable window of how far to look ahead when calculating risk can be analogous to paying attention to short term, medium term, or long term risk of an action. Second, the risk measure can be subject to TD learning so that a global, instead of local, measure of risk is derived. In order to subject the risk measure to dynamic programming, the risk measure must have certain desirable properties, e.g. additivity. Hence, it would be useful to characterize exactly what those desirable properties are, and what other definitions of risk are suitable for application. Lastly, this exploration method can be used to select temporally extended actions.

## References

[Bellman, 1957] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, N.J., 1957.

[Bulitko, 2004] Vadim Bulitko. Rl for life notes. 2004.

[Caraco *et al.*, 1990] T. Caraco, W.U. Blanckenhorn, G.M. Gregory, J.A. Newman, G.M. Recer, and S.M. Zwicker. Risk-sensitivity: ambient temperature affects foraging choice. *Animal Behavior*, 39:338–345, 1990.

[Gaskett, 2003] C. Gaskett. Reinforcement learning under circumstances beyond its control. In *Proceedings of the International Conference on Computational Intelligence for Modelling Control and Automation*, 2003.

[Geibel, 2001] P. Geibel. Reinforcement learning with bounded risk. In C. E. Brodley and A.P. Danyluk, editors, *Proceedings of the Eighteenth International Conference (ICML01)*, pages 162–169, San Francisco,CA, 2001. Morgan Kaufmann Publishers.

[Heger, 1994] M. Heger. Consideration of risk in reinforcement learning. In *Proceedings of the eleventh International Conference on Machine Learning*, pages 105–111, 1994.

[Kahneman and Tversky, 1979] D. Kahneman and A. Tversky. Prospect theory: An analysis of decision under risk. *Econometrica*, 47(2):263–292, 1979.

[March and Shapira, 1992] J.G. March and Z. Shapira. Variable risk preferences and the focus of attention. *Psychological Review*, 99(1):172–183, 1992.

[Milan, 1996] J.R. Milan. Rapid, safe and incremental learning of navigation strategies. In *Proceedings of the IEEE Transactions on Systems, Man, and Cybernetics*, volume 26(3), pages 408–420, 1996.

[Milinski and Heller, 1978] M. Milinski and R. Heller. Influence of a predator on the optimal foraging behavior of stickleback. *Nature*, 275:642–644, 1978.

[Neuneier and Mihatsch, 2002] R. Neuneier and O. Mihatsch. Risk-sensitive reinforcement learning. *Machine Learning*, 49:267–290, 2002.

[Sato and Kobayashi, 2000] M. Sato and S. Kobayashi. Variance-penalized reinforcement learning for risk-averse asset allocation. In K.S. Leung, L.W. Chan, and H. Meng, editors, *IDEAL 2000*, pages 244–249, 2000.

[Singh *et al.*, 1994] S. Singh, A. Barto, R. Grupen, and C. Connolly. Robust reinforcement learning in motion planning. In J.D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 655–662. San Mateo, CA: Morgan Kaufmann, 1994.

[Yang and Qiu, 2005] J. Yang and W. Qiu. A measure of risk and a decision-making model based on expected utility and entropy. *European Journal of Operation Research*, 164(3):792–799, 2005.

# Best-first Utility-guided Search

**Wheeler Ruml**
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA
ruml at parc dot com

**Elisabeth H. Crawford**
Computer Science Department
Carnegie Mellon University
Pittsburgh PA 15213 USA
ehc at cs.cmu dot edu

## Abstract

In many shortest-path problems of practical interest, insufficient time is available to find a provably optimal solution. In dynamic environments, for example, the expected value of a plan may decrease with the time required to find it. One can only hope to achieve an appropriate balance between search time and the resulting plan cost. Several algorithms have been proposed for this setting, including weighted A*, Anytime A*, and ARA*. These algorithms multiply the heuristic evaluation of a node, exaggerating the effect of the cost-to-go. We propose a more direct approach, called BUGSY, in which one explicitly estimates search-nodes-to-go. One can then attempt to optimize the overall utility of the solution, expressed by the user as a function of search time and solution cost. Experiments in several problem domains, including motion planning and sequence alignment, demonstrate that this direct approach can surpass anytime algorithms without requiring performance profiling.

## 1 Introduction

Many important tasks, such as planning, parsing, and sequence alignment, can be represented as shortest-path problems. If sufficient computation is available, optimal solutions to such problems can be found using A* search with an admissible heuristic [Hart *et al.*, 1968]. However, in many practical scenarios, time is limited or costly and it is not desirable, or even feasible, to look for the least-cost path. Furthermore, in dynamic environments, a plan's chance of becoming invalid increases with time, making any plan based on current knowledge less valuable as time passes. Instead of ensuring an optimal solution, search effort should be carefully allocated in a way that balances the cost of the paths found with the required computation time. This trade-off is expressed by the user's utility function, which specifies the subjective value of every combination of solution quality and search time. In this paper, we introduce a new shortest-path algorithm called BUGSY that explicitly acknowledges the user's utility function and uses it to guide its search.

A* is a best-first search in which the 'open list' of unexplored nodes is sorted by $f(n) = g(n) + h(n)$, where $g(n)$ denotes the known cost of reaching a node $n$ from the initial state and $h(n)$ is typically a lower bound on the cost of reaching a solution from $n$. A* is optimal in the sense that no algorithm that returns an optimal solution using the same lower bound function $h(n)$ visits fewer nodes [Dechter and Pearl, 1988]. However, in many applications solutions are needed faster than A* can provide them. To find a solution faster, it is common practice to increase the weight of $h(n)$ via $f(n) = g(n) + w \cdot h(n)$, with $w \geq 1$ [Pohl, 1970]. There are many variants of weighted A* search, including $A^*_\epsilon$ [Pearl and Kim, 1982], Anytime A* [Hansen *et al.*, 1997; Zhou and Hansen, 2002], and ARA* [Likhachev *et al.*, 2004]. In ARA*, for example, a series of solutions of decreasing cost is returned over time. The weight $w$ is initially set to a high value and then decremented by $\delta$ after each solution. If allowed to continue, $w$ eventually reaches 1 and the cheapest path is discovered. Of course, finding the optimal solution this way takes longer than simply running A* directly.

These algorithms suffer from two inherent difficulties. First, it is not well understood how to set $w$ or $\delta$ to best satisfy the user's needs. Setting $w$ too high or $\delta$ too low can result in many poor-quality solutions being returned, wasting time. But if $w$ is set too low or $\delta$ too high, the algorithm may take a very long time to find a solution. Therefore, to use a weighted A* technique like ARA* the user must perform many pilot experiments in each new problem domain to find good parameter settings.

Second, for anytime algorithms such as ARA*, the user must estimate the right time to stop the algorithm. The search process appears as a black box that could emit a significantly better solution at any moment, so one must repeatedly estimate the probability that continuing the computation will be worthwhile according to the user's utility function. This requires substantial prior statistical knowledge of the run-time performance profile of the algorithm and rests on the assumption that such learned knowledge applies to the current instance.

These difficulties point to a more general problem: anytime algorithms must inherently provide suboptimal performance due to their ignorance of the user's utility function. It is simply not possible in general for an algorithm to quickly transform the best solution achievable from scratch in time $t$ into the best solution achievable in time $t + 1$. In the worst case, visiting the next-most-promising solution might require starting back at a child of the root node. Without the ability to decide during the search whether a distant solution is worth the expected effort of reaching it, anytime algorithms must be manually engineered according to a policy fixed in advance. Such hardcoded policies mean that there will inevitably be situations in which anytime algorithms will either waste time finding nearby poor-quality solutions or overexert themselves finding a very high quality solution when any would have sufficed.

In this paper we address the fundamental issue: knowledge of the user's utility function. We propose a simple variant of best-first search that represents the user's desires and uses an estimate of this utility as guidance. We call the approach BUGSY (Best-first Utility-Guided Search—Yes!) and show empirically across several domains that it can successfully adapt its behavior to suit the user, sometimes significantly outperforming anytime algorithms. Furthermore, this utility-based methodology is easy to apply, requiring no performance profiling.

## 2 The BUGSY Approach

Ideally, a rational search agent would evaluate the utility to be gained by each possible node expansion. The utility of an expansion is equal to the utility of the eventual outcomes enabled by that expansion, namely the solutions lying below that node. For instance, if there is only one solution in a tree-structured space, expanding any node other than the one it lies beneath has no utility (or negative utility if time is costly). We will approximate these true utilities by assuming that the utility of an expansion is merely the utility of the highest-utility solution lying below that node.

We will further assume that the user's utility function can be captured in a simple linear form. If $f(s)$ represents the cost of solution $s$, and $t(s)$ represents the time at which it is returned to the user, then we expect the user to supply three constants: $U_{default}$, representing the utility of returning an empty solution; $w_f$, representing the importance of solution quality; and $w_t$, representing the importance of computation time. The utility of expanding node $n$ is then computed as

$$U(n) = U_{default} - \min_{s \text{ under } n} (w_f \cdot f(s) + w_t \cdot t(s))$$

where $s$ ranges over the possible solutions available under $n$. (Note that we follow the decision-theoretic tradition of better utilities being more positive, requiring us to subtract the estimated solution cost $f(s)$ and search time $t(s)$.) This formulation allows us to express exclusive attention to either cost or time, or any linear trade-off between them. The number of time units that the user is willing to spend to achieve an improvement of one cost unit is $w_f/w_t$. This quantity is usually easily elicited from users if it is not already explicit in the application domain. (The utility function would also be necessary when constructing the termination policy for an anytime algorithm.) Although superficially similar to weighted A*, BUGSY's node evaluation function differs because $w_f$ is applied to both $g(n)$ and $h(n)$.

Of course, the solutions $s$ available under a node are unknown, but we can estimate some of their utilities by using functions analogous to the traditional heuristic function $h(n)$. Instead of merely computing a lower bound on the cost of the cheapest solution under a node, we also compute the lower bound on distance in search nodes to that hypothetical cheapest solution. In many domains, this additional estimate entails only trivial modifications to the usual $h$ function. Search distance can then be multiplied by an estimate of time per expansion to arrive at $t(s)$. (Note that this simple estimation method makes the standard assumption of constant time per node expansion.) To provide a more informed estimate, we can also compute bounds on the cost and time to the nearest solution in addition to the cheapest. $U(n)$ can then be estimated as the maximum of the two utilities. For convenience, we will also notate by $f(n)$ and $t(n)$ the values inherited from whichever hypothesized solution had the higher utility.

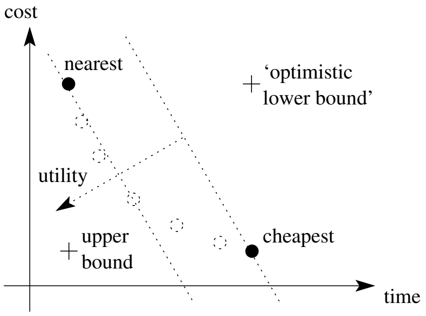


Figure 1: Estimating utility using the maximum of bounds on the nearest and cheapest solutions.

Figure 1 illustrates this process. The two solid dots represent the solutions hypothesized by the cheapest and nearest heuristic functions. The dashed circles represent hypothetical solutions representing a trade-off between those two extremes. The dotted lines represent contours of constant utility and the dotted arrow shows the direction of the utility gradient. Assuming that the two solid dots represent lower bounds, then an upper bound on utility would combine the cost of the cheapest solution with the time to the nearest solution. However, this is probably a significant overestimate. Taking the time of the cheapest and the cost of the nearest is not a true lower bound on utility because the two hypothesized solutions are themselves lower bounds and might in reality lie further toward the top and right of the figure. Note that under different utility functions (different slopes for the dotted lines) the relative superiority of the nearest and cheapest solutions can change.

### 2.1 Implementation

Figure 2 gives a pseudo-code sketch of a BUGSY implementation. The algorithm closely follows a standard best-first search. $U(n)$ is an estimate, not a lower bound, so it can overestimate or change arbitrarily along a path. This implies that we might discover a better route to a previously expanded state. Duplicate paths to the same search state are detected in steps 7 and 10; only the cheaper path is retained. We record links to a node's children as well as the preferred parent so that the utility of descendants can be recomputed (step 9) if

```
BUGSY(initial, $U()$)
1. open ← {initial}, closed ← {}
2. $n$ ← remove node from open with highest $U(n)$ value
3. if $n$ is a goal, return it
4. add $n$ to closed
5. for each of $n$'s children $c$,
6.     if $c$ is not a goal and $U(c) < 0$, skip $c$
7.     if an old version of $c$ is in closed,
8.         if $c$ is better than $c_{old}$,
9.             update $c_{old}$ and its children
10.    else, if an old version of $c$ is in open,
11.        if $c$ is better than $c_{old}$,
12.            update $c_{old}$
13.    else, add $c$ to open
14. go to step 2
```

Figure 2: BUGSY follows the outline of best-first search.

$g(n)$ changes [Nilsson, 1980, p. 66]. The on-line estimation of time per expansion has been omitted for clarity. The exact ordering function used for *open* (and to determine 'better' in steps 8 and 11) prefers high $U(n)$ values, breaking ties for low $t(n)$, breaking ties for low $f(n)$, breaking ties for high $g(n)$. Note that the linear formulation of utility means that *open* need not be resorted as time passes because all nodes lose utility at the same constant rate independent of their estimated solution cost. In effect, utilities are stored independent of the search time so far.

The $h(n)$ and $t(n)$ functions used by BUGSY do not have to be lower bounds. BUGSY requires estimates—there is no admissibility requirement. If one has data from previous runs on similar problems, this information can be used to convert standard lower bounds into estimates [Russell and Wefald, 1991]. In the experiments reported below, we eschew the assumption that training data is available and compute corrections on-line. We keep a running average of the one-step error in the cost-to-go and distance-to-go, measured at each node generation. These errors are computed by comparing the cost-to-go and distance-to-go of a node with those of its children. If the cost-to-go has not decreased by the cost of the operator used to generate the child, we can conclude that the parent's value was too low and record the discrepancy as an error. Similarly, the distance-to-go should have decreased by one. These correction factors are then used when computing a node's utility to give a more accurate estimate based on the experience during the search so far. Given the raw cost-to-go value $h$ and distance-to-go value $d$ and average errors $e_h$ and $e_d$, $d' = d(1 + e_d)$ and $h' = h + d'e_h$. Because on-line estimation of the time per expansion and the cost and distance corrections create additional overhead for BUGSY relative to other search algorithms, we will take care to measure CPU time in our experimental evaluation, not just node generations.

### 2.2 Properties of the Algorithm

BUGSY is trivially sound—it only returns nodes that are goals. If the heuristic and distance functions are used without inadmissible corrections, then the algorithm is also complete if the search space is finite. If $w_t = 0$ and $w_f > 0$, BUGSY reduces to A*, returning the cheapest solution. If $w_f = 0$ and $w_t > 0$, then BUGSY is greedy on $t(n)$. Ties will be broken on low $f(n)$, so a longer route to a previously visited state will be discarded. This limits the size of *open* to the size of the search space, implying that a solution will eventually be discovered. Similarly, if both $w_f$ and $w_t > 0$, BUGSY is complete because $t(n)$ is static at every state. The $f(n)$ term in $U(n)$ will then cause a longer path to any previously visited state to be discarded, bounding the search space and ensuring completeness. Unfortunately, if the search space is infinite and $w_t > 0$, BUGSY is not complete because a pathological $t(n)$ can potentially mislead the search forever.

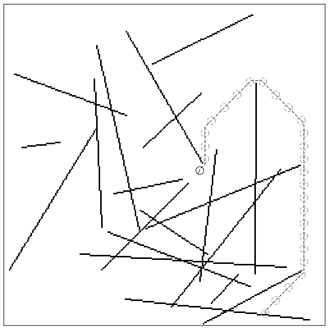
If the utility estimates $U(n)$ are perfect, BUGSY is optimal. This follows because it will proceed directly to the highest-utility solution. Assuming $U(n)$ is perfect, when BUGSY expands the start node the child node on the path to the highest utility solution will be put at the front of the open list. BUGSY will expand this node next. One of the children of this node must have the highest utility on the open list since it is one step closer to the goal than its parent, which previously had the highest utility, and it leads to a solution of the same quality. In this way, BUGSY proceeds directly to the highest utility solution achievable from the start state. It incurs no loss in utility due to wasted time since it only expands nodes on the path to the optimal solution.

It seems intuitive that BUGSY might have application in problems where operators have different costs and hence the distance to a goal in the search space might not correspond directly to its cost. But even in a search space in which all operators have unit cost (and hence the nearest and cheapest heuristics are the same), BUGSY can make different choices than A*. Consider a situation in which, after several expansions, it appears that node A, although closer to a goal than node B, might result in a worse overall solution. (Such a situation can easily come about even with an admissible and consistent heuristic function.) If time is weighted more heavily than solution cost, BUGSY will expand node A in an attempt to capitalize on previous search effort and reach a goal quickly. A*, on the other hand, will always abandon that search path and expand node B in a dogged attempt to optimize solution cost regardless of time.

In domains in which the cost-to-goal and distance-to-goal functions are different, BUGSY can have a significant advantage over weighted A*. With a very high weight, weighted A* will find a solution only as quickly as the greedy algorithm. BUGSY however, because its search is guided by an estimate of the distance to solutions as well as their cost, can actually find a solution in less time than the greedy algorithm.

## 3 Empirical Evaluation

To determine whether such a simple mechanism for time-aware search can be effective in practice with imperfect estimates of utility, we compared BUGSY against seven other algorithms on three different domains: gridworld path planning (12 different varieties), dynamic robot motion planning (used by Likhachev *et al.* [2004] to evaluate ARA*), and multiple sequence alignment (used by Zhou and Hansen [2002] to



```
# # ##  #
  oooooooo
  o #    o
  o    # o
  o  #   o
##o#    #o
Soo#     G
```

```
jhgiggh
j-grmgg
o-grm-o
```

Figure 3: Examples of the test domains: dynamic motion planning (left), gridworld planning (top right), and multiple sequence alignment (bottom right).

evaluate Anytime A*). All algorithms were coded in Objective Caml, compiled to native code, and run on one processor of a dual 2.6GHz Xeon machine with 2Gb RAM, measuring CPU time used. The algorithms were:

**A*** detecting duplicates using a closed list, breaking ties on $f$ in favor of high $g$,

**weighted A*** with $w = 3$,

**greedy** A* but preferring low $h$, breaking ties on low $g$,

**speedy** greedy but preferring low time to goal ($t(n)$), breaking ties on low $h$, then low $g$,

**Anytime A*** weighted A* ($w = 3$) that continues, pruning the open list, until an optimal goal has been found,

**ARA*** performs a series of weighted A* searches (starting with $w = 3$), decrementing the weight ($\delta = 0.2$, following Likhachev et al.) and reusing search effort,

**$A^*_\epsilon$** from among those nodes within a factor of $\epsilon$ (3) of the lowest $f$ value in the open list, expands the one estimated to be closest to the goal.

Note that greedy, speedy, and A* do not provide any inherent mechanism for adjusting their built-in trade-off of solution cost against search time; they are included only to provide a frame of reference for the other algorithms. The first solution found by Anytime A* and ARA* is the same one found by weighted A*, so those algorithms should do at least as well. We confirmed this experimentally, and omit weighted A* from our presentation below. On domains with many solutions, Anytime A* often reported thousands of solutions; we therefore limited both anytime algorithms to only reporting solutions that improve solution quality by at least 0.1%. $A^*_\epsilon$ performed very poorly in our preliminary tests, taking a very long time, so we omit its results as well. [1]

## 3.1 Dynamic Robot Motion Planning

Following Likhachev *et al.* [2004], this domain involves motion planning for a mobile robot (see Figure 3 for an example). Rather than finding the shortest path, the objective is to find the fastest path, taking into account the maximum acceleration of the robot and its inability to turn quickly at high speed. Solution cost corresponds to the duration of the planned robot trajectory. In effect, each utility function specifies a different trade-off between planning time and plan execution time. The state representation records position, heading, and speed. The path cost heuristic ($h(n)$) is simply the shortest path distance to the goal, divided by the maximum speed. This is precomputed to all cells at the start of the search. The plan cost lower bound $f(n)$ is the usual cost-so-far ($g(n)$) plus this cost-to-go ($h(n)$). For speedy and BUGSY, the distance in moves to the goal is also precomputed. The search cost estimate $t(n)$ is this distance divided by the number of search nodes expanded per second, which was estimated on-line as discussed above. No separate estimates were made for BUGSY of the distance to the cheapest goal or cost of the nearest goal, so $U$ was estimated only on this single $f$ and $t$ values. Legal state transitions (ignoring position) were precomputed. Unlike the heuristics, this was the same for all algorithms and was not included in the search time. We used 20 worlds 100 by 100 meters (discretized as in Likhachev et al. every 0.4 meters), each with 20 linear obstacles placed at random. Starting and goal positions and headings were selected uniformly at random. Instances that were solved by A* in less than 10 seconds or more than 1000 seconds were replaced.

| $U()$ | BUGSY | ARA* | Sp | Gr |
|---|---|---|---|---|
| time only | 72 | 66 | 75 | 88 |
| 10 microsec | 72 | 66 | 75 | 88 |
| 100 microsec | 69 | 66 | 74 | 88 |
| 1 msec | 58 | 63 | 70 | 83 |
| 10 msec | 51 | 47 | 47 | 56 |
| 0.1 sec | 66 | 59 | 53 | 55 |
| 1 sec | 69 | 65 | 56 | 56 |
| 10 secs | 67 | 69 | 53 | 54 |
| 100 secs | 67 | 69 | 53 | 53 |

Table 1: Results on dynamic robot motion planning.

Table 1 compares the solutions obtained by each algorithm under a range of different possible utility functions. Each row of the table corresponds to a different utility function. Recall that each utility function is a weighted combination of path cost and CPU time taken to find it. The relative size of the weights determines how important time is relative to cost. In other words, the utility function specifies the maximum amount of time that should be spent to gain an improvement of 1 cost unit. This is the time that is listed under U() for each row in the table. For example, ”1 msec” means that a solution that takes 0.001 seconds longer to find than another must be at least 1 unit cheaper to be judged superior. The utility functions tested range over several orders of magnitude from one in which only search time matters to one in which 100 seconds can be spent to obtain a one unit improvement in the solution cost.

Recall that, given a utility function at the start of its search, BUGSY returns a single solution representing the best trade-

[1] Although Pearl and Kim do not discuss implementation techniques (their results are presented solely in terms of node expansions), it seems that their algorithm could be made to operate more efficiently by designing a special coordinated heap and balanced binary tree data structure. We have not pursued this yet.

off of path cost and search time that it could find based on the information available to it. Of course, the CPU time taken is recorded along with the solution cost. Greedy (notated Gr in the table) and speedy (notated Sp) also each return one solution. These solutions may score well according to utility functions with extreme emphasis on time but may well score poorly in general. The two anytime algorithms, Anytime A* and ARA*, return a stream of solutions over time. For these experiments, we allowed them to run to optimality and then, for each utility function, post-processed the results to find the optimal cut-off time to optimize each algorithm's performance for that utility function. Note that this 'clairvoyant termination policy' gives Anytime A* and ARA* an unrealistic advantage in our tests. However, both A* and Anytime A* performed extremely poorly in this domain and are omitted from Table 1. To compare more easily across different utility functions, all of the resulting solution utilities were linearly scaled to fall between 0 and 100. Each cell in the table is the mean across 20 instances.

The results suggest that Bugsy is competitive with or better than ARA* on all but perhaps one of the utility functions. In general, Bugsy seems to offer a slight advantage when time is important. Given that Bugsy does not require performance profiling to construct a termination policy, this is encouraging performance. As one might expect, Greedy performs well when time is very important, however as cost becomes important the greedy solution is less useful. Compared to greedy, speedy is not able to overcome the overhead of computing two node evaluation functions.

### 3.2 Gridworld Planning

We considered several classes of path planning problems on a 500 by 300 grid, using either 4-way or 8-way movement, three different probabilities of blocked cells, and two different cost functions. In addition to unit costs, under which every move is equally expensive, we used a graduated cost function in which moves along the upper row are free and the cost goes up by one for each lower row. Figure 3 shows a small example solution under these costs (the start and goal positions are always in these corners). We call this cost function 'life' because it shares with everyday living the property that a short direct solution that can be found quickly (shallow in the search tree) is relatively expensive while a least-cost solution plan involves many annoying economizing steps. Under both cost functions, simple analytical lower bounds (ignoring obstacles) are available for the cost ($g(n)$) and distance (in search steps) to the cheapest goal and to the nearest goal. These quantities are then used to compute the $f(n)$ and $t(n)$ estimates. Because A* can perform well in this domain and our experiments include utility functions that make it worth finding the optimal solution, we diluted Bugsy's estimated lower-bound correction factors by dividing them by 5, decreasing the severity of any overestimation.

| $U()$ | Bugsy | ARA* | AA* | Sp | Gr | A* |
|---|---|---|---|---|---|---|
| | *unit costs, 8-way movement, 40% blocked* | | | | | |
| time only | 99 | 100 | 99 | 99 | 100 | 69 |
| 500 microsec | 98 | 96 | 96 | 95 | 95 | 69 |
| 1 msec | 98 | 91 | 93 | 90 | 91 | 69 |
| 5 msec | 95 | 60 | 68 | 56 | 56 | 68 |
| 10 msec | 94 | 44 | 57 | 34 | 34 | 74 |
| 50 msec | 95 | 85 | 77 | 33 | 33 | 91 |
| cost only | 95 | 96 | 96 | 33 | 33 | 96 |
| | *unit costs, 4-way movement, 20% blocked* | | | | | |
| time only | 97 | 98 | 98 | 98 | 99 | 19 |
| 100 microsec | 95 | 94 | 95 | 94 | 95 | 21 |
| 500 microsec | 91 | 67 | 70 | 61 | 62 | 28 |
| 1 msec | 86 | 62 | 43 | 28 | 29 | 50 |
| 5 msec | 82 | 81 | 42 | 22 | 22 | 91 |
| 10 msec | 79 | 87 | 46 | 20 | 20 | 92 |
| cost only | 76 | 93 | 93 | 19 | 19 | 93 |
| | *'life' costs, 4-way movement, 20% blocked* | | | | | |
| time only | 99 | 92 | 88 | 100 | 96 | 16 |
| 1 microsec | 97 | 94 | 90 | 93 | 98 | 17 |
| 5 microsec | 92 | 89 | 85 | 52 | 92 | 18 |
| 10 microsec | 93 | 86 | 83 | 12 | 88 | 30 |
| 50 microsec | 97 | 86 | 87 | 11 | 85 | 87 |
| 100 microsec | 97 | 91 | 89 | 11 | 85 | 94 |
| cost only | 94 | 97 | 97 | 11 | 82 | 97 |

Table 2: Results on three varieties of gridworld planning.

Table 2 shows typical results from three representative classes of gridworld problems. As before, the rows represent a broad spectrum of utility functions, including those in which speedy and A* are each designed to be optimal. Each value represents the mean over 20 instances. Anytime A* is notated AA*. In the top group (unit costs, 8-way movement, 40% blocked), we see Bugsy performing very well, behaving like speedy and greedy when time is important, like A* when cost is important, and significantly surpassing all the algorithms for the middle range of utility functions. In the next group (4-way movement, 20% blocked), Bugsy performs very well as long as time has some importance, again dominating in the middle range of utility functions where balancing time and cost is crucial. However, its inadmissible heuristic means that it cannot perform quite as well as A* or ARA* at the edge of the spectrum when cost becomes critical. (Of course, one can always disable Bugsy's correction factors when running under such circumstances, but presumably in practice one would be using A* search anyway if search time weren't an important consideration.) In the bottom group in the table ('life' costs, 4-way movement, 20% blocked), we see a similar general pattern: Bugsy performs very well across a wide range of utility functions, dominating other algorithms for the middle range of utility functions. However, it does fall slightly short of A* when solution cost is the only criterion.

### 3.3 Multiple Sequence Alignment

Alignment of multiple strings has recently been a popular domain for heuristic search algorithms [Hohwald *et al.*, 2003]. An example alignment is shown in Figure 3. The state representation is the number of characters consumed so far from each string; a goal is reached when all characters are consumed. Moves that consume from only some of the strings represent the insertion of a 'gap' character into the others. We computed alignments of three sequences at a time, using the

| $U()$ | Bugsy | ARA* | AA* | Sp | Gr | A* |
|---|---|---|---|---|---|---|
| time only | 99 | 100 | 100 | 100 | 100 | 22 |
| 1 msec | 100 | 99 | 99 | 97 | 98 | 22 |
| 5 msec | 99 | 97 | 97 | 88 | 92 | 24 |
| 10 msec | 98 | 92 | 94 | 74 | 83 | 26 |
| 50 msec | 87 | 80 | 81 | 14 | 42 | 90 |
| 0.1 sec | 69 | 89 | 68 | 11 | 33 | 93 |
| cost only | 57 | 95 | 95 | 9 | 27 | 95 |

Table 3: Results on multiple sequence alignment.

standard 'sum-of-pairs' cost function in which a gap costs 2, a substitution (mismatched non-gap characters) costs 1, and costs are computed by summing all the pairwise alignments. Sequences were over 20 characters, representing amino acid triplets. The uniform random sequences that are popular benchmarks for optimal alignment algorithms are not suitable in our setting because the solution found by the speedy algorithm (merely traversing the diagonal, resulting in many substitutions) is very often the optimal alignment. Instead, we use biologically-inspired benchmarks which encourage optimal solutions that contain significant numbers of gaps and matches. Starting from a 'common ancestor' string which does not become part of the instance, we create sequences by deleting and substituting characters uniformly at random. In the instances used below, the ancestors were 1000 characters long and the probabilities of deletion and substitution were both 0.25 at each position. The heuristic function $h(n)$ was based on optimal pairwise alignments that were precomputed by dynamic programming. The lower bound on search nodes to go was simply the maximum number of characters remaining in any sequence. As in gridworld, A* is a feasible algorithm and thus we dilute Bugsy's correction factors by 5.

Table 3 shows the results, with each row representing a different utility function and all raw scores again normalized between 0 and 100. Each cells represents the mean over 5 instances (there was little variance in the scores in this domain). Again we see the same pattern of performance. Bugsy performs very well when time is important and surpasses the other algorithms when balancing between cost and time. It does fall short of A* when cost is paramount, due to its inadmissible heuristic.

## 4 Discussion

We have presented empirical results, using actual CPU time measurements and a variety of search problems, demonstrating that Bugsy is at least competitive with state-of-the-art anytime algorithms. For utility functions with an emphasis on solution time or on balancing time and cost, it often performs significantly better than any previous method. However, for utility functions based heavily on solution cost it can sometimes perform worse than A*. Bugsy appears quite robust across different domains and utility functions.

When its utility estimates are perfect, Bugsy is optimal. However, more work remains to understand the exact tradeoff between accuracy and admissibility. Our empirical experience demonstrates that attempting to correct lower bounds into more accurate estimators can impair Bugsy's performance when solution quality is very important. However, it seems foolish not to take advantage of on-line error estimation to bring these bounds closer to the accurate estimates that would allow Bugsy to be optimal. In this paper, we have chosen to merely dilute the correction factors. In the future, we hope to be able to analyze the given utility function in the context of the domain and determine whether admissibility is worth preserving.

We have done preliminary experiments incorporating simple deadlines into Bugsy, with encouraging results. Because it estimates the search time-to-go, it can effectively prune solutions that lie beyond a search time deadline. Another similar extension applies to temporal planning: one can specify a bound on the sum of the search time and the resulting plan's execution time and let Bugsy determine how to allocate the time.

Note that Bugsy solves a different problem than Real-Time A* [Korf, 1990] and its variants. Rather than performing a time-limited search for the first step in a plan, Bugsy tries to find a complete plan to a goal in limited time. This is particularly useful in domains in which operators are not invertible or are otherwise costly to undo. Having a complete path to a goal ensures that execution does not become ensnared in a deadend. It is also a common requirement when planning is but the first step in a series of computations that might further refine the action sequence.

In some applications of best-first search, memory use is a prominent concern. In a time-bounded setting this is less frequently a problem because the search doesn't have time to exhaust available memory. However, the simplicity of Bugsy means that it may well be possible to integrate some of the techniques that have been developed to reduce the memory consumption of best-first search if necessary.

When planning in a dynamic environment, we assume not only that Bugsy is provided with a utility function that captures the decrease in expected plan value as a linear function of time, but also that the algorithm has full access to knowledge of how the domain changes. It would be very interesting to combine the utility-based search of Bugsy with techniques to exploit localized changes in the search space, such as used in ARA*.

## 5 Conclusions

As Nilsson notes, "in most practical problems we are interested in minimizing some *combination* of the cost of the path and the cost of the search required to obtain the path" yet "combination costs are never actually computed ... because it is difficult to decide on the way to combine path cost and search-effort cost" [1971, p. 54, emphasis his]. Bugsy addresses this problem by letting the user specify how path cost and search cost should be combined.

This new approach provides an alternative to anytime algorithms. Instead of returning a stream of solutions and relying on an external process to decide when additional search effort is no longer justified, the search process itself makes such judgments based on the node evaluations available to it. Our empirical results demonstrate that Bugsy provides a simple and effective way to solve shortest-path problems when computation time matters. We would suggest that search procedures are usefully thought of not as black boxes to be controlled by an external termination policy but as complete intelligent agents, informed of the user's goals and acting on the information they collect so as to directly maximize the user's utility.

## References

[Dechter and Pearl, 1988] Rina Dechter and Judea Pearl. The optimality of A*. In Laveen Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence*, pages 166–199. Springer-Verlag, 1988.

[Hansen *et al.*, 1997] Eric A. Hansen, Shlomo Zilberstein, and Victor A. Danilchenko. Anytime heuristic search: First results. CMPSCI 97-50, University of Massachusetts, Amherst, September 1997.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, SSC-4(2):100–107, July 1968.

[Hohwald *et al.*, 2003] Heath Hohwald, Ignacio Thayer, and Richard E. Korf. Comparing best-first search and dynamic programming for optimal multiple sequence alignment. In *Proceedings of IJCAI-03*, pages 1239–1245, 2003.

[Korf, 1990] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[Likhachev *et al.*, 2004] Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of NIPS 16*, 2004.

[Nilsson, 1971] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.

[Nilsson, 1980] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co, 1980.

[Pearl and Kim, 1982] Judea Pearl and Jin H. Kim. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(4):391–399, July 1982.

[Pohl, 1970] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1:193–204, 1970.

[Russell and Wefald, 1991] Stuart Russell and Eric Wefald. *Do the Right Thing: Studies in Limited Rationality*. MIT Press, 1991.

[Zhou and Hansen, 2002] Rong Zhou and Eric A. Hansen. Multiple sequence alignment using anytime A*. In *Proceedings of AAAI-02*, pages 975–976, 2002.

# Heuristic Speed-Ups for Learning in Complex Stochastic Environments

**Christian J. Darken**
MOVES Institute
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

## Abstract

We describe a novel methodology by which a software agent can learn to predict future events in complex stochastic environments together with an important heuristic-based acceleration technique for computing the prediction. This speed-up enables us to use much more context in our predictions than was previously possible [Darken, 2005]. We present results gathered from a first prototype of our approach.

## 1 Introduction

A significant challenge for intelligent software agents is making them proactive, i.e. able to understand their environment to the degree that they are able to predict what is likely to happen next and can therefore take appropriate measures. The ability to predict likely next events can in principle be converted into intelligent action selection along the lines suggested by [Sutton and Barto, 1981]. We propose that simple, transparent learning schemes may enable agents to predict the likely course of events. The prediction algorithm has been previously described in [Darken, 2005], but the acceleration techniques and results they enable are new.

In order to explore our hypothesis, we have created a simple game in the RPG (role-playing game) family. We then implemented a sensory interface that passes percepts coded in a first-order logic subset to the agent. The agent then attempts to predict the next percept that it will see. This environment is both stochastic and complex. ”Stochastic” implies that future percepts are not a function of the sequence of previous ones. This environment may be considered complex in many senses, beginning with the fact that, although it is a small and simple game as such games go, its state space is very large. More significant, we believe, is the fact that there is no obvious way for the agent to sum up its information about the world in a representation of fixed dimension, i.e. that some aspects of first-order logic are apparently needed in order to accomplish the task. Our impression is that learning algorithms that can succeed in stochastic domains without obvious representations of fixed dimension are of interest for many domains stretching far beyond interactive entertainment applications.

## 2 Related Work

Anticipation of hostile unit behavior in the context of computer games has previously been addressed in [Laird, 2001], who had the agent apply its own action selection procedure based on the information probably possessed by the hostile unit in order to guess what the hostile would do. In this work, we are attempting to learn to anticipate without hand-coded rules. Further, while hostile unit behavior is one of the things we would like to predict, it is not the only thing.

We have not been successful in finding known algorithms that we can productively compare with our approach. Logical rules, including some types of predictive rules, can be learned by algorithms such as FOIL [Mitchell, 1997]. However, these algorithms assume a deterministic domain. Hidden Markov Models [R. Duda and Stork, 2001] are well suited to stochastic domains, but assume a finite state space, and in practice state spaces that are finite but large are problematic. We are more optimistic about the scaling of variable order Markov models [R. Begleiter and Yona, 2004], but these also assume a finite state space.

After submitting this paper, the reviewers suggested that several recent models may be related to the one presented in this paper. We have not been able to follow up these suggestions in as much detail as we would have liked, but we offer the following preliminary comments. Predictive State Representations [Singh *et al.*, 2003] and Schema Learning [Holmes, 2005] are recent approaches to prediction in stochastic environments. Both are focused on predicting the results of agent actions. We believe both approaches are currently limited to finite state spaces, and we are aware of tests on only very small domains (tens of states). Relational Reinforcement Learning (for example, [Gretton and Thiebaux, 2004]) also considers relational, stochastic domains like the approach described in this work, though it appears to be focused on action selection (as is conventional reinforcement learning) rather than prediction.

## 3 Benchmark Environment

Our benchmark environment is a simple virtual environment with a text interface modeled after the DikuMUD family of combat oriented MUD’s. This family of games is instantly comprehensible to a player of World of Warcraft or Everquest 2, to name two current exemplars, and is arguably a progen-

```
Paperville
Terrified eyes peer from every window
  of this besieged hamlet.
Contents: pitchfork, wand, Conan

get pitchfork

You get the pitchfork.

equip pitchfork

You equip the pitchfork.

w

The Eastern Meadow
All the grass has been trampled into
  the dirt, and tiny footprints are
  everywhere.
Contents: Conan
```

Figure 1: The beginning of a session with the benchmark environment as it appears to a human player named Conan.

itor of these systems. Players of this type of game assume the role of a young adventurer. The goal of the game is to expand the power of one's in-game avatar to the maximum extent possible. This goal is primarily accomplished by slaying the monsters that roam the virtual environment. Slaying monsters results in improvements to the avatar's capabilities through an abstracted model of learning ("experience points") and also though the items ("loot") that the slain monsters drop or guard, which either consist of or may be traded for more powerful combat gear.

The benchmark environment consists of 19 locations, four monsters of three different types, and four different weapon types, of which there may be any number of instantiations. Growth of combat capabilities through experience has not been modeled, therefore, improved capability comes only by acquiring more powerful weapons. The environment as a whole may be conceived of as a discrete event system with a state that consists of the Cartesian product of some number of variables. The system remains in a state indefinitely until an event is received, at which time it may transition to a new state.

The benchmark environment together with networking and multiplayer infrastructure was coded from scratch in Python. The system uses a LambdaMOO-like method dispatch mechanism to determine which game object should process a player action. An unusual feature is the ability to provide output in English text and/or in a first-order logic fragment, as shown in Figures 1 and 2.

## 4 Perceptual Model

We have implemented text-based interfaces to allow both humans and software agents to interact with the benchmark environment. The human interface consists of English text. We describe the agent interface below.

```
(A 40.6979999542 look)
(+ 40.7079999447 location pitchfork
  Paperville)
(+ 40.7079999447 location wand
  Paperville)
(+ 40.7079999447 location Conan
  Paperville)

get pitchfork

(A 44.6440000534 get pitchfork)
(E 44.6440000534 get Conan pitchfork)
(- 44.6440000534 location pitchfork
  Paperville)
(+ 44.6440000534 location pitchfork
  Conan)

equip pitchfork

(A 47.6080000401 equip pitchfork)
(+ 47.6080000401 equipping Conan
  pitchfork)

w

(A 51.2130000591 w)
(E 51.2130000591 go Conan west)
(- 51.2130000591 location wand
  Paperville)
(- 51.2130000591 location Conan
  Paperville)
(+ 51.2130000591 location Conan
  The_Eastern_Meadow)
```

Figure 2: The beginning of the same session described in Figure 1 with the benchmark environment as it would appear to a software agent named Conan. The first four percept fields are: type, time stamp, percept name. These are followed by the percept arguments, if any.

Perception for software agents in the benchmark environment is modeled as direct access to a subset state variables and system events. The subset of visible events and variables depends upon the location of the agent in the environment, i.e. an agent receives information only about occurrences in his immediate location. The agent's own actions also generate percepts. Thus, four types of percepts are required. 'A' represents agent actions. 'E' represents events. '+' represents the beginning of a time interval in which a variable was sensed to have a particular value. When the variable changes value, or it can no longer be sensed, a '-' percept is received. We form logical atoms from percepts whenever needed by appending the percept type to the percept name to create a predicate (i.e. a percept of type 'E' with name 'location' would correspond to an atom with predicate 'locationE') and taking the remaining elements of the percept as the arguments of the predicate (the time stamp is ignored). At any given time, we define the "sensation" of the agent to be the set of all variables and their values that are currently being sensed.

## 5 Prediction

After the agent is turned on for the first time, and percepts start to arrive, a percept predictor is constructed on the fly, i.e. the agent learns as it goes along, just like animals do. As each percept is received, the new data is used to enhance ("train") the predictor, and the enhanced predictor is immediately put to use to predict the next percept. Prediction depends upon a few key notions. The first is the notion of a "situation".

Our statistical one-step-ahead percept predictor is a function whose input is the percept sequence up to the time of prediction and whose output is a probability distribution over all percepts that represents the probability that each percept will be the next one in the percept sequence. Of course, all percepts in the percept sequence are not equally useful for prediction. In particular, one might expect that, as a general rule, more recent percepts would be more useful than older ones. On this basis, we discriminate the "relevant" subset of the percept sequence, and ignore the rest. We define a recency threshold $T$. For predictions at time $t$, a percept in the percept sequence is relevant if either its time-stamp is in the interval $[t - T, t]$, or it is a '+' type percept whose corresponding '-' percept has not yet been received (this would indicate that the contents of the percept are still actively sensed by the agent). Given the set of relevant percepts, we produce the multiset of relevant atoms (multisets are sets that allow multiple identical members, also known as bags) by stripping off the timestamps and appending the type to the predicate to produce a new predicate whose name reflects the type. We call these relevant atom multisets "situations".

Our predictor function takes the form of a table whose left column contains a specification of a subset of situations and whose right column contains a prescription for generating a predictive distribution over percepts given a situation in the subset. The table contains counters for the number of times each left column and right column distribution element is encountered. We have investigated two different methods of specifying subsets and generating the corresponding predictions.

### 5.1 Exact Matching

In this technique, each left column entry consists of a single situation. A new situation matches the entry only if it is identical (neglecting the order of the atoms). Each right column entry consists of a distribution of situations. If a new situation matches a left column entry, the predicted percept distribution is the list of atoms in the right-hand column together with probability estimates which are simply the value of the counter for the list member element divided by the value of the counter for the situation in the left column.

As each percept arrives, it is used to train the predictor function as follows. The situation as it was *at the time of the arrival of the last percept* is generated and matched against all entries in the left-hand column of the table. Because of how the table is constructed, it can match at most one. If a match is found, the counter for the entry is incremented. Then the new percept is matched against each element of the predicted percept distribution. If it matches, the counter for that element is incremented. If it fails to match any element of the distribution, it is added as a new element of the distribution with a new counter initialized to one. If the situation matches no left-hand column entry, a new entry is added.

Next, the current situation (including the percept that just arrived) is constructed and matched against the left-hand column entries to generate the predicted distribution for the next percept to arrive. If the situation does not match any entry, there is no prediction, i.e. the situation is completely novel to the agent.

An instructive example to illustrate the algorithm's function can be found in [Darken, 2005].

### 5.2 Patterns with Variables

The above technique makes predictions that are specific to specific objects in the environment. In environments where an object may be encountered only once and never again, for example, this is not very useful. By replacing references to specific objects by variables, we produce a technique that generalizes across objects. In this technique, left column entries contain variables instead of constants. A new situation matches the entry if there is a one-to-one substitution of the variables to constants in the situation. A one-to-one substitution is a list of bindings for the variables, that has the property that one and only one variable can be bound to one specific constant. The reason for the constraint to one-to-one substitutions is to ensure that each situation matches at most one pattern (left column entry). This restriction is not necessary, but it is convenient. Right column entries can also contain variables in this model. Given a match of a pattern to a situation, the predicted percept distribution is given by applying the substitution to the atoms in the right column distribution. Note that it may be the case that some variables remain in the prediction even after the substitution is applied.

As each percept arrives, it is used to train the predictor function as follows. The situation is generated and matched against all entries in the left-hand column of the table. It can match at most one. If a match is found, the substitution (list of variable-to-constant bindings) is kept, and the counter for the entry is incremented. Then the substitution is applied to each element of the predicted percept distribution, and the percept is matched against it. If it matches, the counter for that element is incremented. If it fails to match any element of the distribution, it is "variablized" by replacing each constant with the corresponding variable from the substitution, and replacing each remaining constant with a new variable, and then added as a new element of the distribution with a new counter initialized to one. If the situation matches no left-hand column entry, a new entry is added, consisting of the situation with each constant replaced by a variable.

Note that one can conceive of interesting schemes that are combinations of the two presented techniques. For example, one might try to predict the next percept with an exact matching model first, but if no prediction was available (or if the prediction was based on too little data), one might revert to a simultaneously developed variable-based predictor. Alternatively, one might design the environment so that percept references to objects were either existentially quantified variables or constants. A hybrid model could be developed which would then produce patterns with variables or constants based on what was present in the percept. This places the burden of deciding how the predictor should behave onto the percept designer.

# 6 Accelerated Search

Initially we implemented a back-tracking depth-first search to match situations to table entries. Using back-tracking search and progressing linearly through the predictor table proved too slow. We wanted to experiment with higher recency thresholds. But a higher recency threshold corresponds directly to larger situations, and a great deal more time performing backtracking search.

For the exact matching algorithm, it is the case that each situation corresponds to a unique string which is the constituent atoms (taken as lists of strings which are the predicate and constant arguments) put into lexical order. These strings are then placed in a hash table. Now a new situation can be tested against the table by constructing its string and checking the hash table.

For the variable pattern approach, simply sorting the atoms will not work, as they contain variables whose names are not significant. Our approach is to compute an invariant of the situation pattern that does not depend on the names of the variables. For each variable, we construct two lists of predicates, the list of predicates where the variable appears as the first argument and the list of predicates where the variable appears as the second. All of our predicates are binary. Were this not the case, more lists could be used, or the higher degree atoms reduced to a semantically equivalent set of binary atoms. We then put this list of list pairs into lexical order and then hash them into a table. Two situations that are identical up to substituting the names of variables must hash to the same location in the table. Unfortunately, situations that are different in more than just variable names can nonetheless hash to the same location, so a backtracking search must be performed on each situation in the hash cell to determine whether the match is genuine or not. Still, hash collisions occur relatively rarely, and this approach is very much faster than backtracking search over every row of the predictor table.

## 6.1 Example

The following example provides proof that the "list of list pairs" invariant, described above, is not sufficient to discriminate all situations that are legitimately different. Consider the following situation description, as might appear as a left column entry in the variable pattern method. Only one predicate, "P", is used.

```
P(?v,?w)
P(?w,?x)
P(?x,?y)
P(?z,?y)
```

Constructing the two lists for each variable as described above yields:

```
?v: [P] []
?w: [P] [P]
?x: [P] [P]
?y: []  [P P]
?z: [P] []
```

Here is a similar, yet different situation description.

```
P(?v,?w)
P(?w,?x)
P(?y,?x)
P(?z,?y)
```

And here is the corresponding list of list pairs.

```
?v: [P] []
?w: [P] [P]
?x: []  [P P]
?y: [P] [P]
?z: [P] []
```

After lexical sort, both cases become:

```
[]  [P P]
[P] []
[P] []
[P] [P]
[P] [P]
```

# 7 Results

## 7.1 First Experiment

We created a software agent that takes random actions (one every 0.25 seconds) and connected it to the benchmark environment. Since the action generator is not very intelligent, many actions elicit what are essentially error messages from the environment. We do not consider this a problem. In fact, we would like the agent to learn when an action will be fruitless.

We describe the results of a typical run. For this run, percepts were defined as relevant if they had been received in the last 0.1 seconds or if they were in the agent's current sensation. The agent was allowed to explore the environment for about one and one quarter hours of real time while the learning algorithm ran concurrently. 38519 percepts were received and processed during the run.

The exact matching approach produced 5695 predictors (rows in the table). The approach with variables produced only 952, much fewer, as might be expected.

Numeric results are given in Figures 3 and 4. The average predicted probability of the percepts as a function of time is presented in Figure 7. Note that by the end of the run, both curves are fairly flat. The exact match curve is lower, but increasing faster.

For the approach with variables, the prediction is considered correct if it matches the actual next percept (to within a one-to-one variable substitution). Note that the agent's own actions, being randomly generated, were the most difficult to predict. Neglecting type 'A' percepts, the average predicted probability of all remaining percepts is 66.6 percent for the exact match model and 70.5 percent for the model with variables. This strikes us as reasonably high given the fine-grained nature of the predictions, the simplicity of the algorithm and the high degree of remaining irreducible randomness in the environment caused by random movements of monsters and outcomes of each attempted strike in combat. A significant number of mistakes seemed to be caused by forgetting of important percepts caused by the severe recency threshold used (0.1 sec). We have found that the simple table-based predictive model does not scale well to the recency threshold of multiple seconds that would be seem to be necessary to solve the problem without modifying the agents perception to be more informative.

Detailed analysis of the top five types of errors for each algorithm shows that both algorithms are strongly impacted by the 0.1 sec recency threshold. The worst symptom is that the algorithms are unable to predict combat-related messages accurately because they can not tell that they are in combat. They can not tell that they are in combat because there is nothing in the sensation that indicates ongoing combat, and combat messages are spaced at intervals of one to two seconds.

For the exact matching algorithm, the most common errors stem from the simple fact that, being completely unable to generalize, many situations look completely novel, even at the end of the run. This difference can be clearly seen in the histograms of the last 5000 prediction probabilities presented as Figures 5 and 6. The exact match algorithm has more predictions with probability one than the variable-based algorithm, but it also has more with probability zero, indicating the absence of a match with any table entry.

The variable-based approach scored better that the exact matching algorithm overall. Nonetheless, the lack of predicates for indicating object type in the benchmark environment caused an interesting problem for this approach. For example, this approach was unable to predict the results of attempts to 'get X', and therefore had to hedge its bets between success and an error message. This was no issue for the exact match algorithm, as it could learn that 'get Troll' would provoke an error while 'get sword' would succeed. Note that the addition of a 'portable' predicate, for example, would mitigate this problem.

## 7.2 Second Experiment

In the second experiment, a fresh run of the agent was performed with the time between actions greatly increased (from

| Type | Avg. Probability | Occurrences | Error |
|---|---|---|---|
| A | 7.65% | 14488 | 65.5% |
| E | 72.09% | 14905 | 20.3% |
| + | 45.92% | 4563 | 12.1% |
| - | 69.28% | 4563 | 6.9% |

Figure 3: Performance summary for exact matching. The average predicated probability over all percepts was 44.43%. Error is the expected fraction of the total number of prediction errors for percepts of the given type.

| Type | Avg. Probability | Occurrences | Error |
|---|---|---|---|
| A | 7.82% | 14488 | 65.3% |
| E | 66.39% | 14905 | 24.5% |
| + | 65.12% | 4563 | 7.8% |
| - | 89.32% | 4563 | 2.4% |

Figure 4: Performance summary for patterns with variables. The average predicted probability over all percepts was 46.94%. Error is the expected fraction of the total number of prediction errors for percepts of the given type.

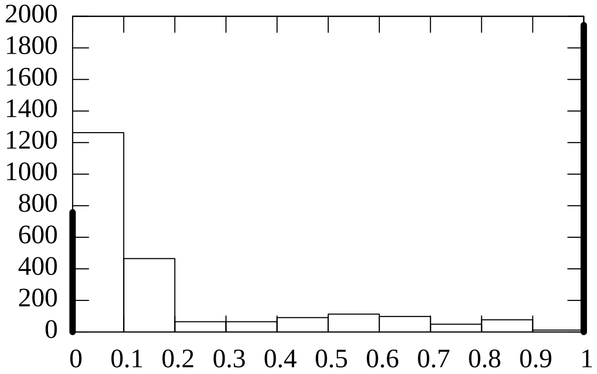


Figure 5: Prediction probability for the last 5000 percepts of the run with constants. The black bars represent the predictions of exactly 0 or 1.

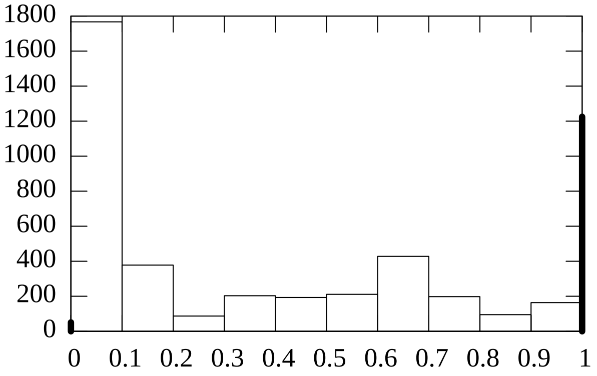


Figure 6: Prediction probability for the last 5000 percepts of the run with variables. The black bars represent the predictions of exactly 0 or 1.

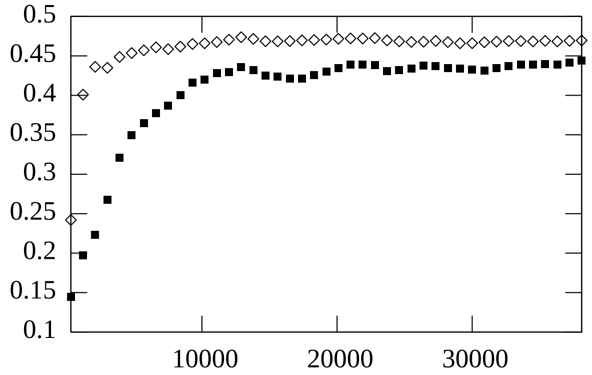


Figure 7: Average prediction probability as a function of the number of percepts received. White diamonds represent the algorithm with variables and black squares the algorithm with constants.

| | 0.1s | 2.1s |
|---|---|---|
| Exact Match | 62.2% | 57.3% |
| Variable Pattern | 64.8% | 62.3% |

Figure 8: Performance summary on the second experiment on all percepts except ’A’ type percepts.

0.25 seconds to 2.5 seconds between successive actions. The reason for the increase was because at 0.25 seconds per action and two seconds per combat round, the agent would attempt up to four actions in between successive combat “blows”. The combat messages were thus somewhat “buried”. This run was longer than that of the previous experiment. It consisted of 170762 percepts received over 38 hours of real time.

Using the acceleration techniques described above, we tested both 0.1 second and 2.1 second recency thresholds with both the exact match and variable pattern techniques. Results on all percepts excluding ’A’ type percepts are presented in Figure 8. As in the first experiment, the variable pattern approach performs better. The additional context provided by the higher recency threshold seems to hurt overall performance rather than helping. Apparently the extra information in the larger context is not enough to overcome the need for more training data. However, these results are very new, and we are still analyzing them in detail.

## 8 Discussion

A few comments on the structural characteristics of the methods presented in this paper are in order.

One very positive characteristic of them is that there is a clear “audit trail” that can be followed when the agent makes unexpected predictions. I.e. each row in the table can be traced to a specific set of prior experiences that are related to the predictions it makes in an obvious way. Many machine learning techniques do not share this characteristic.

Note that the situations in the left column of the table divide all possible percept sequences into a set of equivalence classes, i.e. many percept sequences can map into a single situation set. To the agent, only the sequence sets specified in the left column of the table matter. It will never be able to discriminate between different percept sequences that map into the same sequence set. The temptation naturally arises to make these sets as differentiated as possible by, for example, increasing the recency threshold or using exact matching instead of patterns with variables. But increasing the fineness of the situation sets is a two-edged sword. While it does indeed make it possible for the agent to discriminate between different percept sequences that it could not differentiate before, it also makes it increasingly rare that the agent visits situations that it knows about. Figures 5, 6, and 7 illustrate this fact.

## 9 Future Work

Although we have not discussed it previously, note that it is possible to extend the system as described to making predictions about *when* the next percept will be received in addition to what the next percept will be along the lines described in [Kunde and Darken, 2005].

A key direction for further investigation is improved predictive models and systematic exploitation of the predictions. The technique described in this work is very limited in its generalization capabilities. Unlike FOIL, which searches through candidate atoms and includes only the most promising in the model, the current approach takes all atoms that have passed the relevance test. It would be nice to have an approach that could perhaps learn from experience which of the relevant atoms were actually necessary to accurate prediction.

While we take for granted that many special-purpose schemes can be constructed which can improve agent behavior based on the ability to predict future percepts, it seems worth pointing out that one can search over the space of potential courses of action using the predictive model and a quality function to decide which course to adopt. This is a homogeneous and general-purpose method of exploiting prediction very similar in spirit to the model predictive control techniques that are an established part of chemical engineering [Morari and Lee, 1997]. It has been explored within the computer science literature as well [Sutton and Barto, 1981].

## 10 Acknowledgements

Partial funding for this work was provided by the U.S. Army TRADOC Analysis Center (TRAC) Monterey and the Naval Modeling and Simulation Management Office. The author wishes to thank the anonymous reviewers for very helpful references, comments and advice.

## References

[Darken, 2005] C. Darken. Towards learned anticipation in complex stochastic environments. In *Proc. Artificial Intelligence for Interactive Digital Entertainment 2005*, Marina Del Rey, CA, 2005.

[Gretton and Thiebaux, 2004] C. Gretton and S. Thiebaux. Exploiting first-order regression in inductive policy selection - extended abstract. In *Proc. of the ICML '04 Workshop on Relational Reinforcement Learning*, 2004.

[Holmes, 2005] M. Holmes. Schema learning: Experience-based construction of predictive action models. In *Neural Information Processing Systems*, 2005.

[Kunde and Darken, 2005] D. Kunde and C. Darken. Event prediction for modeling mental simulation in naturalistic decision making. In *Proc. BRIMS 2005*, Universal City, CA, 2005.

[Laird, 2001] J. Laird. It knows what you're going to do: adding anticipation to a quakebot. In Jörg P. Müller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 385–392, Montreal, Canada, 2001. ACM Press.

[Mitchell, 1997] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, Boston, 1997.

[Morari and Lee, 1997] M. Morari and J. Lee. Model predictive control: Past, present and future, 1997.

[R. Begleiter and Yona, 2004] R. El-Yaniv R. Begleiter and G. Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research (JAIR)*, 22:385–421, 2004.

[R. Duda and Stork, 2001] P. Hart R. Duda and D. Stork. *Pattern Classification*. John Wiley & Sons, New York, 2001.

[Singh *et al.*, 2003] S. Singh, M. Littman, N. Jong, D. Pardoe, and P. Stone. Learning predictive state representations. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML) 2003*, pages 99—106, 2003.

[Sutton and Barto, 1981] R. Sutton and A. Barto. An adaptive network that constructs and uses an internal model of its world. *Cognition and Brain Theory*, 4(3):217–246, 1981.

# Deduction and Exploratory Assessment of Partial Plans

**Jacek Malec** and **Sławomir Nowaczyk**
Jacek.Malec@cs.lth.se and Slawomir.Nowaczyk@cs.lth.se
Department of Computer Science, Lund University,
Box 118, 221 00 Lund, Sweden

## Abstract

In this paper we present a preliminary investigation of rational agents who, aware of their own limited mental resources, use learning to augment their reasoning. In our approach an agent creates and deductively reasons about possible plans of actions, but — aware of the fact that finding complete plans is in many cases intractable — it executes partial plans which look promising. By doing so, it can acquire new knowledge from results of performed actions, which allows it to plan further into the future in a more effective way.

We describe a possible application of Inductive Logic Programming to learn which of such partial plans are most likely to lead to reaching the goal. We also discuss how one can use ILP framework for generalising partial plans, thus allowing an agent to discover, after a number of episodes, a complete plan — or at least a good approximation of it.

## 1 Introduction

The basic idea of this project is to investigate a methodology for developing rational agents — both virtual and physical ones — that would be able to learn from experience, becoming more efficient at solving their tasks. A rational agent is expected to use deductive reasoning in order to take advantage of whatever domain knowledge it has been provided with. Besides that, it should perform inductive learning to benefit from experience it has gathered, correcting missing or inaccurate parts of that knowledge. Finally, it must acknowledge the fact that both reasoning and acting takes time, and try to balance those activities in a reasonable way.

In this paper we present how such rational agents can deal with planning in domains where complexity makes finding complete solutions intractable. Clearly, in many domains (especially those that are, at least from agent's point of view, nondeterministic) it is not realistic to expect an agent to be able to find a total plan which solves a problem at hand. Therefore, we investigate how an agent can create and reason about *partial plans*. By that we mean plans which bring it somewhat closer to achieving the goal, while still being simple and short enough to be computable in reasonable time. Currently we mainly focus on plans which allow an agent to acquire additional knowledge about the world.

By executing such "information-providing" partial plans, an agent can greatly simplify further planning process — it no longer needs to take into account the vast number of possible situations which will be inconsistent with newly observed state of the world. Thus, it can proceed further in a more effective way, by devoting its computational resources to more relevant issues.

We believe that the research field of planning has currently matured enough that it is time to explore new, more ambitious settings, in order to bring artificial agents closer to what humans are capable of. Our goal is to create an agent that is able to function in an adversary environment which it can only partially observe and which it only partially understands. Moreover, the agent is supposed to face a large number of episodes, learning from its mistakes and improving its efficiency.

We will base our examples on a simple game of Wumpus, a well-known test bed for intelligent agents, which is straightforward enough to properly illustrate our approach. In its basic form, the game takes place on a square board through which an agent is allowed to move. One square is inhabited by the Wumpus. Agent's goal is to kill the monster by shooting an arrow onto the square it occupies, while avoiding getting eaten by the monster. Luckily, Wumpus is a smelly creature, so the player always knows if the monster is on one of the squares adjacent to his current position — but unfortunately, not on which one. We leave the exact details of whether and how fast Wumpus can move open for now, since we will vary it in order to illustrate different ideas.

The main problem in the game of Wumpus is to learn the position of the monster. In order to plan for achieving this objective, an agent needs to be able to reason about its own knowledge and about how will it change as a result of performing various actions. Thus, the logic it utilises in its reasoning needs to strongly support epistemic concepts. At the same time, a notion of time-awareness is necessary, as we require our agent to consciously balance planning and acting.

To accommodate those requirements, we employ a variant of Active Logic [Elgot-Drapkin *et al.*, 1999] as the agent's underlying reasoning apparatus. This logic was designed for non-omniscient agents and has mechanisms for dealing with uncertain and contradictory knowledge. We believe it is a good reasoning technique for versatile agents, as it has been successfully applied to several different problems — including some in which planning plays a very prominent role [Purang *et al.*, 1999].

The domain of Wumpus game has one more interesting feature, namely that the interesting behaviour of the agent consists of two phases. First, it has to gather some information ("Where is the Wumpus?") and, after that, it needs to exploit this knowledge ("How to get rid of it from there?"). Since this knowledge only becomes available during plan execution, not while agent is creating the plan, it needs to make its choice of actions depend on the previous observations of the world. Therefore, it has to create, reason about and execute conditional plans. Currently we have chosen a simple, straightforward way of representing conditional actions, although quite a few more advanced formalisms can be found in the literature [Russell and Norvig, 2003].

To summarise, our agent will create several different plans and reason about usefulness of each one — including what knowledge can be acquired by executing it. Further, it will judge whether it is more beneficial to immediately begin executing one of those plans or rather to continue deliberation. In other words, the agent will be performing on-line planning, interleaving it with plan execution. Moreover, we expect it to live much longer than any single planning episode lasts, so it should generalise each solution it finds. In particular, the agent needs to extract domain-dependent control knowledge and use it when solving subsequent, similar problem instances. Finally, it will have to be able to handle non-stationary, adversary environment, to cooperate with others in multi-agent setting and to plan for goals more complex than simple reachability properties (such as temporally extended goals and restoration goals).

All of the features mentioned above have been extensively studied in the planning literature, including ideas how to integrate various combinations of them — and we will discuss some of this work through this paper. However, to the best of our knowledge, nobody has yet attempted to merge all, or even most, of those features together in one, consistent framework.

This work is divided in the following way: the next section presents the architecture of our agent, describing the important modules and their functions, as well as how they interact. Sections 3–5 provide more detailed overview of each module separately. In Section 6 we briefly present some of the most relevant work done by other researchers. We conclude with summary and several ideas for further work.

## 2 Architecture

The architecture of our agent, presented in Figure 1, consists of three main elements. First of them is the Deductor, which performs deductive reasoning about world, actions and their consequences. Its main aim is to generate plans applicable in current situation. Furthermore, it predicts — at least as far as agent's past experience and imperfect domain knowledge allows — effects each of those plans will have, including what new knowledge can be acquired.

The second component is the Actor, which chooses and executes plans created by Deductor. It is also responsible for observing the world and introducing effects of actions — and, potentially, other changes in the environment — into agent's knowledge base. It is important to note that Actor determines *when* to stop deliberation and start execution of the chosen plan.

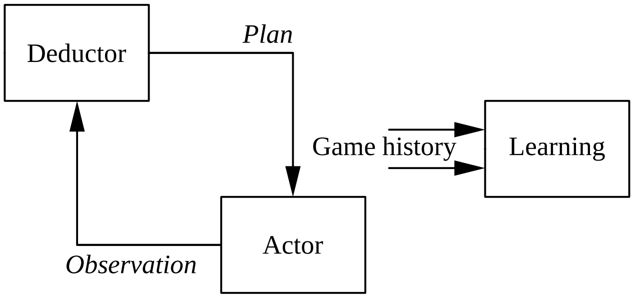


Figure 1: The architecture of the system.

These two modules form the core of the agent. By creating and executing a sequence of partial plans our agent moves progressively closer and closer to its goal, until it reaches a point where a winning plan can be directly created by Deductor, and its correctness can be proven.

However, success depends on whether the chosen partial plans are indeed moving an agent *closer* to the solution. Since agent's knowledge is incomplete and moreover it does not have enough resources to fully utilise the knowledge it possesses, there is — in principle — no guarantee that it will be so. In particular, if an Actor makes a mistake, the chosen plan may lead to loosing the game.

This is the reason for including the third module in our architecture. After the game is over, regardless of whether the agent has won or lost, learning system attempts to inductively generalise experience it has gathered — attempting to improve Deductor's and Actor's performance. We intend to use the learned information to fill gaps in the domain knowledge, to figure out generally interesting reasoning directions, to discover relevant subgoals and, finally, to more efficiently choose the best partial plan.

In principle, learning could take place at any time, but we do not currently see much benefit of learning in the middle of the game. Our variant of Wumpus game is simple enough that a single episode does not last very long, and there is some useful information that is only available to an agent after the game is finished — information which can be very valuable during learning.

## 3 Deductor

In order to present Deductor we begin with a description of the chosen knowledge representation formalism. Next we introduce those concepts from Active Logic which are necessary for understanding the rest of this text. We then present how the conditional actions are incorporated in our framework, and finally we illustrate how the three elements are combined for creating (partial) plans.

### 3.1 Knowledge representation

The language used by Deductor is the First Order Logic (FOL) augmented with Situation Calculus mechanisms for describing action and change. Within a given situation, knowledge is expressed using standard FOL. In particular, we do not put any limitations on the expressiveness of the language, as some mechanisms we later employ would invalidate benefits of restricting ourselves to languages such as Horn clauses or description logics. Predicate $K$ describes knowledge of the agent, e.g.,

$$K[\, smell(a) \leftrightarrow \exists_x (Wumpus(x) \land Neigh(a,x)) \,]$$

meaning: *agent knows that it smells on exactly those squares which neighbour Wumpus' position*. The predicate $K$ may be nested, although it is seldom useful. We use standard reification mechanism for putting formulae as parameters of the $K$ predicate.

The next step is to introduce action and change representation. We use the well-known Situation Calculus approach, introducing predicates $Holds(situation, formula)$ to denote that the *formula* holds in *situation* and $Informs(action, groundedwff)$ to denote that *action* provides information whether *groundedwff* holds. We also introduce function $Result(situation, action)$, which returns the set of situations resulting from applying *action* in *situation*.

Another important concept in our formalism is a plan, which is a sequence of actions. Plans may be subject to concatenation operation. In every place where this might matter (in particular at argument list of the $K$ predicate) we introduce two additional parameters. We denote by them, respectively, the set of situations and the set of plans to be executed (starting from those situations) in order to make the third argument true. So, actually, the formula shown above should look as follows:

$$K[\{s\}, \{p\},\ smell(a) \leftrightarrow \exists_x (Wumpus(x) \land Neigh(a,x)) \,]$$

meaning: *in a situation $s$ agent knows that if it executes plan $p$ then it smells on exactly those squares which neighbour Wumpus' position*.

This particular formula is true regardless of the chosen $s$ and $p$ (it is an universal law), a fact which we can denote, for example, by $\mathbb{S}$ (set of all situations) and either $\emptyset$ (empty plan) or $\mathbb{P}$ (the set of all plans). Still, there are many interesting formulae — like ones in the form "$Wumpus(x)$" — which are true *only* for specific $s$ and $p$.

Please observe that the main notion our agent reasons about is its own knowledge about the world. Similar idea was introduced in [Petrick and Bacchus, 2004], where authors investigate how various actions and observations of their effects modify agent's belief state. They describe how such modifications can be propagated backwards and forwards through the state history: as the agent gains new knowledge, it can infer that various statements *did* hold in past states of the world, even if it did not know it then. Authors also show how such propagation can be used to deal with temporally extended and restoration goals.

### 3.2 Active Logic

Active Logic (AL) is intended to describe the deduction as an ongoing process, instead of characterising just some infinite, fixed-point consequence relation. To this end, it annotates every formula with a time-stamp (usually an integer) of when it was first derived, and bookkeeps the reasoning process by incrementing the label with every application of an inference rule. E.g.,

$$\frac{i:\quad a, a \to b}{i+1:\quad b}$$

Additional features, available in AL and important for this work, include the $Now$ predicate, true only during current time point (i.e., "$i : Now(j)$" is true for all $i = j$, but false for all $i \neq j$) and the *observation function*, which delivers axioms that are valid since a specific time-point. It is used to model agent acquiring new knowledge from the environment. This way the reasoning process may refer, via $Now$, to the current (absolute or relative) time and conclude whether it has passed a deadline or not. It can also describe change that is not a result of performing any action — thus lifting two important limitations present in the classical situation calculus.

### 3.3 Conditional plans

The conditional plans we consider consist of a concatenation of classical and conditional actions, where each conditional action may be described as $(predicate\ ?\ action_1 : action_2)$, meaning that $action_1$ will be executed if $predicate$ holds, and $action_2$ will be executed otherwise. We consider the possibility of introducing a more complex structure of conditions (like *while* loops), but within this application simple conditionals will suffice.

This type of conditional actions introduces a high branching factor in case of longer plans, but this effect is unavoidable at some level of consideration and will not be further discussed here. It has received some attention in the works by other authors (see [Russell and Norvig, 2003] for extended bibliography).

For a well-developed discussion of conditional partial plans and interleaving planning and execution see for example [Bertoli *et al.*, 2004], where authors introduce notion of *progressive plan* — intuitively, one that provably moves the agent closer to the goal. They also present an algorithm for finding such plans in a nondeterministic but fully known domain and prove that it is guaranteed to find a solution if one exists.

A somewhat similar, very interesting idea was pursued in [Nyblom, 2005], where author uses classical planner to plan for "optimistic" case, where an agent can choose the most favourable outcome of each non-deterministic action. From such an optimistic plan it is then possible, using knowledge of probabilities of each action outcome, to generate more realistic plans by updating relative costs of optimistic actions.

### 3.4 Reasoning about plans

Finally, the representation language needs to be augmented with reasoning capabilities. It is done using a set of rather natural, although not quite trivial, inference rules. Their presentation, however, is outside the scope of this paper. Using those rules, the Deductor may conclude, from the example formula shown earlier, that

$$\forall_x \; K[\mathbb{S}, \mathbb{P}, \; \neg smell(a) \wedge Neigh(a, x) \;] \leftrightarrow$$
$$K[Result(s, p), \emptyset, \; \neg Wumpus(x) \;]$$

i.e., that *if it doesn't smell in position $a$ then the agent will know that there is no Wumpus on any of its neighbour positions*. This may be further used for creating a useful plan of actions given that the agent currently is, or has been before, in position $a$.

One of the reasons we have chosen symbolic representation of plans, as opposed to a policy (an assignment of value to each state–action pair) is that we intend to deal with other types of goals than just reachability ones. For a discussion of possibilities and rationalisation of why such goals are interesting, see for example [Bertoli *et al.*, 2003], where authors present a solution for planning with goals described in Computational Tree Logic. This formalism allows to express goals of the kind "value *a* will never be changed", "*a* will be restored to its original value" or "value of *a* after time *t* will always be *b*" etc.

Furthermore, one of our ideas is to extend the solution presented in this paper to the case of multi-agent cooperative planning, where benefits of symbolic plan representation are even more clear.

To summarise, the agent uses the formalism presented in this section in order to deductively develop plans. Given the complexity of the domain and vastly branching proof procedure (currently it can only perform forward chaining) the created plans are usually partial, i.e. they lead to some intermediate states of the game, where the final outcome is not yet decided.

## 4 Actor

The Actor module supervises the deduction process and breaks it at selected moments, e.g., when it notices a particularly interesting plan or when it decides that sufficiently long time has been spent on planning. It then *evaluates* existing partial plans and executes the best one of them. The evaluation process is crucial here, and we expect the subsequent learning process to greatly contribute to its improvement. In the beginning, the choice may be done at random, or some simple heuristic may be used. After execution of partial plan, a new situation is reached and the Actor lets the Deductor create another set of possible plans.

This is repeated as many times as needed, until the game episode is either won or lost. Losing the game clearly identifies bad choices on the part of the Actor and leads to an update of the evaluation function.

Winning the game also yields feedback that may be used for improving this function, but it also provides a possibility to (re)construct a complete plan, i.e. one which originates from the initial situation and ends in a winning state. If such a plan can be found, it may be subsequently used to immediately solve any problem instance for which it is applicable. Moreover, even if such plan is not applicable, an Actor can use it when evaluating other plans found by the Deductor. Those which have similar structure to the successful one are more likely to lead to the goal.

In other words, the intention is for Actor to acquire generalised knowledge of the domain, which can be used to guide an agent in more promising directions.

In a sense this is similar to ideas discussed in [Fern *et al.*, 2004], where authors use Markov Decision Process to represent planning domains and approximate policy iteration as means of learning agent's behaviour. They use long random walks to create progressively harder goals, thus bootstrapping the agent in its learning of domain-dependent control knowledge.

## 5 Learning

As we mentioned earlier, our agent will be presented with large number of tasks to solve. Therefore, upon finishing each game episode, the events (actions, observations and the result) are fed into a learning module. This module attempts to generalise this information and provide guidelines for Actor and Deductor to improve their performance. In this paper we will mainly investigate the learning module from Actor's perspective, as using ILP framework to evaluate quality of partial plans is, to the best of our knowledge, a novel idea. In further work we also intend to improve domain knowledge and to identify interesting reasoning directions, but those later ideas are — while definitely interesting and non-trivial — mainly a matter of integrating the already available techniques.

### 5.1 Goal of learning

The first task we would like our learning module to address is how an Actor is to choose which one of the plans being considered by Deductor should it execute. Clearly, the longer it allows planning phase to proceed, the better plans will it get to choose from, and the more information about consequences of each plan will be known. On the other hand, more of the deduction effort will be wasted by considering potential situations which will not take place in this particular game.

At some point, however, an Actor must choose one plan, from those created by Deductor, for immediate execution. Some of those plans are better than others — but it cannot be determined exactly and with full confidence until those plans extend to the terminal state of the game. And for problems we intend to tackle, that is intractable — agent's computational resources do not suffice to *completely* solve problems we are interested in. Therefore, the Actor needs some heuristic method of evaluating quality of partial plans and of comparing them.

There is quite a bit of knowledge that domain experts could provide — but our aim is to have a solution which does not *require* such experts. At the same time, if people familiar with particular domain are available, the agent should take advantage of whatever information they can provide. Therefore, Inductive Logic Programming appears to fit our needs quite well: it uses background knowledge when it is available, but can also solve problems when it is not.

It is important to keep in mind that our agent has a dual aim, very akin to the exploration and exploitation dilemma, well-studied in reinforcement learning and related research areas. On one hand, it wants to win the current game, but at the same time it needs to learn as much general knowledge as possible — in order to improve its performance at subsequent tasks.

## 5.2 Choosing plans

There are clearly many features which can distinguish between good and bad plans. And with sufficiently rich history of game episodes, it is possible to learn this distinction. In the simplest case the agent can start with Actor randomly choosing plans for execution. After a couple of games — some of which will be won but, likely, many will be lost — it will have enough experience to learn some useful rules.

The main problem is that most work on ILP, as well as on Machine Learning in general, has been dealing with the problem of *classification*, while what we need is rather *evaluation*. There is no predefined set of classes into which plans should be assigned. What our agent needs is a way to choose the *best* one of them.

Still, in order to be able to take advantage of the vast amount of research done in the Inductive Logic Programming framework, in the first step we recast our problems as a classification one. In particular, we attempt to distinguish plans that leading to losing the game from all the others. In our initial architecture this part is relatively easy — we assume that the Deductor has perfect knowledge of consequences of execution of each plan, so it can deduce (for some plans $p$) a fact "$K[\{s\}, \{p\}, \neg die]$".

A separate question is whether an Actor can *learn* to choose only plans for which "$K[\neg die]$" has been deduced. After all, not every plan for which such fact cannot be proven actually *does* lead to losing every game.

Moreover, it is worth noting that if the Wumpus is allowed to move, there exist plans which do not lead to agent's death, but which do lead to states where winning it is no longer possible — for example, if an agent gets stuck in a corner with Wumpus blocking its way out. It may be difficult for an agent to notice and learn that the mistake has been made in the previous step, not in the one when the agent was killed.

## 5.3 Application of ILP

From the above analysis it becomes clear than the notion of *positive* and *negative* examples, as used in ILP algorithms, is not quite appropriate for what we would like to express in our framework. What they correspond to, informally, are conditions that are both *necessary* and *sufficient* — while we are mainly interested those that are sufficient.

An interesting line of research, which possibly could be useful in our case, was presented in [Gretton and Thiebaux, 2004], where authors attempt to deductively generate domain-specific hypothesis language which is as simple as possible, and yet expressive enough to represent all the necessary concepts in a particular domain. This language is then used by inductive learning algorithm to create generalised policies from solutions of small problem instances.

Let us assume that we restrict ourselves to dividing plans into two classes: those that can lead to agent's death and those that cannot. Each partial plan executed at some previous game can be seen as a single example. First issue we need to deal with is which example belongs to which class. It is easy to note that some plans — namely those that in agent's experience *do* lead to losing the game — are definitely examples of bad plans. However, not every plan which does not cause the agent to die is, indeed, a *good* plan. What is more, not every plan that leads to *winning* a game is a good one. An agent executing a dangerous plan might have just gotten lucky, if in a particular episode Wumpus was in favourable position.

Therefore, if we want ILP algorithm to learn the concept of bad plans, we do have a set of positive examples, and a set of examples for which we do not — at least not immediately — know their affiliation. We have decided to use PROGOL as a learning algorithm. The standard version is presented in [Muggleton, 1995] and can be described here, in a somewhat simplified manner, by the following steps:

1. Select an example to be generalised. If no more examples exist, stop.
2. Construct the most specific clause, within provided language restrictions, which entails selected example. This is called the ”bottom clause”.
3. Find, by searching for some subset of the literals in the bottom clause, more general clauses. Choose one with the best "score".
4. Add the clause found in the previous step to the current theory, and remove all clauses made redundant. It is worth noting that the best clause may make clauses other than the examples redundant. Move back to Step 1.

In our case we can define as positive examples those plans which lead — or can be *proven* to *possibly* lead — to agent's death. On the other hand, those plans which can be proven to *never* lead to the agent's death are treated as negative examples. We are working on ways to utilise other plans in some way, those for which neither of the above assertions can be proven (within reasonable time) — right now we simply exclude them from learning.

With the definitions as above, we can use standard ILP algorithm, be it PROGOL or almost any other, to have Actor learn to choose only *non-losing* plans for execution.

However, this is only a beginning. After all, it is not quite enough not to die, as an agent which moves in circles, without exploring the world, clearly does not get eaten by the Wumpus — but it never wins either. On the other end of the spectrum, the feature "plan which kills the Wumpus" is clearly non-operational.

Hopefully, we will be able to report more details on practical applicability of the ideas described above when our implementation is finished and we have run some experiments.

## 5.4 Further ideas

One very promising idea seems to be exploring the epistemic quality of plans. An agent should pursue those plans which provide it with the most important knowledge. Clearly, in the Wumpus domain *important* is directly linked with monster's true position — or at least that is what human players consider it to be. Therefore, as a next step, we can redefine *bad* plans as those that lead to the agent's death or do not provide any interesting knowledge. Again, we can use one of many ILP algorithms to learn such concepts.

Another very general and important way of expressing distinction between good and bad partial plans, and one we feel can lead to very good results, is related to discovering relevant subgoals and landmarks in the plans, akin to the work done in [Hoffmann *et al.*, 2004].

The problem is that those ideas require more domain knowledge than we are comfortable with. For example, what we would like to have is an agent figuring out that "position of Wumpus" is important just from the definition of the rules and goals of the game. In principle, it appears to be possible — it is not difficult to deduce that knowing Wumpus' position suffices for winning the game (the plan to win once Wumpus' position is known is simple and can be found easily). However, it is not clear how to combine such reasoning with learning as expressed above. It is our understanding that some modifications to the learning algorithm will be required.

To summarise, it is easy (for a human) to see some general rules distinguishing good plans from bad ones. For example, a plan for which an agent doesn't know that it will not lose the game is a bad plan. Such knowledge can be easily provided by domain expert and most ILP algorithms are ready to use it. Interesting question, however, is whether and how can this knowledge be discovered by an agent itself.

One way would be to try something along the lines of research presented in [Walker *et al.*, 2004], where authors randomly sample a large number of relational features and evaluate them on small problems. The idea is that features found to work satisfactory on such sample problems should also describe larger problems sufficiently well.

## 6 Related Work

Combination of planning and learning is an area of active research, in addition to the extensive amount of work being done separately in those respective areas. However, most of the related work we are aware of is devoted to either using state-of-the-art learning in a rather limited planning framework, or to using limited learning in a more complex planning setup. Comparisons of the two areas are also relatively common, while the true, nontrivial combination will apparently require much more investigation. Since we believe it to be very promising, this paper is aiming at attracting attention to this line of research.

The first to mention is [Dietterich and Flann, 1995], which presented results establishing conceptual similarities between explanation-based learning and reinforcement learning. In particular, they discussed how EBL can be used to learn action strategies and provided important theoretical results concerning its applicability to this aim.

There has been significant amount of work done in learning about what actions to take in a particular situation. One notable example is [Khardon, 1999], where author showed important theoretical results about PAC-learnability of action strategies in various models.

In [Moyle, 2002] author discussed a more practical approach to learning Event Calculus programs using Theory Completion. He used extraction-case abduction and the ALECTO system in order to simultaneously learn two mutually related predicates ($Initiates$ and $Terminates$) from positive-only observations.

Recently, [Könik and Laird, 2004] developed a system which is able to learn low-level actions and plans from goal hierarchies and action examples provided by experts, within the SOAR architecture.

The work mentioned above focuses primarily on learning how to act, without focusing on reaching conclusions in a deductive way. In a sense, the results are somewhat more similar to the reactive-like behaviour than to classical planning system, with important similarities to the reinforcement learning and related techniques. In case of large search spaces this approach may not be as effective as a suitable combination of learning and deduction. Therefore, some effort have been devoted to searching for a suitable combination.

One attempt to escape the trap of large search space has been presented in [Džeroski *et al.*, 2001], where relational abstractions are used to substantially reduce cardinality of search space. Still, this new space is subjected to reinforcement learning, not to a symbolic planning system.

A conceptually similar idea, but where relational representation is actually being learned via behaviour cloning techniques, is presented in [Morales, 2004].

Outside the domain of planning, there is a lot of interesting research being done in the learning paradigm.

Recently, [Colton and Muggleton, 2003] showed several ideas about how to learn interesting facts about the world, as opposed to learning a description of a predefined concept. A somewhat similar result, more specifically related to planning, has been presented in [Fern *et al.*, 2004], where the system learns domain-dependent control knowledge beneficial in planning tasks.

From another point of view, [Khardon and Roth, 1995] presented a framework of learning done "specifically for the purpose of reasoning with the learned knowledge" — an interesting early attempt to move away from the *learning to classify* paradigm.

Yet another track of research focuses on (deductive) planning, taking into account incompleteness of agent's knowledge and uncertainty about the world. Conditional plans, generalised policies, conformant plans and universal plans are the terms used by various researchers [Cimatti *et al.*, 2004; Bertoli *et al.*, 2004] to denote in principle the same idea: generating a plan which is "prepared" for all possible reactions of the environment. This approach has much in common with control theory, as observed in [Bonet and Geffner, 2001] or earlier in [Dean and Wellman, 1991]. We are not aware of any such research that would attempt to integrate learning.

As can be seen, many of the ideas we investigate in this paper have been analysed previously, but an attempt to merge them into a single, consistent framework has not yet been made.

## 7 Conclusions and Further Work

The work presented here is more a discussion of an interesting track of research than a report on some concrete results. However, we think that this idea is important and promising enough to be subjected to wider discussion, and therefore we have decided to present it in this forum.

We have introduced an agent architecture facilitating resource-aware deductive planning interwoven with plan execution and supported by inductive, life-long learning. The particular deduction mechanism used is based on Active Logic, in order to incorporate time-awareness into the deduction itself. The plans created in deductive way are conditional, taking into account possible results of future actions, in particular information-gathering ones.

The learning mechanism employed is based on PROGOL, although in principle any standard ILP algorithm would be suitable as well. Learning is expected to provide an evaluation of the current deductive knowledge in order to improve the agent's performance in the long run.

We are at the moment working on implementation of the system and expect to be able to report results of first experiments at the time of the workshop.

In the future we intend to continue this work in the following directions:

- Discovering subgoals and subplans. It seems that one of the most useful capacities of humans problem solving is the ability to divide a complex problem into subproblems and then to solve each of them separately before combining their solutions into a global one. We would like to force our agent to discover this possibility. In our example domain a useful subgoal/subproblem could be "First, find a place where it smells."
- Discovering general rules which Deductor will be able to use later on. An example of such a rule might be "Don't shoot if you don't know Wumpus' position". It seems that availability of such rules can save a substantial amount of work for Deductor, if it can establish early on that some plans would not be usable.
- Generalisation of plans. A clear advantage would be to reuse a valid plan in a different context. As long as the context does not differ substantially, this operation should lead to fast solution of a problem similar to one solved in the past.
- Capability of handling imperfect knowledge. The current setup assumes complete domain knowledge, while in many situations this assumption might be violated (e.g., the agent might not know that the Wumpus actually can move). The system should allow the agent to learn domain knowledge, if possible, to complete its understanding of the environment.
- Last, but not least, allow interaction with a user. Domain experts might be an invaluable source of knowledge that the agent should be able to exploit, if possible. For example, to better adjust tradeoff between spending time on deduction and induction, the agent could be guided by an external observer (the user) providing a feedback about its performance.

The list above does not cover all the possible further investigations and extensions of the proposed system; it is just a biased presentation of the authors' own interests and judgements.

## Acknowledgements

The authors are grateful to the anonymous reviewers for their very thorough comments and suggestions, which lead to a substantial improvement of this paper.

## References

Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, and Paolo Traverso. A framework for planning with extended goals under partial observability. In *International Conference on Automated Planning and Scheduling*, pages 215–225, 2003.

Piergiorgio Bertoli, Alessandro Cimatti, and Paolo Traverso. Interleaving execution and planning for nondeterministic, partially observable domains. In *European Conference on Artificial Intelligence*, pages 657–661, 2004.

Blai Bonet and Hector Geffner. Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence*, 14(3):237–252, 2001.

Alessandro Cimatti, Marco Roveri, and Piergiorgio Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1-2):127–206, 2004.

Simon Colton and Stephen Muggleton. ILP for mathematical discovery. In *13th International Conference on Inductive Logic Programming*, 2003.

Thomas Dean and Michael P. Wellman. *Planning and Control*. Morgan Kaufmann, 1991.

Thomas G. Dietterich and Nicholas S. Flann. Explanation-based learning and reinforcement learning: A unified view. In *International Conference on Machine Learning*, pages 176–184, 1995.

Saso Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine Learning*, 43(1/2):7–52, 2001.

Jennifer Elgot-Drapkin, Sarit Kraus, Michael Miller, Madhura Nirkhe, and Donald Perlis. Active logics: A unified formal approach to episodic reasoning. Technical Report CS-TR-4072, University of Maryland, 1999.

Alan Fern, SungWook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In *International Conference on Automated Planning and Scheduling*, 2004.

Charles Gretton and Sylvie Thiebaux. Exploiting first-order regression in inductive policy selection. In *Conference on Uncertainty in Artificial Intelligence*, 2004.

Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

Khardon and Roth. Learning to reason with a restricted view. In *Proceedings of the Workshop on Computational Learning Theory, Morgan Kaufmann Publishers*, 1995.

Roni Khardon. Learning to take actions. *Machine Learning*, 35:57–90, 1999.

Tolga Könik and John Laird. Learning goal hierarchies from structured observations and expert annotations. In *14th International Conference on Inductive Logic Programming*, 2004.

Eduardo P. Morales. Relational state abstraction for reinforcement learning. In *Proceedings of the ICML'04 Workshop on Relational Reinforcement Learning*, 2004.

Steve Moyle. Using theory completion to learn a robot navigation control program. In *12th International Conference on Inductive Logic Programming*, 2002.

Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

Per Nyblom. Handling uncertainty by interleaving cost-aware classical planning with execution. In *Swedish AI Society Workshop*, 2005.

Ronald P. A. Petrick and Fahiem Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 2–11, 2004.

Khemdut Purang, Darsana Purushothaman, David Traum, Carl Andersen, and Donald Perlis. Practical reasoning and plan execution with active logic. In *Proceedings of the IJCAI-99 Workshop on Practical Reasoning and Rationality*, pages 30–38, 1999.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in AI, 2nd edition, 2003.

Trevor Walker, Jude Shavlik, and Richard Maclin. Relational reinforcement learning via sampling the space of first-order conjunctive features. In *In working notes of ICML-04 Workshop on Relational Reinforcement Learning*, 2004.

# Robust and Opportunistic Planning for Planetary Exploration

**Daniel M. Gaines, Tara Estlin, Caroline Chouinard, Forest Fisher**
**Rebecca Castaño, Robert C. Anderson, and Michele Judd**

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr., Pasadena CA 91109
{*firstname.lastname*}@jpl.nasa.gov

## Abstract

Planning for rover operations involves a significant amount of uncertainty. With limited a priori knowledge of the area a rover will explore, it is difficult to predict the effects of actions including their duration and the amount of resources they will consume. In addition, the system may not even know ahead of time all of the goals it will be asked to achieve as new opportunities may be identified during the mission. We are developing the OASIS system to enable rovers to generate and execute high quality mission operations plans and to identify and exploit new science opportunities that may arise during the mission. OASIS combines planning and machine learning techniques to achieve these results. In this paper we discuss how OASIS handles these types of uncertainties and present results from testing the system in simulation and on rover hardware.

## 1 Introduction

Planetary exploration by its nature involves a significant amount of uncertainty. The objective of such missions is to gather information about previously unknown areas. As such, little a priori information may be available about the nature of the terrain a rover must explore or the obstacles that it will encounter. This makes it challenging to develop an operation sequence as it is difficult to estimate the time and resources required by rover activities.

In addition, we are developing technologies that enable rovers to identify potentially interesting science opportunities on their own. This will provide important capabilities for rovers such as enabling rovers to identify opportunities that might have otherwise gone unnoticed or to take advantage of short-lived science opportunities such as a passing dust devil. However, this capability also adds another element of uncertainty to mission operations as the rover will not know ahead of time all the science goals it will be asked to work on. New goals with different priorities may be posted to the system at any time.

We have developed the OASIS (Onboard Autonomous Science Investigation System) integrated science analysis and planning system that enables planetary rovers to generate and execute high quality mission operations plans in the presence of these types of uncertainty. OASIS includes a continuous planning system to generate operations plans given prioritized science goals and mission constraints and to monitor and repair plans during execution. The system also includes a data analysis unit that uses machine learning algorithms to perform onboard processing of collected science data. When a science opportunity is detected, one or more requests are sent to the planning and execution system which attempts to accomplish these additional objectives while still achieving current mission goals.

## 2 OASIS

The OASIS system provides onboard science analysis coupled with planning and execution. The system enables a rover to carry out prioritized science goals commanded from Earth as well as opportunistic science goals identified by onboard data analysis. Figure 1 shows the main components of the OASIS system and how they interact to analyze data and re-task the rover to respond to opportunistic science events. OASIS consists of the following components:

**Planning and Scheduling:** generates operations plans for mission goals and dynamically modifies plan in response to new science requests.

**Execution:** carries out the rover functional capabilities to perform the plan and collect data. Oasis TDL [Simmons and Apfelbaum, 1998] for its Executive and the CLARAty [Nesnas *et al.*, 2003] functional layer for low-level robotic capabilities.

**Feature Extraction:** detects rocks in images and extracts rock properties (e.g. shape and texture).

**Data Analysis:** uses extracted features to assess the scientific value of the planetary scene and to generate new science objectives that will further contribute to this assessment.

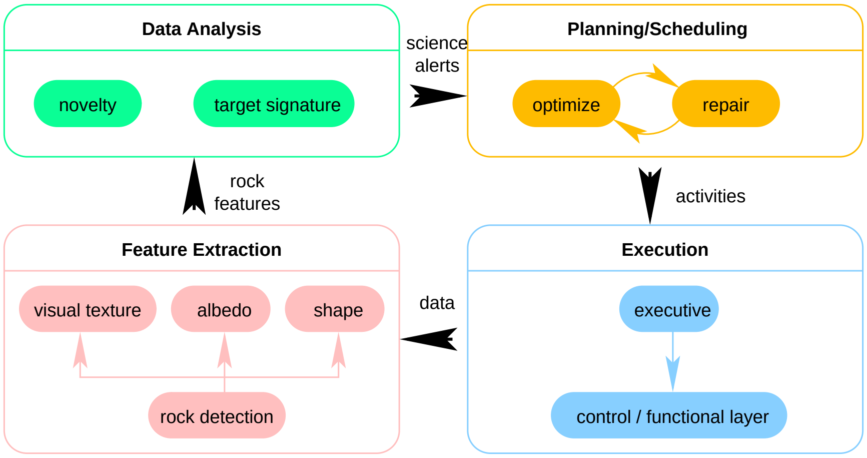


Figure 1: OASIS architecture.

The feature extraction and data analysis components of OASIS have been described previously in [Castano *et al.*, 2004]. Here we will give a brief overview of these components and concentrate on the planning and scheduling unit and how it supports opportunistic science.

## 2.1 Feature Extraction

Our initial emphasis in OASIS has focused on image analysis and the characterization of surface rocks. Rocks are among the primary features populating the Martian landscape and the understanding of rocks on the surface is a first step leading to more complex regional geological assessments.

Images are segmented using a rock detection algorithm based on edge detection and tracing. Next, a set of properties is extracted from each rock. Our feature extraction priorities are based upon our knowledge of how a geologist in the field would extract information. Important features to look for and categorize include albedo (an indicator of rock surface reflectance properties), visual texture (which provides valuable clues to mineral composition and geological history), shape, size, color and arrangement of rocks. Currently our system identifies the first three of this set; future work will expand this to cover additional features.

## 2.2 Data Analysis

After features have been extracted from each rock, OASIS runs a set of data analysis algorithms to look for interesting rocks. Two of these algorithms can result in the generation of science alerts: key target signature and novelty detection.

**Key Target Signature:** enables scientists to efficiently and easily stipulate the value and importance of certain features. Scientists often have an idea of what they expect to find during a rover mission and/or are looking for specific clues that reflect signs of life or water (past or present). Using this technique, target feature vectors can be pre-specified and an importance value assigned to each of the features. Rocks are then prioritized as a function of the weighted Euclidean distance of their extracted features from the target feature vector.

**Novelty Detection:** detects and prioritizes unusual rocks that are dissimilar to previous rocks encountered. We have looked at three different learning techniques for novelty detection: distance-based using k-means clustering, probability-based using Gaussian mixture models and discrimination-based using kernel one-class classifier. The general idea is that as the rover collects data about the rocks in an area, the machine learning techniques will enable it to build a model of the characteristic rocks. If a new sample falls well outside of this model, then it is considered novel and potentially worthy of further investigation.

## 2.3 Science Alert Protocol

Using the above algorithms, the data analysis software can flag rocks that should be further analyzed and produce a new set of measurement goals. We call this capability the *science alert*, since it alerts other onboard software that new and high priority science opportunities have been detected. OASIS currently supports two types of alerts. A *stop and call home* alert indicates that the rover should remain at its current location until it has received further instructions form Earth. Such an alert would typically be reserved for situations in which data analysis has made an extremely interesting observation and the rover should stay where it is to avoid the risk of losing the target. The second class of alerts is *data sample requests* in which the rover is requested to perform an additional science measurement and then continue on with previously scheduled activities. Achieving this alert may require the rover to change its heading or possibly its position.

## 2.4 Planning and Scheduling

The objectives of OASIS's planning and scheduling component is to maximize the value of the science that is performed by the rover and to ensure that the operations plan satisfies rover and mission constraints. To provide robust execution, the system must respond to problems that might arise during plan execution, such as an activity consuming more resource than expected. To maximize the value of the plan, the system must exploit opportunities that arise. These may include additional available time due to an activity taking less time than expected or a new, highly interesting goal that has been identified by Science Analysis.

Planning and scheduling capabilities in OASIS are provided by CASPER [Estlin *et al.*, 2002; Chien *et al.*, 2000], which employs a continuous planning technique where the planner continually evaluates the current plan and modifies it when necessary based on new state and resource information. Rather than consider planning a batch process, where planning is performed once for a certain time period and set of goals, the planner has a current goal set, a current rover state, and state projections into the future for that plan. At any time an incremental update to the goals or current state may update the current plan. This update may be an unexpected event (such as a new science opportunity) or a current reading for a particular resource level (such as power). The planner is then responsible for maintaining a plan consistent with the most current information.

A plan consists of a set of grounded (i.e., time-tagged) activities that represent different rover actions and behaviors. Rover state in CASPER is modeled by a set of plan timelines, which contain information on states, such as rover position, and resources, such as power. Timelines are calculated by reasoning about activity effects and represent the past, current and expected state of the rover over time. As time progresses, the actual state of the rover drifts from the state expected by the timelines, reflecting changes in the world. If an update results in a problem, such as an activity consuming more memory than expected and thereby over-subscribing RAM, CASPER re-plans, using iterative repair [Zweben *et al.*, 1994], to address conflict.

CASPER includes an optimization framework for reasoning about soft constraints. User-defined preferences are used to compute plan quality based on how well the plan satisfies these constraints. Optimization proceeds similar to iterative repair. For each preference, an optimization heuristic generates modifications that could potentially improve the plan score.

While CASPER provides a framework for integrated planning and scheduling, there is still significant work required to apply the system effectively to a complex domain. For our rover work, this included developing a domain model for rover operations, developing a control algorithm geared toward appropriately responding to problems and opportunities and integrating optimization and repair. In order to realize an operation rover system, we also integrated CASPER with a large number of systems such as path planning, navigation, position estimation, an executive, and science analysis.

We have developed a domain specific control algorithm within CASPER to support the objectives of maximizing plan quality and ensuring robust execution. Figure 2 provides a high level description of this algorithm. Table 1 shows the preferences that are used to compute a plan score ordered from most important to least important. The score for the plan is a weighted sum of each preference. The weights are set to reflect relative importance of these preferences. For example, achieving stop and call home alerts is of highest importance to make sure that if a stop and call home alert is issued the rover will always prefer plans that achieve it. As another example, plans without conflicts are more important than plans that achieve more goals but have conflicts.

| **Preference** |
|---|
| Prefer plans that achieve stop and call home |
| Prefer plans with fewer conflicts |
| Prefer plans with more goals achieved |
| Prefer plans with less time spent traversing |

Table 1: Preferences used to compute plan score ordered from most important to least important

### Initial Plan Generation

We use a Depth First Branch and Bound algorithm to generate the initial operations sequence. The input to the system is a set of prioritized science requests and constraints on the time and energy available for carrying out the mission. We use a "tiered" objective function that first ensures that plans that exceed resource or time constraints are rejected. Next, it computes the value of the science goals in the plan using a "strict priority" scheme in which a plan must achieve higher priority goals before including lower priority goals. Finally, plans are scored based on distance traveled. The result is an initial plan that maximizes the value of science goals that can be achieved with time and resource constraints.

### Plan Execution

CASPER monitors updates from the Executive as the plan is executed, checking for problems that must be resolved or opportunities that can be exploited. A problem can occur with an activity at any point during its lifetime. For examples, an update may indicate that there will be a problem with an activity scheduled to start at some time in the future. In this case, CASPER will use iterative repair as part of the optimization loop to try to resolve the conflict.

Problems may also occur for activities that have already been passed to the Executive but have not yet begun execution. In this case, CASPER will send a rescind message for the problematic activity to the Executive. If the Executive receives the message before the activity has begun execution, it will delete it and send CASPER a

```
Input
    Prioritized science goals from Earth
    Time constraint
    Resource constraints

Initial Plan Generation
    Run Depth First Branch and Bound given initial science goals and constraints

Plan Execution
    While running
        Get current time
        Process any updates from Executive
        For each activity scheduled to start within <n> seconds
            If activity does not contribute to an existing conflict, send to Executive
        If there are conflicts in the schedule
            If an activity already sent to executive is contributing, rescind activity
        Optimize:
            for i = 1 to num_optimize_iterations
                Compute plan score based on preferences in Table 1
                If score of current plan is best so far, save plan
                If there is an unsatisfied opportunistic science goal, satisfy it
                Else, if there are conflicts, perform an iteration of repair
                Else, if there are unsatisfied science goals, satisfy one from set of highest priority science goals
            Reload plan with highest score
        If an opportunistic science goal has not been satisfied for opsci_time_limit, delete the goal
        If no activities are currently executing, check if an activity in the future can be moved up in time
```

Figure 2: CASPER control algorithm for rover domain.

confirmation. If the activity has already begun execution, the Executive will abort the activity and send an update to CASPER once the activity has been aborted.

The Executive itself monitors problems with activities that are currently executing. If a problem is detected, it is the responsibility of the executive to abort the activity and send an update to CASPER to let CASPER know that the activity was aborted.

While the first priority of the planning and scheduling system is to ensure robust execution, it is also continually checking for opportunities to increase the value the mission. An update from the Executive may indicate that an activity took less time or energy than predicted. In this case, it may be possible to achieve a goal that was not included in the initial plan. During the optimization loop, if all conflicts have been resolved, CASPER will select a high priority goal from the set of unsatisfied goals and add it to the schedule. This will most likely introduce new conflicts and the following optimization iterations will be spent trying to resolve them. If the conflicts can be resolved, the plan score will be increased and this plan will be saved as the best seen so far.

If an opportunistic science opportunity has been identified by Data Analysis, CASPER will try to add it to the plan. Again, this is likely to introduce conflicts and iterative repair will be used to try to fix them. It may be that the rover's schedule is too constrained to achieve the opportunistic goal. We set a timer for each opportunistic goal and if the timer expires before the goal is achieved, the goal is permanently deleted.

As a final check to try to maximize the use of rover resources, after the optimize loop, if there are no currently executing activities, CASPER will look ahead in the schedule to see if a future activity can be moved up in time without causing a conflict. If so, this will result in packing the schedule, limiting rover idle time.

## 3 System Testing

To evaluate our system we performed a series of tests both in simulation and using rover hardware in the JPL Mars Yard (Figure 3). These tests covered a wide range of scenarios that included the handling of multiple, prioritized science targets, limited time and resources, opportunistic science events, resource usage uncertainty causing under or over-subscriptions of power and memory, large variations in traverse time, and unexpected obstacles blocking the rover's path.

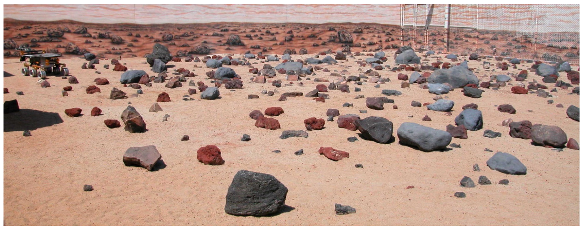


Figure 3: Testing with the FIDO rover in the Mars Yard.

Our testing scenarios typically consisted of a number of science targets specified at certain locations. A map was used that would represent a sample mission-site location where data would be gathered using multiple instruments at a number of locations. Figure 4 shows a sample scenario that was run as part of these tests. This particular map is of the JPL Mars Yard. The pre-specified science targets represented targets that would be communicated by scientists on Earth. These targets were typically prioritized and for many scenarios constraints on time, power or memory would limit the number of science targets that could be handled. The map also shows the path that was planned for the rover and the path the rover actually followed. These are not necessarily the same as the planned path does not account for all the obstacles the rover may have to avoid. A large focus of our tests was to improve system robustness and flexibility in a realistic environment. Towards that goal we used a variety of target locations and consistently selected new science targets and/or new science target combinations that had not been previously tested.

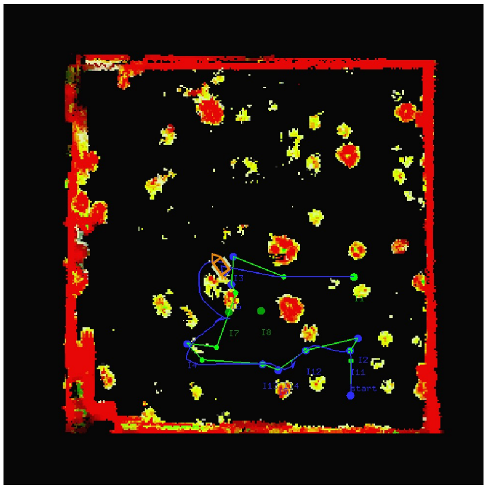


Figure 4: Example scenario.

Another primary scenario element was dynamically identifying and handling opportunistic science events. For these tests, we concentrated on a particular type of event, which was finding rocks with a high albedo measurement (i.e., light or white-colored rocks). This setting was an example of using the data analysis algorithm for target signature, where a particular terrain signature is identified as having a high interest level. If rocks were identified in hazard camera imagery that had a certain interest score, then a science alert was created and sent to the planner. If a science alert was detected the planner attempted to modify the plan so an additional image of the rock of interest was acquired.

Other important scenario elements included adding or deleting ground-specified science targets based in resource under or over-subscriptions. For instance, in some tests, the rover covered distances faster than expected and the planner was able to add in additional science targets that could not be fit into the original plan. Conversely, in other tests, the rover used more power than expected during traverses (or science measurements), which eventually caused a power over-subscription. The planner resolved this situation by deleting some lower priority science targets. Unexpected energy drops during a traverse could also be handled by the executive, which detects the shortfall and stops the current traverse if there is not enough energy to complete it. In all cases, the planning and execution system attempts to preserve as many high priority science targets as possible with current resource and time settings.

## 3.1 Discussion of Test Results

We are in the process of developing a formal evaluation process by which we will be able to obtain quantitative measurements of how well our system provides robust and opportunistic planning and execution. At this point we have more anecdotal results from our extensive testing in simulation and with rover hardware in the JPL Mars Yard.

Tests in the Mars Yard typically consisted of 20-50 meter runs over a 100 square meter area with many obstacles that cause deviations in the rover's path. Most rocks in the Mars Yard are dark in color, thus we brought in a number of whiter rocks to trigger science alerts during rover traverses. Science measurements using rover hardware were always images, since other instruments were not readily available (e.g., spectrometer). However different types of measurements were included when testing in simulation.

As a final test of our system, we performed a several hour long demonstration in October 2004. This demonstration covered the elements previously presented in this section. Further, the combination of science targets used had not been previously tested with. This set also included a science target that was selected that day by a present Mars Exploration Rover (MER) scientist. Rocks intended to cause science alerts were also placed in new locations not previously used. Overall, the demonstration was very successful. Two scenario runs were performed. Both had multiple targets with time or resource constraints preventing all targets from being included in the initial plan. In the first run a number of science alerts were correctly identified and handled. This run also had an additional science target added dynamically in the run due to the rover traveling faster than estimated. In the second run, lower priority targets were deleted due to more power being used in early traverses than expected. The software presented in this paper (planning, scheduling, execution, feature extraction and data analysis) operated correctly in all cases and caused no undesirable behavior. In general, the rovers operated fully autonomously and traveled over 40 meters.

While the system performed well during testing, we have identified some areas for in which the system's handling of uncertainty could be improved. While the planning system can respond appropriately when activities do not run in the estimated time (whether they take more time or less time than predicted) it would be better if the system could make more accurate predictions as the planner could do a better job optimizing the value of the mission plan. This would reduce the time the planner spends replanning and, in some cases, could result in higher quality plans.

The challenge in making such predictions is that the duration of traverse activities depend on the nature of the terrain and the amount of obstacles the rover will encounter, which can be difficult to predict ahead of time. A possible solution may be to allow the rover adjust its predictive model of its activities based on its experience during mission. Techniques such as regression tree learning have been shown to allow robots to learn such predictive models for navigation actions [Balac *et al.*, 2000].

Another improvement would be to explicitly reason about the uncertainty of activities. This would enable the planner to make tradeoffs between actions that may result in the collection of valuable science but may have a high uncertainty in the outcome.

Finally, a significant challenge in developing autonomy for space exploration is developing algorithms that will meet the computational constraints of the flight systems. Processors and memory used in space must be radiation hardened and the available processors are significantly slower than non-radiation hardware. For example, the current Mars Exploration Rovers each have 128 MB of RAM and a 20 Mhz processor. Rover missions within the next 10 years may have processor speeds of about 200 Mhz. Based on computation performance of our system on faster processors, we anticipate requiring 1 to 2 minutes of computation time to update the plan to respond to problems or opportunities. We also estimate requiring 15-20 MB of RAM. As another data-point, CASPER is currently being used in the Autonomous Sciencecraft Experiment to automate the Earth Observing-1 spacecraft which has a 12 Mhz processor [Sherwood *et al.*, 2004].

## 4 Related Work

The objectives of OASIS are similar to those of the Autonomous Sciencecraft Experiment (ASE) [Sherwood *et al.*, 2003] which also uses science analysis to generate additional goals for a planner. OASIS differs from ASE in the types of feature extraction and data analysis that are performed. In addition, while ASE has focused on planning for orbiter missions, the focus for OASIS has been on ground operations. To support this type of planning OASIS must deal with the high degree of uncertainty inherent in ground operations and integrate path planning into the planning and scheduling process. Finally, in OASIS it is often necessary to temporarily halt currently executing activities, such as a traverse, in order to accomplish new science goals.

A number of other systems have used planning methods to coordinate robot behavior (e.g. [Bonasso *et al.*, 1997; Alami *et al.*, 1998]). However, these systems generate plans with a batch approach where plans are generated for a certain time period and if re-planning is required, an entire new plan must be produced. In OASIS, plans are continuously modified in response to changing conditions and goals. The CPS planner generates contingent plans which are then executed onboard a rover and can be modified at certain points if failures occur [Bresina *et al.*, 1999]. Since only a limited number of contingencies can be anticipated, our approach provides more onboard flexibility to new situations.

## 5 Conclusions

OASIS supports opportunistic science by integrating data analysis algorithms, which identifies potentially interesting science measurements, with planning and scheduling algorithms, which enables the rover to respond to these new requests. Our current system has been tested with several scenarios in simulation and on prototype rover hardware. In these scenarios we demonstrate the systems ability to respond appropriately to problems with plan execution and to exploit unexpected opportunities that might arise.

Currently, the planner preserves the original mission goals when attempting to perform opportunistic science. We will relax this constraint and allow the system to use priorities to determine when it is appropriate to achieve opportunistic science at the cost of existing goals. There are significant challenges with introducing autonomous techniques into the mission operations culture. We are taking steps to address this by introducing MER scientists to off-line versions of our software.

## Acknowledgments

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. This work was funded by the NASA Intelligent Systems program and the JPL Interplanetary Network Directorate. We wish to thank the other members of the OASIS team for their work on feature extraction and data analysis and the CLARAty team, especially Issa Nesnas and Max Bajracharya, for their assistance with the rover.

## References

[Alami *et al.*, 1998] R. Alami, R. Chautila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17(4), April 1998.

[Balac *et al.*, 2000] Natasha Balac, Daniel M. Gaines, and Doug Fisher. Using regression trees to learn action models. In *IEEE Systems, Man and Cybernetics*, Nashville, October 2000.

[Bonasso *et al.*, 1997] R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence Research*, 9(1), 1997.

[Bresina *et al.*, 1999] J. Bresina, K. Golden, D. E. Smith, and R. Washington. Increased flexibility and robustness for Mars rovers. In *Proceedings of the Fifth International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, Noordwijk, Netherlands, 1999.

[Castano *et al.*, 2004] R. Castano, M. Judd, T. Estlin, R. Anderson, L. Scharenbroich, L. Song, D. Gaines, F. Fisher, D. Mazzoni, and A. Castano. Autonomous onboard traverse science system. In *IEEE Aerospace Conference*, Big Sky, Montana, March 2004.

[Chien *et al.*, 2000] Steve Chien, Russell Knight, Andre Stechert, Rob Sherwood, and Gregg Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *Fifth International Conference on Artificial Intelligence Planning and Scheduling*, Breckenridge, CO, April 2000.

[Estlin *et al.*, 2002] Tara Estlin, Forest Fisher, Daniel Gaines, Caroline Chouinard, Steve Schaffer, and Issa Nesnas. Continuous planning and execution for a mars rover. In *Third International NASA Workshop on Planning and Scheduling for Space*, Houston, TX, October 2002.

[Nesnas *et al.*, 2003] Issa A. Nesnas, Ann Wright, Max Bajracharya, Reid Simmons, Tara Estlin, and Won Soo Kim. Claraty: An architecture for reusable robotic software. In *Proceedings of SPIE Aerosense Conference*, Orlando, Florida, 2003.

[Sherwood *et al.*, 2003] Robert Sherwood, Steve Chien, Rebecca Castano, and Gregg Rabideau. The autonomous sciencecraft experiment. In *Proceedings of the IEEE Aerospace Conference*, Big Sky, MT, March 2003.

[Sherwood *et al.*, 2004] R. Sherwood, S. Chien, D. Tran, B. Cichy, R. Castano, A. Davies, and G. Rabideau. Operating the autonomous sciencecraft experiment. In *Proceedings of International Conference on Space Operations (SpaceOps 2004)*, Montreal, May 2004.

[Simmons and Apfelbaum, 1998] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Proceedings of Conference on Intelligent Robotics and Systems*, Vancouver Canada, October 1998.

[Zweben *et al.*, 1994] M. Zweben, B. Daun, E. Davis, and M. Deale. Scheduling and rescheduling with iterative repair. In M. Fox and M. Zweben, editors, *Intelligent Scheduling*, pages 241–256. Morgan Kaufmann Publishers Inc., 1994.

# Bayesian Models of Nonstationary Markov Decision Processes

**Nicholas K. Jong** and **Peter Stone**
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
{nkj,pstone}@cs.utexas.edu

## Abstract

Standard reinforcement learning algorithms generate polices that optimize expected future rewards in a priori unknown domains, but they assume that the domain does not change over time. Prior work cast the reinforcement learning problem as a Bayesian estimation problem, using experience data to condition a probability distribution over domains. In this paper we propose an elaboration of the typical Bayesian model that accounts for the possibility that some aspect of the domain changes spontaneously during learning. We develop a reinforcement learning algorithm based on this model that we expect to react more intelligently to sudden changes in the behavior of the environment.

## 1 Introduction

Reinforcement learning (RL) research provides algorithms for generating universal plans from experience, given minimal prior knowledge about the domain [Sutton and Barto, 1998]. Classical RL algorithms assume only that the domain obeys the Markov property: the effects of each action depend only on the currently observed state. However, the behavior of many interesting domains depends on factors that are difficult or impossible to represent in the state space. A robot's effectors may change unexpectedly due to damage. An overturned truck may render a highway suddenly impassable. An opened door in a previously explored area may grant access to new opportunities. Standard RL algorithms adapt only gradually to such drastic changes to the overall system. Enough experience after the change must accumulate to outweigh the outdated knowledge. The agent may *never* notice a change that occurs in a region of the state space that the learned behavior doesn't visit.

In this paper, we consider statistical methods for detecting changes in the domain in a more timely manner. Intuitively, an intelligent agent should notice when the environment ceases to behave as expected. Such an agent should consider throwing out or discounting its old model of the relevant aspects of the environment. We adopt a Bayesian framework that allows us to reason explicitly about uncertainty over the domain [Strens, 2000]. We elaborate the standard probabilistic model to represent the possibility of domain change. We then propose an algorithm that employs statistical inference techniques to behave more robustly in the presence of domain change.

## 2 Background

The standard domain formalism in RL research is the Markov decision problem (MDP). An MDP $\langle S, A, P, R\rangle$ comprises a finite set of states $S$, a finite set of actions $A$, a transition function $P : S \times A \times S \to [0,1]$, and a reward function $R : S \times A \to \mathbb{R}$. Executing an action $a$ in a state $s$ yields an expected immediate reward of $R(s,a)$ and causes a transition to state $s'$ with probability $P(s,a,s')$. A policy $\pi : S \to A$ specifies an action $\pi(s)$ for every state $s$ and induces a value function $V^\pi : S \to \mathbb{R}$ that satisfies the Bellman equations $V^\pi(s) = R(s,\pi(s)) + \gamma \sum_{s' \in S} P(s,\pi(s),s')V^\pi(s')$, where $\gamma \in [0,1]$ is a discount factor for future reward that may be necessary to make the equations satisfiable. For every MDP at least one optimal policy $\pi^*$ exists that maximizes the value function at every state simultaneously. To compute an optimal policy from a fully specified MDP, a number of algorithms are available, including dynamic programming, policy iteration, and linear programming [Littman *et al.*, 1995].

In the RL problem, only the state space $S$ and the action space $A$ are known a priori, but standard approaches assume that the transition function $P$ and reward function $R$ are fixed. An important class of RL algorithms are model-based: they compute policies by first estimating $P$ and $R$ and then solving the estimated MDP. Although solving an MDP is too computationally expensive to perform after every time step, algorithms such as Prioritized Sweeping [Moore and Atkeson, 1993] describe how to propagate incremental updates to a model through a policy learned through dynamic programming. However, model-based algorithms are particularly vulnerable to nonstationary domains, since they typically employ maximum likelihood estimates of parameters of the model given all the available experience data. Hence, changes in the domain will tend to averaged into a large body of outdated prior experience.

In this paper, we elaborate a model-based algorithm called Bayesian dynamic programming [Strens, 2000]. For each state-action pair $(s,a)$, this approach interprets $P(s,a,\cdot)$ as the parameters of an a priori unknown multinomial distribution and $R(s,a)$ as the mean of an unknown normal distribution.[1] We represent our initial uncertainty about these unknown distributions as prior distributions over their parameters. The joint distribution over all the presumed-independent state-action pairs yields a probability distribution over MDPs. Since conjugate families of prior distributions exist for both the multinomial and normal distributions, we can compactly represent and efficiently update these distributions over MDPs. At the beginning of each training episode, Bayesian dynamic programming samples a hypothetical model of the domain from this distribution and then behaves according to the policy obtained from solving the model.

## 3 A Probabilistic Model of Change

We propose a simple model of environmental change: after every episode and for each state-action pair, the associated multinomial successor-state distribution and normal reward distribution reset with some small probability to distributions drawn from the respective original priors. This model caters to the fact that only small parts of a domain may change at a time. A more sophisticated model might also capture the fact that a change in one state-action pair makes a change in another pair more likely, but such models may be quite complex.

Our Bayesian model of the domain must change to accomodate this probability of reset. Suppose that we have $k$ complete training episodes of data. Consider a particular state-action pair $(s, a)$. Let $T$ denote the episode number when the state-action pair $(s, a)$ last reset, so $T = 0$ is the hypothesis that the behavior of $a$ at state $s$ has never changed and $T = k$ is the hypothesis that the behavior reset at the beginning of the current episode. Let $P$ denote the successor state distribution given $(s, a)$. Then we have a hierarchical distribution over $P$, given by $\Pr(P = \vec{p}) = \sum_t \Pr(P = \vec{p}|T = t) \cdot \Pr(T = t)$, where $\Pr(P = \vec{p}|T = t)$ is the result of the standard Bayesian conditioning process, but using only a suffix of all the data. Similar reasoning applies to the distribution over $R$, the reward function evaluated at $(s, a)$. When we sample a hypothesis MDP from our Bayesian model, we must now first draw a sample hypothesizing the last time each state-action pair reset (according to the distribution $\Pr(T)$, before sampling the pieces of the transition and reward functions from posterior distributions conditioned on the corresponding suffixes of the data.

No conjugate family of prior distributions exists for $\Pr(T)$, so in practice we approximate this distribution by maintaining the relative probabilities for a small subset of episode numbers. The computation of these relative probabilities poses another obstacle. Suppose that we have a prior distribution $\Pr(T)$ that we want to update given experience data $D$ that we assume all come from the same distribution. Then from Bayes' Theorem we have $\Pr(T = t|D) \propto \Pr(D|T = t) \cdot \Pr(T = t)$. We can rewrite the model likelihood as an integral over Bayesian models conditioned on a suffix of the data: $\Pr(D|T = t) = \int\int \Pr(D|P = \vec{p}, R = r, T = t) \cdot \Pr(P = \vec{p}|T = t) \cdot \Pr(R = r|T = t)\ d\vec{p}\, dr$. Computing the model likelihood exactly is infeasible, but we can approximate it with Monte Carlo integration by sampling some number of values for $P$ and $R$, again conditioned on the appropriate suffix of the data.

Our Bayesian model of the state-action pair $(s, a)$ is therefore approximate in two ways. First, we approximate $\Pr(T)$ with a bounded-size sample of the most probable values of $T$. Each point in this sample has a scalar weight and a Bayesian model of $P$ and $R$, represented exactly as the appropriate parameters to the conjugate priors for these distributions. The second approximation is in our Bayesian update of our model given new data. We reweight the sample by multiplying each weight by the model likelihood, estimated using Monte Carlo integration.

## 4 Application of the Bayesian Model

We can use the Bayesian model elaborated above directly with Strens' Bayesian dynamic programming algorithm [Strens, 2000]. After each episode, we add to our sample of $\Pr(T)$ the hypothesis for each state-action pair that it reset before that episode. We give this hypothesis some portion of the weight equal to our prior probability of domain change at each episode. Then we condition the weights and each model of $P$ and $R$ on the data from that episode. To keep the sample size reasonable, we select a value of $T$ to discard. Finally, for the next episode, we sample an MDP from the hierarchical model.

This Bayesian approach to recognizing domain change allows us to avoid unilateral commitments to either keeping or discarding old data. Additionally, in the absence of evidence to the contrary, it gradually increases the belief that neglected state-action pairs have reset. If the prior distribution over the reward function is optimistic, then the agent will eventually choose to explore the action again.

Unfortunately, Bayesian dynamic programming does not always work so well when the state-action pair that changed is part of the learned policy. If a previously reliable action suddenly fails entirely, the solution to the sampled MDP may cause the agent stubbornly to retry expensive actions until timing out. The same phenomenon can occur in model-free methods such as Q-learning: depending on the learning rate, the agent may spend quite some time in a negative-reward loop.

We propose a small modification of Bayesian dynamic programming. If the number of visits to a state in a single episode exceeds a certain threshold, we begin to conduct statistical goodness-of-fit tests to evaluate our hypotheses for $P$ and $R$ at the appropriate state-action. If the data collected during that episode cause us to reject the sample of $P$ or $R$, we immediately resample them from a Bayesian model conditioned on only that data. We then update the policy as necessary to solve the updated MDP. Note that even though we resample from a distribution assuming that a reset occurred, we still update the Bayesian model as usual at the end of the episode.

[1] The MDP formalism does not require the rewards to be normally-distributed, but this assumption seems fairly innocuous given that we care only about the mean of the reward distribution.

## 5 Future Work and Discussion

The implementation and evaluation of the algorithm described above remains to be done. However, we believe that this approach of building a Bayesian model of domain uncertainty is very promising. One concern is the computational cost of Bayesian inference, but the proposed modifications to Bayesian dynamic programming should not worsen the runtime much. The Monte Carlo integration only occurs for state-action pairs executed during an episode, and the primary cost of this procedure is the sampling of $P$ and $R$ that the algorithm already performs once for every state-action pair. The goodness-of-fit tests only occur upon revisiting a state several times in the same episode, and some form of exponential backoff can help prevent spurious testing. (Testing again after one additional data point is unlikely to produce a different result.)

The ideas described in this paper are reminiscent of our previous usage of a Bayesian model of MDPs to infer state abstractions [Jong and Stone, 2005] from the solution of one MDP for use in similar MDPs. A particularly promising avenue of future research is the usage of a single Bayesian model to reason about both dynamic domains and state abstraction simultaneously. In this framework we can imagine inducing structure such as state abstraction that continues to aid learning despite continual changes in the reward and transition functions.

## Acknowledgments

This research was supported in part by NSF CAREER award IIS-0237699 and DARPA grant HR0011-04-1-0035.

## References

[Jong and Stone, 2005] Nicholas K. Jong and Peter Stone. State abstraction discovery from irrelevant state variables. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, 2005.

[Littman *et al.*, 1995] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, 1995.

[Moore and Atkeson, 1993] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.

[Strens, 2000] Malcolm Strens. A Bayesian framework for reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 943–950, 2000.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

# Simulation Methods for Uncertain Decision-Theoretic Planning

**Douglas Aberdeen** and **Olivier Buffet**
National ICT Australia
Australian National University
Canberra, Australia
{douglas.aberdeen,olivier.buffet}@nicta.com.au

## Abstract

Experience based reinforcement learning (RL) systems are known to be useful for dealing with domains that are *a priori* unknown. We believe that experience based methods may also be useful when the model is uncertain (or even completely known). In this case experience is gained by *simulating* the uncertain model. This paper explores a simple way to allow experience based RL systems to cope with uncertainty in a model. The particular form of RL we consider is a policy-gradient method. The particular domains we attempt to optimise in are from temporal decision-theoretic planning. Our previous experience with military planning problems indicates that a human specified model of the planning problem is often inaccurate, especially when humans specify probabilities, thus planners that take into account this uncertainty are very useful. Despite our focus on policy-gradient RL for planning, our simple (but approximate) solution for dealing with uncertainty in the model can be applied to any simulation based RL method, such as Q-learning or SARSA. Our attempt to solve decision-theoretic planning problems with a policy-gradient approach is novel in itself, making up another contribution of this paper.

## 1 Introduction

If the true model of a Markov decision problem (MDP) is hidden we must use algorithms that train agents by *interacting* with the MDP. This is done by experiencing trajectories through the state space and forming either an explicit model (transition matrix) or an implicit model (value function or policy) of the system. These Monte-Carloesque methods can be beneficial even when the true model is completely known, especially if the model is too complex to work with directly. E.g., the state space might be too large to enumerate, or it might be continuous. In this case the model is only used in simulating the system, generating state space trajectories that the agent uses to optimise its behaviour. Another argument for simulation based optimisation is the ease of creating a simulator compared to a set of stochastic transition matrices. This is especially true if aspects of the system are unknown or approximated.

This paper explores the idea of using an *uncertain* model to simulate trajectories, allowing an agent to directly optimise its policy in a way that minimises the impact, or risk, associated with the uncertainty in the model. Moreover, this can be achieved in highly complex domains.

The problems we consider come from temporal decision-theoretic planning, where methods that enumerate any part of the state space fail to scale to interesting problems. The model is provided in the form of a set of tasks the planner can choose from. Each task has a pre-defined duration and has probabilistic outcomes that set multiple state variables to true or false. The goal of the planner is to select actions, and schedule them concurrently, to achieve the desired *goal state* values of the state variables. Resources constrain which tasks can be run in combination, and resources are consumed as tasks end. This is a very general expression of the planning problem and only a few probabilistic planners are emerging that can operate in this setting. They can be used to optimise plans in a wide variety of situations, such as Mars rover planning [Mausam and Weld, 2005], military operations planning [Aberdeen *et al.*, 2004], or building site planning. Probabilities might arise from modelling variable battery strength, an opponent's actions, or weather. The outcome probabilities are often estimated from finite data, or guessed by human experts. Thus, the probabilities are subject to some uncertainty.

The contribution of this paper is two fold. Firstly, we describe the factored policy gradient (FPG) Planner: a novel approach to temporal decision-theoretic planning that allows very large domains to be *approximately* optimised. We achieve this by: (1) factoring the policy into simple independent policies for starting each task; (2) using a local optimisation method instead of trying to find a globally optimal solution; (3) using algorithms with memory use that scales linearly with the number of tasks, state variables, and resources, not with the state space.

The second contribution is to demonstrate how uncertainty over the probabilities described in a model can be incorporated into a simulation based optimisation. We assume that probabilities of task outcomes lie in intervals between $[0, 1]$. The width of the interval can be computed based on the quality of the data used, or based on how confident the human guess was. The goal is to find a policy that minimises the risk, or variance, associated with enacting the policy over the range of models implied by the uncertainty. I.e., the policy that still performs relatively well even in the worst case scenario. The key idea is simply to simulate state space trajectories using the most *pessimistic* model. The most pessimistic model might generally be as difficult to compute as the policy, however, we show empirically that a local approximation to the pessimistic model might be sufficient. We can also compute policies based on the most *optimistic* model, to examine the differences in policy or determine how much our uncertainty could be effecting agent performance.

We start by describing background work in temporal probabilistic planning and interval methods for Markov decision problems (MDPs). Section 3 describes MDPs for planning. Section 4 describes the factored policy agents and the estimator we use to compute the gradient of the objective function. Section 5 describes preliminary experiments.

## 2 Background

Previous probabilistic temporal planners include CPTP [Mausam and Weld, 2005], Prottle [Little, 2004], and a military operations planner [Aberdeen *et al.*, 2004]. All these algorithms use some form of dynamic programming (either RTDP [Barto *et al.*, 1995] or AO*) to associate values with each state/action pair. However, this requires that values be stored for each encountered state. Even though these algorithms do not enumerate the entire state space their ability to scale is limited by memory size. Even problems with a few tasks and state variables can produce millions of relevant states. Another probabilistic temporal planner is Tempastic [Younes and Simmons, 2004], which uses the generate, test, and debug planning paradigm. This method may suffer in domains that are highly non-deterministic.

Our FPG-Planner performs gradient ascent in the space of parameters of the factored policies (or policy agents). The policy agents can be any differentiable function approximator. We show that maximising a simple reward function naturally minimises plan durations and maximises the probability of reaching the goal. Gradients are estimated by simulating trajectories through the planning state space and calculating small contributions to the gradient at each step [Baxter *et al.*, 2001]. The FPG-Planner will be described in this paper, but is covered in more detail in Aberdeen [2005].

The use of intervals to describe uncertainty in MDPs was investigated by Givan *et al.* [2000], Hosaka *et al.* [2001], and Strehl and Littman [2004]. Our approach is most closely related to the approach of Buffet and Aberdeen [2005], who use uncertainty intervals to make RTDP robust. RTDP uses simulation to determine which state values should be updated, thus is similar in its use of simulation to help cope with large state spaces. The intervals are used to compute the most pessimistic transition probabilities given the *current* value estimates. Our work deliberately avoids storing values, thus cannot use them to approximate the worst model. Instead, we assume that the simulator can often select the probability in each interval on a state by state basis that will result in a pessimistic model being simulated. Specifically, in our planning domains we assume that each task has two outcomes: success, which is helpful, and failure, which is harmful.

Actions in temporal planning consist of launching multiple tasks concurrently. The number of candidate actions available in a given state is the power set of the tasks that are eligible to start under the current state variables. That is, with $N$ eligible tasks there are $2^N$ possible actions. With only 10 eligible tasks we have 1,024 actions to choose from! Current planners try to explore this action space systematically, pruning actions that lead to low rewards (or equivalently, high costs).

A key contribution of the FPG-Planner is to deal with the explosion of the action space by replacing the single agent choosing from the power-set of tasks with a single simple agent for each task. The policy learnt by each agent is whether to start its associated task given its observation, independent of the decisions made by the other agents. This idea alone does not simplify the problem. Indeed, if the agents all receive perfect state information they could presumably predict the decision of the other agents and still act optimally. The significant reduction in complexity arises from two additional factors: (1) the agents are only provided enough information to make an approximate decision, not an optimal decision; (2) each agent is optimised locally.

## 3 POMDP Formulation of Planning

Our intention is to deliberately simplify the agents by restricting their access to state information. This requires us to explicitly consider partial observability. We now describe the partially observable MDP framework (POMDP), define the state space, our objective function, and the process for simulating the state space.

**Definition 1** *A finite partially observable Markov decision process consists of: a finite set of states $s \in \mathcal{S}$; a finite set of actions $\mathbf{a} \in \mathcal{A}$; probabilities $\Pr[s'|s, \mathbf{a}] : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ of making state transition $s \to s'$ under action $\mathbf{a}$; a reward for each state $r(s) : \mathcal{S} \to \mathbb{R}$; a finite set of observation vectors $\mathbf{o} \in \mathcal{O}$ seen by the agent in place of the complete state description; and probabilities $\Pr[\mathbf{o}|s] : \mathbb{R}^{|\mathbf{o}|} \times \mathcal{S} \to [0, 1]$ of observing vector $\mathbf{o}$ of dimension $|\mathbf{o}|$, given current state $s$.*

In addition, our specification of intervals around each task outcome probability induces intervals around each state transition probability $\Pr[s'|s, \mathbf{a}]$. As will be demonstrated later, our use of simulation on the level of planning tasks, instead of states, means we never need to explicitly compute $\Pr[s'|s, \mathbf{a}]$ probabilities or their intervals.

*Goal states* are states where all the goal variables are satisfied. *Failure states* are states from which it is impossible to reach a goal state (usually because time or resources have run out). These two classes of state are combined to form the set of *reset* states that produce an immediate reset to the initial state $s_0$. A single trajectory through the state space consists of many individual trials that automatically reset to $s_0$ each time a goal state or failure state is reached.

Policies are possibly stochastic, mapping observation vectors to a probability over each action. Let $N$ be the number of basic tasks available to the planner. In our setting an action $\mathbf{a}$ is a binary vector of length $N$. An entry of 1 at index $n$ means 'Yes' begin task $n$, and a 0 entry means 'No' do not start task $n$. The probability of actions is written $\Pr[\mathbf{a}|\mathbf{o}, \theta]$, where conditioning on $\theta$ reflects the fact that the policy is dictated by a set of $p$ real valued parameters $\theta \in \mathbb{R}^p$. We show later how real valued parameters can control probability distributions over actions given observations, thus determining a policy. This paper assumes that all stochastic policies (i.e., all values for $\theta$) reach reset states in finite time when executed from $s_0$. This is enforced by limiting the maximum duration of a plan. Because all policies reach a reset state, and by continuously resetting to the initial state, we ensure the underlying MDP is *ergodic*,[1] which is necessary for producing gradient estimates.

The aim of policy gradient algorithms is to find the set of parameters $\theta$ that induce a policy to move from the initial state $s_0$ to a reset state while maximising the long-term average reward. The long-term average reward is the average of all instantaneous rewards received over an infinite sample trajectory of the POMDP[2]

$$\eta(\theta) = \lim_{T\to\infty} \frac{1}{T} \sum_{t=0}^{T-1} r(s_t).$$

In the context of planning, the instantaneous reward provides the agent with a measure of progress toward the goal. A simple reward scheme is to set $r(s) = 1$ for all states $s$ that represent the goal state, and 0 for all other states. To maximise $\eta(\theta)$, goal states must be reached as frequently as possible. This has the desired property of simultaneously minimising plan duration, as well as maximising the probability of reaching the goal (failure states achieve no reward). It is tempting to provide a negative reward for failure end states, but in this case an agent could partially maximise its reward by avoiding progress altogether, never achieving end states, and therefore never achieving negative (or positive) rewards.

We propose a reward scheme that provides a large reward (1000 in this paper) for reaching the goal as described, plus a reward at each step that heuristically awards progress toward the goal. This additional *shaping* reward provides a reward of 1 for every goal condition achieved, and -1 for every goal condition that becomes unset.

### 3.1 Planning State Space

For probabilistic temporal planning our state description contains: the state's absolute time, a queue of impending events, the status of each task, the truth value of each state variable, and the available resources. In a particular state, only a subset of the eligible tasks will satisfy all preconditions for execution. This subset is called the *eligible* task list. When a decision to start a fixed duration task is made, an end-task event is added to the time ordered event queue. The event queue holds a list of events that the planner is committed to, although the outcome of those events may be uncertain.

The generation of successor states is shown in Alg. 1. The algorithm begins by starting the tasks given by each bit in the action, implementing any immediate effects. An end-task event is added at an appropriate time in the queue. The state update then processes events until there is at least one task that is eligible to begin. Lines 6–17 check for reset states before the next event for the current state $s$ is processed.

**Algorithm 1** findSuccessor(State $s$, Action $\mathbf{a}$)

```
1:  for each $a_n$ ='Yes' in $\mathbf{a}$ do
2:      $s$.beginTask($n$)
3:      $s$.addEvent($n$, s.time+taskDuration($n$))
4:  end for
5:  repeat
6:      if $s$.time $>$ maximum makespan then
7:          s.failureLeaf=true
8:          return
9:      end if
10:     if $s$.operationGoalsMet() then
11:         s.goalLeaf=true
12:         return
13:     end if
14:     if $s$.noEvents() & $\neg s$.anyEligibleTasks() then
15:         s.failureLeaf=true
16:         return
17:     end if
18:     $e$ = $s$.nextEvent()
19:     $s$.time = $e$.time
20:     selectModel($e$.PrSuccessLower, $e$.PrSuccessUpper)
21:     sample $sucess\,\Pr = p$, $failure\,\Pr = 1 - p$
22:     $s$.implementEffects($outcome$)
23: until $s$.anyEligibleTasks()
```

**Algorithm 2** selectModel(lowerBound, upperBound)

```
if pessimistic then
    return ($e$.lowerBound)
else
    if optimistic then
        return ($e$.upperBound)
    else
        return(lowerBound+upperBound)/2
    end if
end if
```

Events have probabilistic outcomes. Uncertain models provide intervals of probabilities for outcomes. The intervals are defined as part of the problem specification. Before sampling we must choose a point in this interval to base the sample on. We assume that maximising the probability of failure also minimises the long-term average reward for the current policy. Thus, to train the agent to operate well under the pessimistic model we always choose the lower bound on the probability of success as the true probability of the event (Alg. 2), and sample the outcome accordingly. Similarly, if we wish the agent to perform well if the optimistic model turns out to be correct, we select the upper bound on the probability of success. If there are more than two outcomes we could put intervals on the probability mass associated with each outcome. We then distribute the probability mass as constrained by the intervals. The worst outcomes gets the maximum probability mass, the next worst outcome gets

[1] Except for the aperiodic condition for ergodicity that is not important for this paper.

[2] Because the underlying MDP is ergodic, $\eta(\theta)$ is independent of the starting state.

the maximum allowed remaining mass, and so on.

This scheme is not guaranteed to select probabilities from the intervals that correspond to the worst overall model. A pessimistic choice at the current state could lead to future states with very little uncertainty in the model, whereas an optimistic choice could lead to future states with massive uncertainty and hence larger scope for poor models. We assume there is a way to approximately measure which outcomes will lead to high or low rewards. Section 6 outlines how we might learn the worst (or best) overall model in the same setting.

Line 21 of Alg. 1 samples one possible outcome from the distributions permitted by the intervals in the problem definition. Alg. 2 is the only point in the algorithm where intervals are considered. The remainder of the algorithm description is independent of our use of uncertain models.

Future states are only generated at points when tasks can be started. If an event outcome is processed and no tasks are enabled, the search recurses to the next event in the queue.

## 4 Policy Gradient Ascent

In this section we describe policy-gradient algorithms for reinforcement learning and how we use this approach for the FPG-Planner. We assume the presence of policy agents, parameterised with independent sets of parameters for each agent $\theta = \{\theta_1, \ldots, \theta_N\}$. There are $p$ parameters in total. We seek to adjust the parameters of the policy to maximise the long-term average reward $\eta(\theta)$.

Baxter *et al.* [2001] describe the GPOMDP algorithm that estimates the gradient $\nabla\eta(\theta)$ of the long-term average reward with respect to the current set of policy parameters. Once an estimate $\hat{\nabla}\eta(\theta)$ is computed, we maximise the long-term average reward with a gradient ascent step: $\theta \leftarrow \theta + \alpha\hat{\nabla}\eta(\theta)$, where $\alpha$ is a small step size. Maximising $\eta(\theta)$ produces better policies, both in terms of duration and probability of failure. Repeating the process of estimating the gradient, followed by a gradient ascent step, optimises the policy until a maxima in the long-term average reward is found.

### 4.1 Estimating Gradients

The GPOMDP gradient estimate algorithm works by sampling a single long trajectory through the state space. The state transitions are generated with Alg. 1 after each task agent has chosen whether to start or not. All agents receive the same reward for the new state and update their gradient estimates independently.

The parameterised policy maps observations to probability distributions over action vectors. The action vector at each step is $\mathbf{a}_t$, a combination of independent 'Yes' or 'No' actions made by the agents. Each agent is parameterised by an independent set of parameters that make up $\theta \in \mathbb{R}^p$: $\theta_1, \theta_2, \ldots, \theta_N$. If $a_{tn}$ represents the binary decision made by agent $n$ at time $t$ about whether to start its corresponding task then the stochastic policy factors into

$$\begin{aligned}\Pr[\mathbf{a}_t|\mathbf{o}_t, \theta] &= \Pr[a_{t1}, \ldots, a_{tN}|\mathbf{o}_t, \theta_1, \ldots, \theta_N] \\ &= \Pr[a_{t1}|\mathbf{o}_t, \theta_1] \times \cdots \times \Pr[a_{tN}|\mathbf{o}_t, \theta_N].\end{aligned}$$

It is not necessary for all agents to receive the same observation, and it may be advantageous to show different agents different parts of the state, leading to a decentralised planning algorithm. Introducing partial observability in similar multi-agent policy-gradient approaches [Peshkin *et al.*, 2000] has been shown to increase the number of local minima. Choosing a good observation vector allows the problem to remain tractable while hopefully avoiding the introduction of severe local maxima.

After some experimentation [Aberdeen, 2005], we chose an observation vector that is a binary description of the eligible tasks (15 bits) and the condition truth values (10 bits) plus a constant 1 bit to provide bias to the agents' linear networks.

The main requirement for each policy-agent is that $\Pr[a_{tn}|\mathbf{o}_t, \theta_n]$ be differentiable with respect to the parameters for each choice task start $a_{tn}$ ='Yes' or 'No'. We choose to represent each agent with a two output linear network mapped into probabilities using a soft-max function:

$$\Pr[a_{tn} = Yes|\mathbf{o}_t, \theta_n] = \frac{exp(\mathbf{o}_t^\top \theta_{n,Yes})}{exp(\mathbf{o}_t^\top \theta_{n,Yes}) + exp(\mathbf{o}_t^\top \theta_{n,No})}$$

$$\Pr[a_{tn} = No|\mathbf{o}_t, \theta_n] = \frac{exp(\mathbf{o}_t^\top \theta_{n,No})}{exp(\mathbf{o}_t^\top \theta_{n,Yes}) + exp(\mathbf{o}_t^\top \theta_{n,No})}.$$

This can be thought of as a two output linear network where the outputs are subsequently normalised to produce a well behaved probability distribution. If the dimension of the observation vector is $|\mathbf{o}|$ then each $\theta_n$ can be thought of as an $|\mathbf{o}| \times 2$ matrix where the columns represent the network weights for the 'Yes' decision output and the 'No' decision output respectively. This expression is a form of logistic regression. The log derivatives, necessary for Alg. 3, are given in Baxter *et al.* [2001]. Initially the parameters are set to small random values: a near uniform random policy. This encourages exploration of the action space. Each gradient step typically moves the parameters closer to a deterministic policy.

Figure 1 shows the selection of actions graphically. Alg. 3 describes the algorithm for computing $\hat{\nabla}\eta(\theta)$. The gradient estimate provably converges to a biased estimate of $\nabla\eta(\theta)$ as $T \to \infty$. The quantity $\beta \in [0, 1)$ controls the degree of bias in the estimate. As $\beta$ approaches 1, the bias of the estimates drop to 0. However if $\beta = 1$, estimates exhibit infinite variance as $T \to \infty$. Thus the parameter $\beta$ achieves a bias/variance tradeoff in the stochastic gradient estimates.

Line 8 computes the log gradient of the sampled action probability and adds the gradient for the $n$'th agent's parameters into an *eligibility trace*. The gradient for parameters not relating to agent $n$ is 0. We do not compute $\Pr[a_{tn}|\mathbf{o}_t, \theta_n]$ or gradients for tasks with unsatisfied preconditions. If all eligible agents decide *not* to start their tasks, we issue a null-action. If the state event queue is not empty, we process the next event, otherwise time is incremented by 1 to ensure all possible policies will eventually reach a reset state.

## 5 Experiments

This section provides some preliminary experiments that validate the ideas in this paper. We compare the present algorithm with that of our earlier RTDP based planner for military operations [Aberdeen *et al.*, 2004]. Both the current tools and our previous tool use the same code to generate states, representing exactly the same domains, providing a fair comparison.

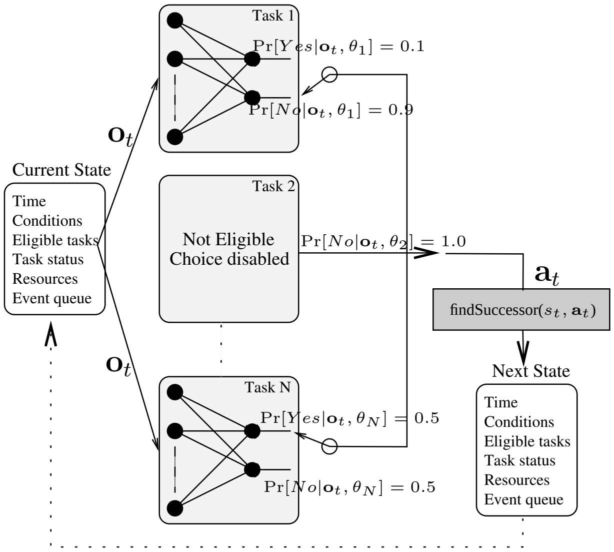

Task 1
$\Pr[Yes|\mathbf{o}_t, \theta_1] = 0.1$
$\Pr[No|\mathbf{o}_t, \theta_1] = 0.9$
$\mathbf{O}_t$
Current State
Time
Conditions
Eligible tasks
Task status
Resources
Event queue
Task 2
Not Eligible
Choice disabled
$\Pr[No|\mathbf{o}_t, \theta_2] = 1.0$
$\mathbf{a}_t$
findSuccessor($s_t$, $\mathbf{a}_t$)
Next State
Time
Conditions
Eligible tasks
Task status
Resources
Event queue
$\mathbf{O}_t$
Task N
$\Pr[Yes|\mathbf{o}_t, \theta_N] = 0.5$
$\Pr[No|\mathbf{o}_t, \theta_N] = 0.5$

Figure 1: Task-policies receive an observation of the current state and individually choose whether to start or not. The combined decision is fed into the state simulator that probabilistically generates the next state. In this example, the joint action probability is $\Pr[No, No, \ldots, Yes|\mathbf{o}_t, \theta] = 0.45$.

The problem[3] consists of 15 tasks designed to represent the high level process of building a sky-scraper. These tasks achieve a set of 10 state variables needed for operation success. Four of the effects can be established independently by two different tasks, however, resource constraints only allow one of the tasks to be chosen. Furthermore, tasks are not repeatable, even if they fail. The probability of failure of tasks ranges between 0 and 20% with an interval on either side of 20% (unless such an interval would result in a probability of failure of less than 0%).

We have constructed this example to demonstrate the effectiveness of planning with intervals. Thus, for each effect that has two tasks that can achieve it we have selected one task to have a high probability of success, but also a high uncertainty. The second task has a lower probability of success, but an interval of 0 (perfect knowledge of the model). The second (lower) probability of success is chosen to be higher than the *lower bound* on the first tasks success probability. The robust plan should (and does) choose tasks with the lower probability of success, but zero interval. Table 1 shows that the results of using the FPG-Planner with different modes of optimisation: 1- No optimisation at all, the plan is to start each eligible task with a probability of 50%; 2- Optimisation based on a simulation of a pessimistic model; 3- Optimisation based on the original human model (mean model); 4- Optimisation based on a simulation of an optimistic model. Evaluations are repeated three times. The evaluations assume that the true model is: 1- pessimistic, 2- the original model (mean model), 3- optimistic.

**Algorithm 3** Factored Planning Gradient Estimator

```
Set $s_0$ to initial state, $t = 0, \mathbf{e}_t = [0]$
while $t < T$ do
    $\mathbf{e}_t = \beta \mathbf{e}_{t-1}$
    Generate observation $\mathbf{o}_t$ of $s_t$
    for Each eligible task $n$ do
        Compute $\Pr[Yes|\mathbf{o}_t, \theta_n]$ and $\Pr[No|\mathbf{o}_t, \theta_n]$
        Sample $a_{tn}$ =Yes or $a_{tn}$ =No
        Compute $\mathbf{e}_t = \mathbf{e}_t + \nabla \log \Pr[a_{tn}|\mathbf{o}, \theta_n]$
    end for
    Try action $\mathbf{a}_t = \{a_{t1}, a_{t2}, \ldots, a_{tN}\}$
    while mutex or resource prohibits $\mathbf{a}_t$ do
        randomly turn off one task start in $\mathbf{a}_t$
    end while
    $s_{t+1}$ = findSuccessor($s_t, \mathbf{a}_t$)
    $\hat{\nabla}_t \eta(\theta) = \hat{\nabla}_{t-1} \eta(\theta) - \frac{1}{t+1}(r(s_{t+1})\mathbf{e}_t - \hat{\nabla}_{t-1}\eta(\theta))$
    $t \leftarrow t + 1$
end while
Return $\hat{\nabla}_T \eta(\theta)$
```

The parameters of the GPOMDP algorithm are: $T = 50,000$ gradient estimation steps and $\beta = 0.9$. Optimisation time was limited to 5 minutes wall clock time on a single user 3GHz Pentium IV with 1GB ram. All optimisations ran to the complete 5 minutes. After 5 minutes optimisation was terminated and the current policy evaluated. The FPG-Planner has the ‘any time’ property that returns better policies the longer optimisation is allowed to run, thus it is possible we might have gotten improved results with more patience. Results quote the average duration (± the variance), and the percentage of plans that terminate in a failure state. Plans that fail often have short average durations because they fail early in the execution of the plan due to resource limitations or a lack of alternative courses of action. Because optimisation has a stochastic component the results are the average over 100 training runs and 10,000 evaluation runs of the plan.

Unsurprisingly we see that plans formed under pessimistic training perform much better than other training modes when the true (evaluation) model turns out to follow the pessimistic model. Less obviously, the plan formed under a pessimistic model has a more uniform failure probability and durations over the possible true models. This is highly desirable because it means we have less variance in the outcome of plans despite operating over a wide range of models. We emphasise that this result is largely dependent on the particular domain and is a result of a true assumption that experiencing a less pessimistic true model can only benefit the policy. We have seen similar effects on other domains using the RTDP planner [Buffet and Aberdeen, 2005].

The RTDP results are quite similar to the FPG-Planner results. The FPG-Planner is performing somewhat better than RTDP if the true model turns out to be pessimistic, but training assumed a mean or optimistic model. RTDP gets the best overall result when the true model is optimistic and training assumed a mean or optimistic model. In this case RTDP is finding a global maxima in long-term average reward, but FPG gets stuck in local maxima.

[3] The problem definition can be found on `http://rsise.anu.edu.au/~daa`, written using an XML version of the PDDL language).

Table 1: Average failure prob and duration of the optimised Building plan. The columns are different training conditions. The rows are different evaluation conditions. Optimisation is performed with the FPG-Planner

| True model | No train | | Pess. train | | Mean train | | Opt train | |
|---|---|---|---|---|---|---|---|---|
| | Fail% | Dur. | Fail% | Dur. | Fail% | Dur. | Fail% | Dur. |
| Pessimistic | 0.657 | 4.80±2.22 | 0.549 | 4.42±1.78 | 0.688 | 3.35±2.89 | 0.694 | 3.42±2.37 |
| Mean | 0.403 | 5.97±1.69 | 0.420 | 5.16±1.34 | 0.378 | 4.98±1.64 | 0.381 | 5.03±1.67 |
| Optimistic | 0.325 | 6.30±1.36 | 0.386 | 5.35±1.12 | 0.278 | 5.46±1.03 | 0.277 | 5.51±1.05 |

Table 2: Same results as Table 1, but this time optimised with an RTDP based planner.

| True model | No train | | Pess. train | | Mean train | | Opt train | |
|---|---|---|---|---|---|---|---|---|
| | Fail% | Dur. | Fail% | Dur. | Fail% | Dur. | Fail% | Dur. |
| Pessimistic | 0.657 | 4.80±2.22 | 0.541 | 4.62±3.12 | 0.729 | 5.32±1.46 | 0.724 | 5.40±1.76 |
| Mean | 0.403 | 5.97±1.69 | 0.428 | 5.09±1.80 | 0.272 | 6.49±1.12 | 0.272 | 6.54±1.19 |
| Optimistic | 0.325 | 6.30±1.36 | 0.386 | 5.18±1.65 | 0.099 | 6.90±0.451 | 0.100 | 6.90±0.452 |

For problems of this size RTDP can enumerate the state space in memory, giving it a significant advantage because it can compute the optimal global policy. Thus, we do not expect to be able to generally perform better RTDP on this problem. The fact that we outperform RTDP at all is due to the fact that we use the labelled variant of RTDP [Bonet and Geffner, 2003], with a non-zero labelling threshold that results in some degree of approximation in the policy. However, as Aberdeen [2005] demonstrates, when problems are too large to fit into main memory, the FPG-Planner can perform significantly better than RTDP based planners.

## 6 Discussion

The main requirement for learning with policy-gradient POMDP methods is that a trajectory of state observations is available. Even if we have *no model* of the planning problem we can still use FPG-Planning provided we can interact with the real-world to generate trajectories.

The greatest drawback of our work is the assumption that the poorest (or best) global model can be approximated by always trying to simulate the extremes of the intervals in each state. We can avoid this assumption by simultaneously learning the worst model at the same time as learning the best policy. This can be achieved with a second agent assigned to each planning task. The second agent learns, again using a gradient method, the most pessimistic point in the interval that should be used to simulate trajectories. We plan to try this approach soon, borrowing on the work of Bowling [2005].

To summarise, we have demonstrated an algorithm with great potential to produce policies that are robust to a degree of 'guesswork' in constructing the model. It is critical that real-world planning tools are tolerant of errors in the description of the model. Human beings are bad at estimating probabilities, and it is rare that we have sufficient data to perfectly estimate all parameters of a system. Further work will attempt to justify our claim that the simulation approach to dealing with uncertainty has merit in very large domains.

### Acknowledgements

National ICT Australia is funded by the Australian Governmnent's Backing Australia's Ability program and the Centre of Excellence program. This project was also funded by the Australian Defence Science and Technology Organisation.

## References

[Aberdeen *et al.*, 2004] D. Aberdeen, S. Thiébaux, and L. Zhang. Decision-theoretic military operations planning. In *Proc. ICAPS'04*, 2004.

[Aberdeen, 2005] D. Aberdeen. Probabilistic temporal planning by factored policy gradient. Technical report, NICTA, 2005.

[Barto *et al.*, 1995] A.G. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72, 1995.

[Baxter *et al.*, 2001] J. Baxter, P. Bartlett, and L. Weaver. Experiments with infinite-horizon, policy-gradient estimation. *JAIR*, 15:351–381, 2001.

[Bonet and Geffner, 2003] Blai Bonet and Hector Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proceedings of ICAPS-03*, 2003.

[Bowling, 2005] Michael Bowling. Convergence and no-regret in multiagent learning. In *Proc. of NIPS'04*, volume 17, 2005.

[Buffet and Aberdeen, 2005] O. Buffet and D. Aberdeen. Planning with robust (l)rtdp. In *Proc. of IJCAI'05*, 2005.

[Givan *et al.*, 2000] R. Givan, S. Leach, and T. Dean. Bounded parameter markov decision processes. *Artificial Intelligence*, 122(1-2):71–109, 2000.

[Hosaka *et al.*, 2001] M. Hosaka, M. Horiguchi, and M. Kurano. Controlled markov set-chains under average criteria. *Applied Mathematics and Computation*, 120(1-3):195–209, 2001.

[Little, 2004] I. Little. Probabilistic temporal planning. Honours thesis, Australian National University, 2004.

[Mausam and Weld, 2005] Mausam and Daniel S. Weld. Concurrent probabilistic temporal planning. In *Proc. ICAPS'05*, 2005.

[Peshkin *et al.*, 2000] L. Peshkin, N. Meuleau K.-E. Kim, and L. P. Kaelbling. Learning to cooperate via policy search. In *UAI*, 2000.

[Strehl and Littman, 2004] A. Strehl and M. Littman. An empirical evaluation of interval estimation for markov decision processes. In *Proc. of ICTAI'04*, 2004.

[Younes and Simmons, 2004] Hakan L. S. Younes and Reid G. Simmons. Policy generation for continuous-time stochastic domains with concurrency. In *Proc. of ICAPS'04*, volume 14, 2004.

# Dynamic Domains in Data Production Planning

**Keith Golden** **Wanlin Pang**[*]
NASA Ames Research Center
Moffett Field, CA 94035
{kgolden, wpang}@email.arc.nasa.gov

## Abstract

This paper discusses a planner-based approach to automating data production tasks, such as producing fire forecasts from satellite imagery and weather station data. Since the set of available data products is large, dynamic and mostly unknown, planning techniques developed for closed worlds are unsuitable. We discuss a number of techniques we have developed to cope with data production domains, including a novel constraint propagation algorithm based on planning graphs and a constraint-based approach to interleaved planning, sensing and execution.

## 1 Introduction

Petabytes of remote sensing data are now available from Earth-observing satellites to help measure, understand and forecast changes in the Earth system, but using these data effectively can be surprisingly hard. The volume and variety of data files and formats are daunting. Simple data management activities, such as locating and transferring files, changing file formats, gridding point data, and scaling and reprojecting gridded data, can consume far more personnel time and resources than the actual data analysis. We addressed this problem by developing a planner-based agent for data production, called IMAGEbot [Golden *et al.*, 2003], that takes data product requests as high-level goals and executes the commands needed to produce the requested data products.

The data production problem consists of converting an initial set of low-level data products into higher-level data products that can be used for science or decision support. The data products we are concerned with are geospatial data measuring specific *variables* of the Earth system, such as precipitation, vegetation productivity and fire risk, but our approach is also applicable to other types of data. Higher-level data products may be altered versions lower-level data products, or they may be entirely new products that estimate unknown Earth system variables, such as soil moisture, based on known variables, such as precipitation. These variables are estimated by running one or more computational *models*, such as simulation codes. The models can be precisely characterized in terms of their input and output requirements, which makes them straightforward to represent in an AI planning system. However, there are significant differences between the data production problem and more traditional planning domains, calling for different techniques.

Notable features of data processing domains include large dynamic universes, incomplete information and uncertainty. There are petabytes of data available, with new data becoming available all the time, and the agent itself produces many new data products in the course of fulfilling the user's goal—data products that could be used to fulfill subsequent goals. There is also considerable uncertainty — uncertainty of the time that particular data will be available, or whether the data will arrive at all, uncertainty in the quality of data, even uncertainty as to whether a given processing algorithm will succeed. To cope with this uncertainty, the agent may need to poll for data availability or try alternative courses of action if the one it is pursuing seems unpromising.

We have developed a planner-based agent, called IMAGEbot, to automate data production. The data production problem may be viewed as a planning problem in which the initial state describes the current set of available data products, and whose goal state describes the properties of the desired high-level data products. Planner operators correspond to data transformation and generation tools. IMAGEbot takes data product requests as high-level goals and executes the commands needed to produce the requested data products.

We adopt a planning approach somewhat similar to Graphplan [Blum & Furst, 1997], consisting of a Graphplan-style reachability analysis and a constraint-based search. However, the large universe of the data production problem makes the grounded planning graph of Graphplan inapplicable; instead, we choose a lifted representation where actions and plans contain variables. Because of the lifted representation, and the uncertain and dynamic nature of the data production problem, the reachability analysis and search cannot be separated; instead, IMAGEbot interleaves planning, constraint reasoning and execution.

In this paper, we report on our work on IMAGEbot, with a focus on the constraint reasoning that underlies planning, sensing and execution. Section 2 gives an overview of the IMAGEbot system architecture and high-level planning approach; Section 3 discusses our constraint-based approach to sensing; Section 4 discusses a novel constraint propagation

[*] QSS Group Inc.

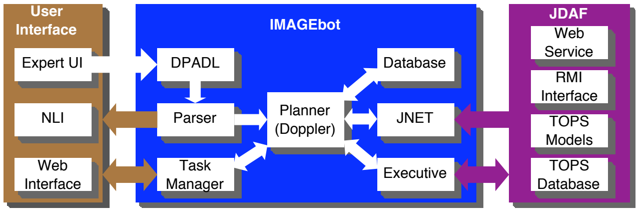


Figure 1: The architecture of IMAGEbot

algorithm based on the planning graph. Section 5 discusses interleaved planning and execution.

# 2 IMAGEbot Overview

## 2.1 System Architecture

The architecture of the IMAGEbot agent is depicted in Figure 1. The main components are:

**JDAF:** The **J**ava **D**istributed **A**pplication **F**ramework comprises the execution environment for IMAGEbot; it provides the agent with a common API for data-processing programs and ecological forecasting models.

**DPADL:** The **D**ata **P**rocessing **A**ction **D**escription **L**anguage [Golden, 2002] is used to provide action descriptions of data-processing programs and available data sources. Goals, in the forms of data product requests, can also be described in DPADL. To support both fine-grained and flexible sensing, DPADL allows constraints to make calls to the underlying runtime environment (Section 3).

**DoPPLER:** The **D**ata **P**rocessing **Planner** accepts goals in the form of data descriptions and synthesizes and executes data-flow programs. It reduces the planning problem to a Constraint Satisfaction Problem (CSP) whose solution provides a solution to the original planning problem.

**JNET:** **J**ava Constraint **Net**work is a constraint representation and reasoning framework that provides the agent with constraint propagation and search capabilities.

The architecture provides a planning framework that interleaves planning with constraint reasoning and plan execution.

## 2.2 Planning Approach

Planning in IMAGEbot is a two-stage process. The first stage consists of a Graphplan-style reachability analysis [Blum & Furst, 1997] to derive heuristic distance estimates and restrict the search space for the second stage, a constraint-based search. These stages are not entirely separate, however; constraint propagation occurs even in the the graph-construction stage, and the graph is refined during the constraint-search phase.

### Lifted planning graphs

Planning domains are specified in DPADL. From the planning problem specification, the planner incrementally constructs a directed graph, similar to a planning graph. Planning graphs were introduced in Graphplan [Blum & Furst, 1997], and have been adopted widely as an efficient data structure for computing reachability heuristics for planning problems. A planning graph consists of alternating layers of proposition nodes and action nodes. After the first proposition layer, which contains all propositions that are true in the initial state, each action layer comprises all actions whose preconditions are supported by the previous proposition layer and each proposition layer comprises all propositions that appear in the effects of the previous action layer. Pairwise mutual exclusion constraints (mutexes) are used to indicate when two actions or propositions cannot possibly appear at the same time. For example, $P$ and $\neg P$ are mutex, as are actions that interfere with each other. The planning graph provides a compact, abstract representation of all possible plans up to a given length. In the initial construction phase, it is built up until *quiescence*, i.e., until there are no changes from one layer to the next, at which point the planner can search backward from the last layer for a plan that satisfies the goal. If no plan is found, the graph is extended by adding more layers and the search is repeated.

A disadvantage of the planning graph, for our purposes, is that it is fully grounded. Since the number of possible objects (and hence the number of propositions and actions) is infinite, we use a lifted representation (*i.e.*, containing variables). We also replace proposition nodes with *object* nodes, which correspond to the data products produced and consumed by actions. Instead of binary mutexes, we rely on a richer set of constraints that specify the inputs, outputs and parameters of actions in terms of those of other actions. This graph is used to obtain distance estimates for heuristic search, obtain bounds on the possible values of variables, and is also the basis for the construction of the CSP. Arcs in the graph are analogous to causal links [Penberthy & Weld, 1992]. A causal link is a triple $\langle \alpha_s, p, \alpha_p \rangle$, recording the decision to use action $\alpha_s$ to support precondition $p$ of action $\alpha_p$. However, instead of recording a commitment of support, it indicates the *possibility* that $\alpha_s$ supports $p$. The lifted graph contains multiple ways of supporting $p$; the choice of the actual supporter becomes a constraint satisfaction problem. We add an extra term to the arc for bookkeeping purposes — the condition, $\gamma_p^{\alpha_s}$, needed in order for $\alpha_s$ to achieve $p$. A link then becomes $\left\langle \alpha_s, \gamma_p^{\alpha_s}, p, \alpha_p \right\rangle$.

Given an unsupported precondition $p$ of action $\alpha_p$, our first task is to identify all the actions that could support $p$. Because the universe is large and dynamic, identifying all possible ground actions that could support $p$ would be impractical, so instead we use a lifted representation, identifying all action *schemas* that could provide support. Given an action schema $\alpha$, we determine whether it supports $p$ by *regressing* $p$ through $\alpha$. The result of regression is the formula $\gamma_p^{\alpha_s}$. If $\gamma_p^{\alpha_s} = \perp$, then $\alpha$ does not support $p$. Initial graph construction terminates when all preconditions have support or (more likely) a potential loop is detected.

### From planning to constraints

After the graph is constructed, heuristic distance estimates for guiding the search are computed, and a constraint problem representing the search space is incrementally built. It is incremental because the planning graph comprises a compact representation of the search space, in which each action node can represent multiple concrete actions in the final plan.

Since the number of possible actions can be large, even infinite, we cannot simply generate all of them at once but do so lazily during search. This is handled using a dynamic CSP (DCSP), in which new variables and constraints can be added for each new action and causal link in the plan.

A **Constraint Satisfaction Problem (CSP)** is a representation and reasoning framework consisting of variables, domains, and constraints. Formally, it can be defined as a structure $< X, D, C >$ where $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, $D = \{d(x_1), d(x_2), \ldots, d(x_n)\}$ is a set of domains containing values the variables may take, and $C = \{C_1, C_2, \ldots, C_m\}$ is a set of constraints. Each constraint $C_i$ is defined as a relation $R$ on a subset of variables $V = \{x_i, x_j, \ldots, x_k\}$, called the constraint scope. $R$ may be represented extensionally as a subset of cartisan product $d(x_i) \times d(x_j) \times \ldots \times d(x_k)$. A constraint $C_i = (V_i, R_i)$ limits the values the variables in $V$ can take simultaneously to those assignments that satisfy $R$. A consistent instantiation of all variables in $X$ is a *solution*. The central reasoning task (or the task of solving a CSP) is to find one or more solutions.

The CSP contains: 1) boolean variables for all arcs, nodes and conditions; 2) variables (of any type) for all parameters, input and output variables and function values; 3) for every condition in the graph, a constraint specifying when that condition holds (for conditions supported by arcs, this is just the XOR of the arc variables); 4) for conjunctive and disjunctive expressions, the constraint is the respective conjunction or disjunction of the boolean variables corresponding to appropriate sub-expressions; 5) for every arc in the graph, constraints specifying the conditions under which the supported fluents will be achieved (i.e., $\gamma_p^\alpha \Rightarrow p$, where $\gamma_p^\alpha$ is the precondition of $\alpha$ needed to achieve $p$) ; 6) user-specified constraints; and 7) constraints representing structured objects.

### Constraint-based search

After converting the planning problem to a CSP, the planner searches the CSP for a solution. At a high level, the planner, guided by heuristic distance estimates extracted from the planning graph, selects subgoals to achieve and actions to achieve them (Algorithm 2). After the subgoal and action selection, the planner (or more accurately, the CSP solver) finds values for variables representing planner action parameters. This is necessary to make actions executable. During the search, propagation is performed whenever a value is assigned to a variable. The search is an iterative process involving possible backtracks; that is, if there are no valid parameters for a chosen action, the planner has to search for another plan; if it is impossible to extract a plan from the current planning graph, the planning graph is extended, by adding more layers, or search fails. Extending the planning graph and repeating the search corresponds to searching for longer plans.

## 3 Constraint-based sensing

In order to find out what data products relevant to the task at hand are available, the agent needs to sense its environment. One way of doing this is to introduce *sensing actions* [Golden & Weld, 1996], which the agent can execute in order to obtain information. This approach has the advantage that it can be used to capture sensing actions that have preconditions, but it also requires the plan to be at least partially executed before the information can be obtained. We follow an alternative approach of representing low-cost precondition-free sensors using *procedural constraints*. That is, we can implement constraints as procedures that can perform database queries or invoke other information-gathering operations in the course of identifying the domain of values a given variable can have. This constraint-based sensing approach is much more flexible than the sensing-action approach, as the order of sensing operations is based on constraint propagation, and information dependencies are inherently multi-directional.

For example, suppose we have a set of satellite images, each of which corresponds to a given region of the Earth's surface for a given day. Procedure calls are available to identify satellite images for a particular region and day and to identify specific attributes of individual images. To obtain the set all the images for a given region $\rho$ and day $d$, we can call findTiles($\rho, d$). To obtain the resolution of image $t$, we call $t$.getResolution(). To find the percent cloud cover for $t$, we call $t$.getCloudCover(). Other attributes of $t$ can be determined similarly. In order to make these procedures usable to the constraint solver, we must represent them as constraints. The DPADL language makes this easy, by allowing the code for the procedure calls to be embedded directly into constraint definitions [Golden, 2003]. The details of the syntax are unimportant. All that matters is that when the constraint solver encounters the constraint $r$ = resolution($t$), it can enforce it by calling the getResolution method on individual values from the domain of $t$, which are Java objects. The results are combined and intersected with the original domain of $r$. If the result of the intersection is non-empty, then the constraint is still valid. Similarly, findTiles($r, d$) can be called to restrict the domain of $t$.

What information-gathering procedures are called, and the order in which they are called, depends on the constraint propagation algorithm used by the constraint solver. In general, if the domain of any variable is reduced, all constraints involving that variable will be enforced, possibly causing the domains of other variables to be reduced as well, which will trigger other constraints to be enforced. Propagation stops when this process reaches quiescence or some variable domain becomes empty, which indicates an inconsistency. Constraint propagation provides a powerful and flexible way of sensing. The specific set of operations performed depends on what information is "known" to the constraint solver. Invoking a sensing operation may trigger further sensing through constraint propagation. For example, suppose we are interested in finding a high resolution satellite image of Oregon for a day in June that had no rainfall. We can represent that as a set of constraints:

```
resolution(i)≤250m, region(i)=Oregon,
day(i)=d, month(d)=June, year(d)=2005,
rainfall(d) = 0.
```

Initially, the domains of $i$ and $d$ are "full," *i.e.*, completely unknown, but enforcing the constraints on month and year restricts $d$ to a set of 30 possible values. The rainfall constraint can be evaluated for each value of $d$ by executing a procedure that does a database lookup on past meteorological data. Days that had any rainfall will be eliminated from the domain of $d$. The procedure findTiles$(r, d)$ can then be called with each of those dates for region "Oregon." to restrict the domain of $i$ to a finite set. Once the images are known, getResolution can be called on each one, and images with inadequate resolution will be eliminated from the domain of $i$. At this point, all values in the domain of $i$ satisfy all the constraints. As this example shows, the order of sensing operations depends on what information is "known" and what information is needed.

We can also represent more traditional sensing actions, using actions that produce new objects (data files), which contain information. Acquiring these objects can, in turn, trigger more constraint propagation, resulting in more implicit sensing. For example, a data-acquisition action may obtain a set of satellite images from a remote location. Once these images are available, additional operations can be performed to obtain information about the images, such as data quality. These additional operations can be implemented as constraints rather than actions, which removes them from the set of deliberate decisions that the planner needs to make.

# 4 Action-based Constraint Propagation

As we have discussed, data production problems, due to their large, uncertain and dynamic universes, are not suitable for a grounded representation. The lifted planning graph is a much more concise representation than the grounded planning graph, but it is potentially less informative, which makes conventional constraint propagation and search less effective. The CSP derived from the lifted planning graph contains variables with infinite domains [Golden & Frank, 2002], so there is no way to enumerate solutions by search alone, yet the traditional constraint propagation that establishes certain levels of consistency does not work well either. For example, we have a constraint propagator in JNET that enforces a partial[1] *generalized arc-consistency* (GAC) [Bessiere & Ch, 1997; Katsirelos & Bacchus, 2001]. The definition of GAC is built upon the variables and their values; namely, a CSP is GAC if all its variables are GAC; a variable is GAC is all its values are GAC; a value $v$ of a variable $x$ is GAC if it has support from other variables in every constraint on $x$. Establishing consistency requires evaluating every value to see if it satisfies certain constraints, which is not possible in general for infinite variable domains. A combination of propagation and search will eventually find a solution, but propagation does not become informative until late in the search .

We have developed a new constraint propagation algorithm that propagates changes among the actions in the planning graph, which yields much more information, even before search begins. It not only restricts the domains of variables by eliminating inconsistent values, but it also may add values to the variable domains when new information is available (e.g., a new object is created). In this section, we first describe the propagation algorithm, then illustrate how it works with an example, and discuss its role in the planning search and constraint search.

**Algorithm 1** Action Constraint Propagation

Given a lifted plan graph $G$. Let $A$ be the set of actions in $G$, let $P = (X, D, C)$ be the CSP derived from the lifted plan graph, and let $A'$ be a subset of actions to be propagated:

```
propagate($G, A, P, A'$)
  1. while ($A' \neq \emptyset$) do
     (a) let $a \leftarrow$ an action removed from $A'$
     (b) let $C_a \leftarrow$ constraints relevant to $a$
     (c) $< d(\mathcal{I}(a)), d(\mathcal{O}(a)) > \leftarrow$ enforce($P, C_a$)
     (d) for ($\forall i \in \mathcal{I}(a)$ s.t. $d(i) = \emptyset$)
         remove supporting link to $i$
     (e) for ($\forall o \in \mathcal{O}(a)$ s.t. $d(o) = \emptyset$)
         remove supporting link from $o$
     (f) for ($\forall i \in \mathcal{I}(a)$ s.t. $d(i)$ changed)
         i. for ($\forall b \in A$ s.t. $b$ supports $a$)
            if (revise($P, \mathcal{O}(b), i$)) $A' \leftarrow A' \cup \{b\}$
     (g) for ($\forall o \in \mathcal{O}(a)$ s.t. $d(o)$ changed)
         i. for ($\forall b \in A$ s.t. $a$ supports $b$)
            if (revise($P, \mathcal{I}(b), o$)) $A' \leftarrow A' \cup \{b\}$
  2. return
```

## 4.1 Algorithm

Formally, a data-processing action schema can be seen as a tuple $\langle \mathcal{I}, \mathcal{O}, \mathcal{P}, \Pi, \mathcal{E}, \chi \rangle$, where $\mathcal{I}$,$\mathcal{O}$,$\mathcal{P}$ are the *input* variables, *output* variables and *parameters* respectively. The parameters are unknowns that may appear in constraints on either or both input and output. $\Pi$ is the *precondition*, $\mathcal{E}$ is the *effect* and $\chi$ is a procedure for executing the action that may reference any variable in $\mathcal{I} \cup \mathcal{P}$ and must set every variable in $\mathcal{O}$. A lifted planning graph can be seen as a partially ordered set of action schemas $(A, \prec)$, where $a \prec b$ *iff* action $a$ supports $b$ or $a$ supports $c$ and $c \prec b$. In the CSP derived from the lifted planning graph, we have constraints specifying the relationships among variables inside an action and constraints specifying relationships of two actions if one supports another. For an individual action, if something changes — for example, if a value is assigned to a variable in the action input due to search — the change to this variable can be propagated to other variables in the output, which may change their domains. For two actions $a$ and $b$, where $a$ supports $b$, changes in the input of $b$ can be propagated to the output of $a$; similarly, changes in the output of $a$ can be propagated to the input of $b$. The idea of this propagation is outlined in Algorithm 1.

In Algorithm 1, function enforce$(P, C_a)$ enforces every constraint $c \in C_a$ associated with action $a$. It restricts domains of variables in $c$ by eliminating inconsistent values. Function revise$(P, \mathcal{O}(b), i)$ (or revise$(P, \mathcal{I}(b), o)$ ) computes the domains of variables in $\mathcal{O}(b)$ (or $\mathcal{I}(b)$ ), where $i$ is an input (or $o$ an output) of action $a$ and action $b$ supports (or is supported by) $a$. The function revise may remove inconsistent values or add newly discovered values depending on the planning graph structure. It returns true if any variable

[1] We call it partial GAC for two reasons: 1) not every constraint procedure enforces the GAC; and 2) not every constraint is executed in the propagation.

domain has been revised, in which case the action $b$ is added to $A'$, waiting to be propagated.

In addition to removing inconsistent values or discovering new values for variables in an action, this propagation also removes certain supporting links if it identifies inconsistency. If all links from an action $a$ supporting other actions are removed, the action $a$ is useless in the planning graph so it can be safely removed. If all links to an input of an action $a$ are removed, this action cannot be executed because one of its inputs does not have support. The planner either has to find other support for this action (e.g., expanding the planning graph by inserting more actions) or remove this action from the planning graph.

## 4.2 Example

For illustration, we consider a simplified version of constructing a *mosaic*. Many satellites continuously image whatever portion of the Earth they pass over, like giant hand-held scanners. For convenience, the resulting *swath* data is usually reprojected onto a 2D *map* and chopped up into *tiles*, corresponding to a regular grid drawn over the map. To obtain the data pertaining to a particular region of the Earth, we first identify and obtain the tiles that cover that region and then combine them into a single image, known as a mosaic, and crop away the pixels outside the region of interest.

These tiles are represented in the planner as first-class objects. The attributes of a tile describe, among other things, the physical measurement the data in the tile represent, the position of the tile on the grid, the projection used to flatten the globe, and the region of the Earth covered by the pixels in the image. For simplicity, we assume in this example that tiles have only two attributes: the *region* a tile covers and the *cloudiness* when the image was taken. A simplified task becomes to take some tiles from thousands of available tiles and compose them to create a mosaic that covers a specified region without too much cloud cover.

Specifically, a *region* is a pair of points $\langle ul, lr \rangle$ where $ul$ is the upper-left corner and $lr$ the lower-right corner. A point is a pair of coordinates $(x, y)$. Normally $x$ and $y$ would be longitude and latitude, but as a further simplification, we will assume both $x$ and $y$ are non-negative integers. The cloudiness is represented by a real number from 0 to 1, where 0 is clear sky and 1 is totally obscured. Further, we assume there are only three actions the planner may take: compose two tiles horizontally (*comp2h*) or vertically (*comp2v*), or get a tile with its *ul* point as a parameter (*getTile*). A real mosaic command is not limited to combining two tiles. Figure 2 shows action preconditions and effects with respect to the region. In addition, the effect of composing two images is that their combined cloudiness is treated as the maximum of the cloudiness of the input tiles.

A problem instance we consider here consists of some small tiles, such as those covering the region$\langle(0,0),(1,2)\rangle$, or $\langle(2,3),(3,5)\rangle$. The goal is to compose a mosaic for the region $\langle(0,0),(3,2)\rangle$ with no more than 30% cloud cover. As shown in Figure 4, the region $\langle(0,0),(3,2)\rangle$ consists of 6 unit squares denoted by $B_1, B_2, ..., B_6$. For example, $B_1$ refers to the region $\langle(0,0),(1,1)\rangle$, and $B_1B_2$ together refer to the region $\langle(0,0),(2,1)\rangle$. The mosaic is composed of tiles covering $B_1, B_2, ..., B_6$ (referring to the region $\langle(0,0),(3,2)\rangle$), which may or may not be available locally; if not, we assume that action *getTile*$((x, y))$ can be executed to get any available tiles covering the region $\langle(x, y),(x+m, y+n)\rangle$.

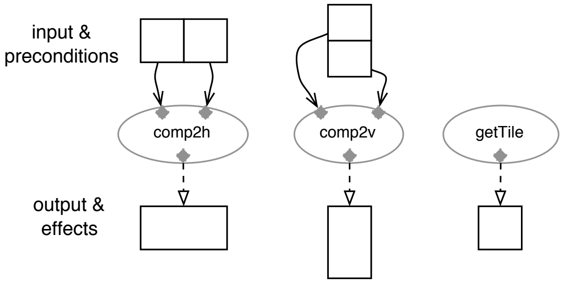


Figure 2: The planner actions: the dots inside actions are inputs and outputs. Parameters are not shown.

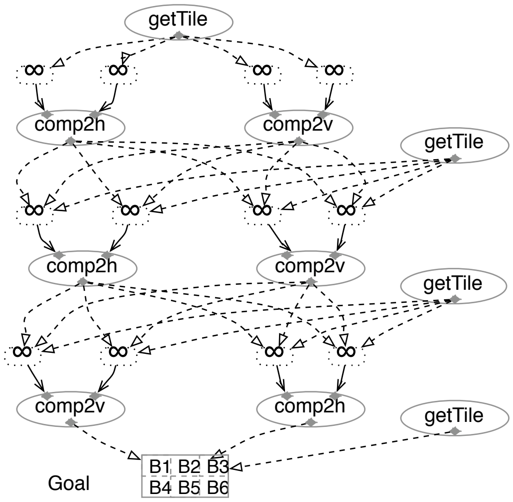


Figure 3: A planning graph

The planning graph created by the planner is shown in Figure 3, where nodes represent lifted actions or objects, and arcs the supporting relations. The dots inside action nodes are inputs and outputs of the actions, each representing a set of objects, possibly infinite. At the time when a CSP is derived from this planning graph, these unknown objects, inputs and outputs of the actions and their parameters, are represented as variables with infinite domains.

The action-based constraint propagation can be invoked to restrict some of the infinite domains. Since the planner goal, a mosaic of region $\langle(0,0),(3,2)\rangle$ with cloudiness no more than 0.30, is known, the output of action *comp2v,* which supports the goal, is also known; applying the propagation on *comp2v* from output to input, we have the domains of its two inputs, both of which are singletons, namely a mosaic covering the region$\{\langle(0,0),(3,1)\rangle\}$ and another one covering$\{\langle(0,1),(3,2)\rangle\}$, both with cloudiness of more than 0.30. Similarly, the output of *comp2h* is known from the goal; applying the propagation on *comp2h*, we have the

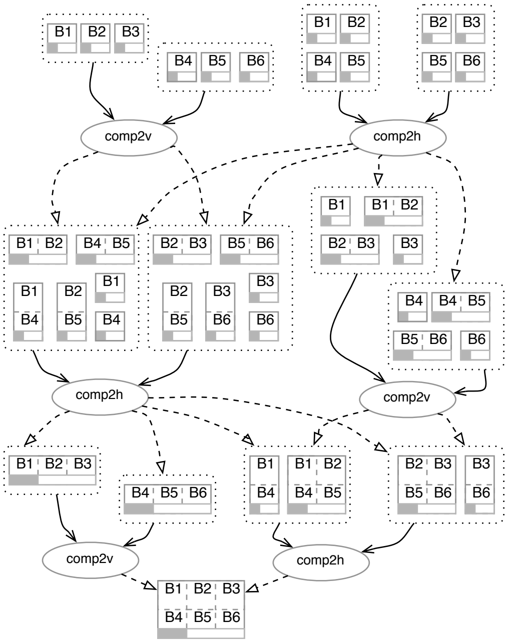


Figure 4: Constraint propagation in the planning graph. Objects in a dotted rectangles are inputs to an action, an object divided by dashed line is a single or a composed object; whether a single object is available in the initial state or can be obtained with *getTile* (not shown) depends on the task.

domains of its two inputs, both of which contain two mosaics covering the regions: $\{\langle(0,0),(1,2)\rangle, \langle(0,0),(2,2)\rangle\}$ and, $\{\langle(1,0),(3,2)\rangle, \langle(2,0),(3,2)\rangle\}$, respectively. Again, all these mosaics (some are possibly single tiles, depending on the sizes of tiles available) must have cloudiness of at most 0.30. The changes to inputs of the these actions are propagated to the prior level actions supporting them. This process continues until there are no more changes to the input and output of any actions. At the end of the backward propagation, we have a much more limited search space, as shown in Figure 4, where the tiles in the inputs and outputs are restricted to specified regions and cloudiness. Notice also that many links appearing in the planning graph (see Figure 3) have been removed by the propagation. For example, all links from *comp2v* to *comp2v* have been removed.

At this point, none of the actual tiles are available, since the *getTile* actions have yet to be executed, so the domains of the variables representing cloudiness are all $[0 \dots 0.30]$, based on propagation from the goal. No further propagation or pruning can be done. We continue with this example in the next section.

## 5 Planning and Execution

**Algorithm 2** Plan construction and execution. Iteratively supports subgoals and executes actions until all goals are supported and all actions are executed. The keyword **pick** indicates a choice that is not a backtrack point. The keyword **choose** indicates nondeterministic choice (backtrack point) The keyword **fail** indicates a backtrack.

```
public void PlanAndExecute(goal, actions)
1. let $G \leftarrow$ BuildPlanGraph(goal, actions)
2. let $P \leftarrow$BuildConstraintNet($G$), $A \leftarrow$ Actions in $G$
3. let agenda $\leftarrow$ {goal}, unexecuted $\leftarrow$ {goal}
4. set $d$(goal) $\leftarrow$ {true}
5. while (propagate($G, A, P, A$) returns false)
   if (ExpandGraph($G, P$) returns false) fail
6. while (unexecuted $\neq \emptyset$) pick
   (a) pick $\alpha \in$unexecuted
       if (execute($\alpha$) returns true)
       remove $\alpha$ from unexecuted
   (b) let $p \leftarrow$ remove from agenda
       i. choose $\langle \alpha_s, \gamma_p^{\alpha_s}, p, \alpha_p \rangle$ in $G$
       ii. add $\gamma_p^{\alpha_s}$ to agenda and set $d(\gamma_p^{\alpha_s}) = \{true\}$
       iii. add $\alpha_s$ to unexecuted
       iv. if (propagate($G, A, P, \{\alpha_s, \alpha_p\}$) returns false)
           fail
   (c) ExpandGraph($G, P$)
```

Although our constraint-based approach to sensing helps to cope with large, unknown domains, there is still some uncertainty, even for a “complete” plan. Data products may turn out to be of a lesser quality than expected, due to cloud cover for instance, or may even turn out to be missing entirely. Processing algorithms may fail to perform as well as expected, perhaps due to problems with the input data, or they may simply crash. Some quality problems can be automatically detected, but only after the data products are in hand, meaning after the plan has been at least partially executed. Fortunately, the non-destructive nature of data production domains means the cost of plan execution is limited to the time and resources consumed, so it is natural to view plan execution as an extension of the search process. If partial execution of a plan reveals a violation of a constraint or preference, it is a simple matter to backtrack and try something else, since there are no state changes to be undone. Furthermore, actions may be executed before the plan is complete, yielding information to reduce search or choose between competing options. For example, if there are two candidate data sets, each of unknown quality and each of which requires different processing steps, the planner can execute the actions to obtain both sets of data and decide which one to use before wasting time planning out all the processing operations for data that may not be used.

Here, again, the planning graph representation is useful, because it provides a guide to which data sources and actions are relevant to a problem without requiring a complete plan to be generated. Once an action has been executed and its outputs produced, the output variables are instantiated with the results from execution and the constraints are re-propagated, which may further restrict the domains of other variables, reducing the amount of search.

This approach to interleaving planning, sensing and execution can be contrasted with contingency planning [Warren, 1976; Pryor & Collins, 1996; Draper, Hanks, & Weld, 1994], in which explicit branches are inserted into the plan after sensing actions in order to respond appropriately to the information obtained from sensing without the need for replanning; *e.g.*, "look-outside; if (raining), bring-umbrella." By interleaving planning with execution, we remove the need for explicit branches. The planner deals with the contingency after executing the sensing action. Suppose *look-outside* is executed, and the result is that it is raining. If the planner did not already commit to whether to select the *bring-umbrella* action, that choice is now forced by constraint propagation, and *bring-umbrella* becomes part of the plan. On the other hand, if the choice was already made not to bring the umbrella, then after constraint propagation the plan will be inconsistent; the planner will then backtrack and consider a plan that involves bringing the umbrella. Because there are no destructive actions in a data-production domain, there is never any harm from (re)planning for contingencies when they arise, other than wasted time and resources from executing unnecessary actions. On the other hand, as noted above, there is also no harm in planning *and executing* sub-plans both contingencies, which is essentially *conformant* planning [Smith & Weld, 1998].

Backtracking over execution cannot generate looping behavior. First, unlike [Golden, 1998], it never requires corrective actions to restore the world to a previous state because all state change is monotonic, creating new objects but never changing or destroying old ones. Second, backtracking occurs in the context of an overall search algorithm that is systematic.

## 5.1 Example

In Section 3, we discussed sensing constraints that can perform database queries or invoke other information-gathering operations to obtain available tiles. Sensing constraints, like other types of constraints, are invoked during the propagation. It is convenient to use constraints to represent low-cost, precondition-free sensors. However, we use actions to represent sensors that are costly or require some setup; this allows the planner to reason about the cost of sensing or consider alternative ways of achieving the preconditions for sensing. In the previous example, we used a constraint to model the sensor that informs the agent of the availability of tiles, since that availability test can be performed using a simple database query. However, obtaining and inspecting the actual files may require costly file transfers, so we represent that as an action, *getTile* (not shown in the figures).

To continue our previous example, suppose that we execute all the *getTile* actions in the planning graph before doing any explicit search, as shown in Figure 5. Some of the tiles believed to be available based on the database lookup are not delivered, due to network problems. These tiles, as well as the actions that depend on them, are eliminated through constraint propagation. Additionally, now that the files are available, the actual cloudiness values (black bars) are determined, and these values are intersected with the prior domain values, which come from the goal requirement. The tile spanning $B_1$ and $B_2$ has cloudiness of $0.60$, and all the others have cloudiness values less than $0.30$. Since we know, from the goal and from the previous backward propagation, that any tile with cloudiness of more than $0.30$ is useless in this task, the tile covering $B_1 B_2$ will be removed from the available tiles. As a result of further propagation, other tiles in the input and output of actions depending on that tile will be removed as well (crossed out by thick gray line). The final search space after the propagation is shown in Figure 6.

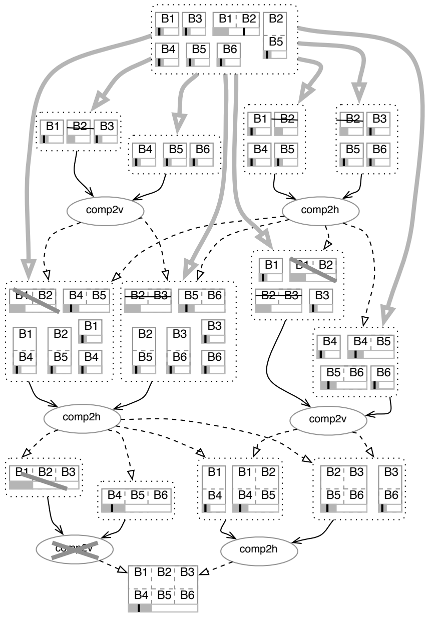


Figure 5: After execution, some tiles that were expected were not returned; constraint propagation in the planning graph eliminates these (crossed out). The actual cloud cover for the remaining tiles is determined (black bars; the original cloud cover domains, based on the goal, are shown in grey).

# 6 Conclusions

IMAGEbot is implemented and has been integrated into an ecological forecasting application [Golden *et al.*, 2003], which produces "nowcasts" and forecasts of socioeconomic importance, such as crop health and fire risk. New contributions presented in this paper include the algorithm for action-based constraint propagation and the constraint-based algorithm for interleaving planning, sensing and execution.

The idea of interleaving planning with execution as a way of coping with uncertainty is not new. Our approach is loosely based on [Golden, Etzioni, & Weld, 1994; Golden, 1998], which in turn was inspired by [Ambros-Ingerson & Steel, 1988]. However, interleaving planning and execution is much more natural in monotonic problem domains like data production.

We believe the constraint-based sensing and planning-graph propagation approaches introduced in this paper would be equally suitable to other software domains that involve large, unknown dynamic domains. Related applications to which planners have been applied include Internet softbots

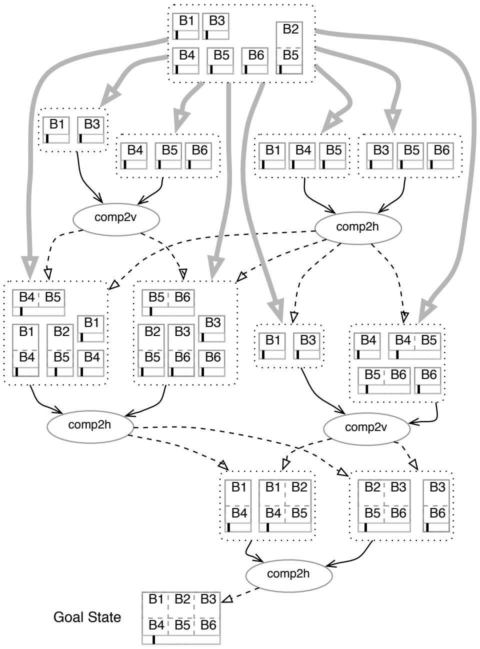


Figure 6: The planning graph after execution and propagation. The intervals for cloud cover have been replaced with singleton values.

[Golden, 1998; Etzioni, Golden, & Weld, 1997], web services [Srivastava & Kholer, 2003], image processing [Lansky, 1998; Chien *et al.*, 1997], and grid-based computing [Blythe *et al.*, 2003].

## References

[Ambros-Ingerson & Steel, 1988] Ambros-Ingerson, J., and Steel, S. 1988. Integrating planning, execution, and monitoring. In *Proc. 7th Nat. Conf. AI*, 735–740.

[Bessiere & Ch, 1997] Bessiere, C., and Ch, J. 1997. Arc-consistency for general constraint networks: Preliminary results. In *Proceedings of IJCAI-97*, 398–404.

[Blum & Furst, 1997] Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *J. Artificial Intelligence* 90(1–2):281–300.

[Blythe *et al.*, 2003] Blythe, J.; Deelman, E.; Gil, Y.; Kesselman, C.; Agarwal, A.; Mehta, G.; and Vahi, K. 2003. The role of planning in grid computing. In *Proc. 13th Intl. Conf. on Automated Planning and Scheduling (ICAPS)*.

[Chien *et al.*, 1997] Chien, S.; Fisher, F.; Lo, E.; Mortensen, H.; and Greeley, R. 1997. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining*.

[Draper, Hanks, & Weld, 1994] Draper, D.; Hanks, S.; and Weld, D. 1994. A probabilistic model of action for least-commitment planning with information gathering. In *Proc. 10th Conf. Uncertainty in Artifical Intelligence*.

[Etzioni, Golden, & Weld, 1997] Etzioni, O.; Golden, K.; and Weld, D. 1997. Sound and efficient closed-world reasoning for planning. *J. Artificial Intelligence* 89(1–2):113–148.

[Golden & Frank, 2002] Golden, K., and Frank, J. 2002. Universal quantification in a constraint-based planner. In *Proc. 6th Intl. Conf. Automated Planning Systems*.

[Golden & Weld, 1996] Golden, K., and Weld, D. 1996. Representing sensing actions: The middle ground revisited. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning*, 174–185.

[Golden *et al.*, 2003] Golden, K.; Pang, W.; Nemani, R.; and Votava, P. 2003. Automating the processing of earth observation data. In *International Symposium on Artificial Intelligence, Robotics and Automation for Space*.

[Golden, Etzioni, & Weld, 1994] Golden, K.; Etzioni, O.; and Weld, D. 1994. Omnipotence without omniscience: Sensor management in planning. In *Proc. 12th Nat. Conf. AI*, 1048–1054.

[Golden, 1998] Golden, K. 1998. Leap before you look: Information gathering in the PUCCINI planner. In *Proc. 4th Intl. Conf. AI Planning Systems*.

[Golden, 2002] Golden, K. 2002. DPADL: An action language for data processing domains. In *Proceedings of the 3rd NASA Intl. Planning and Scheduling workshop*, 28–33. to appear.

[Golden, 2003] Golden, K. 2003. An domain description language data processing. In *ICAPS 2003 Workshop on the Future of PDDL*.

[Katsirelos & Bacchus, 2001] Katsirelos, G., and Bacchus, F. 2001. GAC on conjunctions of constraints. In *Proceedings of CP-2001*.

[Lansky, 1998] Lansky, A. 1998. Localized planning with action-based constraints. *Artificial Intelligence* 98(1–2):49–136.

[Penberthy & Weld, 1992] Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. Principles of Knowledge Representation and Reasoning*, 103–114.

[Pryor & Collins, 1996] Pryor, L., and Collins, G. 1996. Planning for contingencies: A decision-based approach. *J. Artificial Intelligence Research*.

[Smith & Weld, 1998] Smith, D., and Weld, D. 1998. Conformant graphplan. In *Proc. 15th Nat. Conf. AI*.

[Srivastava & Kholer, 2003] Srivastava, B., and Kholer, J. 2003. Web service composition - current solutions and open problems. In *ICAPS 2003 Workshop on Planning for Web Services*. available at http://www.isi.edu/info-agents/workshops/icaps2003-p4ws/program.html.

[Warren, 1976] Warren, D. 1976. Generating Conditional Plans and Programs. In *Proceedings of AISB Summer Conference*, 344–354.

# Hedged learning: Regret-minimization with learning experts

**Yu-Han Chang, Leslie Pack Kaelbling**
CSAIL, Massachusetts Institute of Technology
32 Vassar Street, Cambridge, MA 02139 USA
{ychang, lpk}@csail.mit.edu

## Abstract

In non-cooperative multi-agent situations, there cannot exist a globally optimal, yet opponent-independent learning algorithm. Regret-minimization over a set of strategies optimized for potential opponent models is proposed as a good framework for deciding how to behave in such situations. Using longer playing horizons and experts that learn as they play, the regret-minimization framework can be extended to overcome several shortcomings of earlier approaches to the problem of multi-agent learning.

## 1 Introduction

In recent years, there has been increasing interest in multi-agent learning. A large body of this work tries to marry game theoretic concepts such as Nash equilibrium to learning in various types of games. However, as Reinhard Selton, 1995 Economics Nobel Prize winner (along with Nash and Harsanyi) once wrote in a personal communication, “Game theory is for proving theorems, not for playing games.”

What does this mean for AI researchers interested in designing algorithms that learn to play good strategies in multi-agent domains? There are three issues related to applying equilibrium results from game theory directly: computational efficiency, learning dynamics, and opponent assumptions. In some cases, straight computation of Nash or correlated Nash equilibria in a given game is quite useful, and there have been a number of recent advances exploiting game structure to compute such equilibria efficiently. Moreover, some of these algorithms use learning dynamics to converge to correlated equilibria, thus addressing the core of Selton’s complaint: while the classic Nash equilibrium is a stable point, it is not necessarily the stable point of reasonable system dynamics.

Although these advances begin to resolve the issues of computational efficiency and learning dynamics, the problem of opponent assumptions remains more elusive. When we face an unknown opponent, we have no guarantee that the opponent will be playing strategically (as in classical Nash results), following any particular learning rule (as in the more recent work on correlated equilibria), or even playing vaguely intelligently. Perhaps the opponent has broken sensors or actuators, or lacks some crucial information. If we try to play our half of an equilibrium strategy, we may end up worse off if the opponent does not play its half of the strategy. To counter this problem, we would like to be able to model as many different types of potential opponents as possible. When one of our opponent models is correct, we would like to be performing optimally with respect to that model. If none of our models is correct, we would still like to avoid performing too poorly. The framework of regret minimizing, or hedging, algorithms provides a useful setup for approaching this problem. Using this framework, we can simultaneously achieve both of these goals: perform optimally if one of our models is correct, and still perform reasonably well if none of our models are correct. Since we know that it is impossible to design an algorithm that performs optimally with respect to all possible opponents [Nachbar & Zame, 1996], this is the best we can hope to do in a non-cooperative setting.

$$r_1 = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \qquad r_1 = \begin{bmatrix} 1 & -1 \\ 2 & 0 \end{bmatrix}$$

$$r_2 = -r_1 \qquad r_2 = \begin{bmatrix} 1 & 2 \\ -1 & 0 \end{bmatrix}$$

(a) Matching pennies (b) Prisoner’s Dilemna

Figure 1: Common examples of matrix games.

## 2 Mathematical setup

Repeated games, such as those in Figure 1, form the simplest framework for studying multi-agent learning. We will focus on this setup, although most of our ideas can be extended to stochastic games as well. We will assume that the reader has some familiarity with classical equilibrium concepts, such as Nash equilibrium of the one-shot game.

Modern game theory often takes a more general view of optimality in repeated games, considering actions that are defined as strategies over time, rather than only as a probability distribution over actions in a single instance of the matrix game. The machine learning community has also recently begun adopting this view [Chang & Kaelbling, 2001] [de Farias

& Meggido, 2004]. Rather than treating policies as a single probability distribution, we now define policies $\mu_i : H \rightarrow A_i$, where $H = \bigcup_t H^t$ and $H^t$ is the set of all possible histories of length $t$. Histories are observations of joint actions, $h^t = (a_i, a_{-i}, h^{t-1})$. Player $i$'s strategy at time $t$ is then expressed as $\mu_i(h^{t-1})$.

**Definition 1** *A $\tau$-length behavioral strategy $\mu^\tau$ is a mapping from all possible histories $H^\tau$ to actions $a \in A$. Let $M^\tau$ be the set of all possible $\tau$-length behavioral strategies $\mu^\tau$.*

We note that $|M^\tau| = |A|^{|A|^{2\tau}}$. In the case where we take $H^t = H$, we could even consider learning algorithms themselves to be a possible "behavioral strategy" for playing a repeated game. This definition of our strategy space is clearly more powerful, and allows us to define a much larger set of potential equilibria, where players are following a stable pair of behavioral strategies and have no incentive to deviate. However, when the opponent is not rational, it is no longer advantageous to find and play an equilibrium strategy. In fact, given an arbitrary opponent, the Nash equilibrium strategy may return a lower payoff than some other action. Indeed, the payoff may be even worse than the original Nash equilibrium value. Thus, we turn to regret minimization algorithms.

### 2.1 Regret-minimization

In repeated games, the standard regret minimization framework enables us to perform almost as well as the best action, if that single best action were played in every time period. Suppose we are playing using some regret-minimizing algorithm which outputs action choices $a_t \in A$ at each time period. Then our reward over $T$ time periods is $R(T) = \sum_{t=1}^{T} r_{a_t}(t)$.

**Definition 2** *Our regret is defined to be $R_{\max}(T) - R(T)$, where $R_{\max}(T) = \max_{a \in A} \sum_{t=1}^{T} r_a(t)$. If our algorithm randomizes over possible action choices, we also define expected regret to be $R_{\max}(T) - E[R(T)]$. The set of actions against which we compare our performance is called the comparison class.*

Both game theorists and online learning researchers have studied this framework [Fudenburg & Levine, 1995] [Freund & Schapire, 1999]. We will refer frequently to the EXP3 algorithm (and its variants) explored by Auer et al. (1995). In the original formulation of EXP3, we choose single actions to play, but we do not get to observe the rewards we would have received if we had chosen different actions. The authors show that the performance of EXP3 exhibits a regret bound of $2\sqrt{e-1}\sqrt{TN \ln N}$. Generally speaking, these regret-minimizing algorithms hedge between possible actions by keeping a weight for each action that is updated according to the action's historical performance. The probability of playing an action is then its fraction of the total weights mixed with the uniform distribution. Intuitively, better experts perform better, get assigned higher weight, and are played more often. Sometimes these algorithms are called experts algorithms, since we can think of the actions as being recommended by a set of experts.

It is important to note that most of these existing methods only compare our performance against strategies that are best responses to what are often called *oblivious* or *myopic* opponents. That is, the opponent does not learn or react to our actions, and essentially plays a fixed string of actions. Our best response would be to play the single best-response action to the empirical distribution of the opponent's actions. Under most circumstances, however, we might expect an intelligent opponent to change their strategy as they observe our own sequence of plays.

For example, consider the game of repeated Prisoner's Dilemma. If we follow the oblivious opponent assumption, then the best choice of action would always be to "Defect." Given any fixed opponent action, the best response would always be to defect. This approach would thus miss out on the chance to earn higher rewards by cooperating with opponents such as a "Tit-for-Tat" opponent, which cooperates with us as long as we also cooperate. These opponents can be called *reactive* opponents.

## 3 Extending the experts framework

Our extensions to the regret-minimization framework follow along the lines of the super-game setup proposed by Mannor and Shimkin (2001). Instead of choosing actions from $A$, we choose behavioral strategies from $M^\tau$. $M^\tau$ also replaces $A$ as our comparison class, essentially forcing us to compare our performance against more complex and possibly better performing strategies. While executing $\mu^\tau \in M^\tau$ for some number of time periods $\lambda$, the agent receives reward at each time step, but does not observe the rewards he would have received had he played any of his other possible strategies. This is reasonable since the opponent may adapt differently as a particular strategy is played, causing a different cumulative outcome over $\lambda$ time periods. Thus, the opponent could be an arbitrary black-box opponent or perhaps a fixed finite automaton. While the inner workings of the opponent are unobservable, we will assume the agent is able to observe the action that the opponent actually plays at each time period.

For example, we might consider an opponent whose action choices only depend on the previous $\tau$-length history of joint actions. Thus, we can construct a Markov model of our opponent using the set of all possible $\tau$-length histories as the state space. If our optimal policy is ergodic, we can use the mixing time of the policy as our choice of $\lambda$, since this would give us a good idea of the average rewards possible with this policy in the long run. We will usually assume that we are given $\lambda$.

**Definition 3** *Let $M$ be a Markov decision process that models the environment (the opponent), and let $\pi$ be a policy in $M$ such that the asymptotic average reward $V_M^\pi = \lim_{T \to \infty} V_M^\pi(i, T)$ for all $i$, where $V_M^\pi(i, T')$ is the average undiscounted reward of $M$ under policy $\pi$ starting at state $i$ from time 1 to $T'$. The $\epsilon$-commitment time $\lambda_\pi$ of $\pi$ is the smallest $T$ such that for all $T' \geq T$, $|V_M^\pi(i, T') - V_M^\pi| \leq \epsilon$ for all $i$.*

Thus, if we are executing a policy $\pi$ learned on a particular opponent model $M$, then we must run the policy for at least $\lambda$ time periods to properly estimate the benefit of using that policy. Given a fixed commitment length $\lambda$, we may like to be able to evaluate all possible strategies in order to choose the optimal strategy. However, there are $|A|^{|A|^{2\lambda}}$ possible strategies to evaluate. Not only would this take a long time to try each possible strategy, but the regret bounds also become exceedingly weak. The expected regret after $T$ time periods is:

$$2\sqrt{e-1}|A|^{|A|^{2\lambda}/2}|A|^{2\lambda}\sqrt{T\lambda\ln|A|},$$

Clearly this amounts to a computationally infeasible approach to this problem. In traditional MDP solution techniques, we are saved by the Markov property of the state space, which reduces the number of strategies we need to evaluate by allowing us to re-use information learned at each state. Without any assumptions about the opponent's behavior, as in the classic regret minimization framework, we cannot get such benefits.

## 4 Learning Algorithms as Experts

However, we might imagine that not all policies are useful or fruitful ones to explore, given a fixed commitment length of $\lambda$. In fact, in most cases, we probably have some rough idea about the types of policies that may be appropriate for a given domain. For example, in our Prisoner's Dilemma example, we might expect that our opponent is either a Tit-for-Tat player, an Always-Defect or Always-Cooperate player, or a "Usually Cooperate but Defect with probability $p$ player", for example.

Given particular opponent assumptions, such as possible behavioral models, we may then be able to use a learning algorithm to estimate the model parameters based on observed history. For example, if we believe that the opponent may be Markov in the $\tau$-length history of joint actions, we can construct a Markov model of the opponent and use an efficient learning algorithm (such as E3 from Kearns and Singh (1998)) to learn the $\epsilon$-optimal policy in time polynomial to the number of states, $|A|^{2\tau}$. In contrast, the hedging algorithm needs to evaluate each of the exponentially large number of possible policies, namely $|A|^{|A|^{2\tau}}$ possible policies. To make this precise, we state the following lemma.

**Proposition 4** *Given a model of the opponent that is Markov in the $\tau$-length history of joint actions $\{a^i_{t-\tau}, a^{-i}_{t-\tau}, \ldots, a^i_{t-1}, a^{-i}_{t-1}\}$, and given a fixed mixing time $\lambda$, the number of actions executed by E3 and a hedging algorithm such as EXP3 in order to arrive at an $\epsilon$-optimal policy is at most $O\left(|A|^{10\tau}\right)$ for E3, and at least $O\left(|A|^{|A|^{2\tau}}\right)$ for the hedging algorithm.*

Of course, using this method, we can no longer guarantee regret minimization over all possible policies, but as we will discuss in the following section, we can choose a subset of fixed policies against which we can compare the performance of any learning algorithms we decide to use, and we can guarantee no-regret relative to this subset of fixed policies, as well as relative to the learning algorithms.

In some ways, using learning algorithms as experts simply off-loads the exploration from the experts framework to each individual learning algorithm. The computational savings occurs because each learning algorithm makes particular assumptions about the structure of the world and of the opponent, thus enabling each expert to learn more efficiently than hedging between all possible strategies.

### 4.1 Example

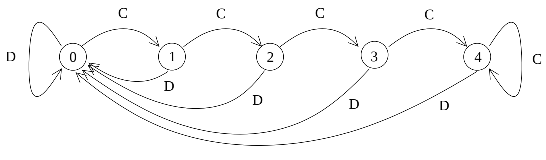


Figure 2: A possible opponent model with five states. Each state corresponds to the number of consecutive "Cooperate" actions we have just played.

For example, consider again the repeated Prisoner's Dilemma game. We might believe that the opponent reacts to our past level of cooperation, cooperating only when we have cooperated a consecutive number of times. If the opponent cooperates only when we have cooperated four periods in a row, then the opponent model shown in Figure 2 would correctly capture the opponent's state dynamics. This model is simpler than a full model using all possible 4-period histories, since it assumes that the opponent's state is completely determined by our half of the joint history. In the figure, the labeled transitions correspond to our actions, and the opponent only cooperates when it is in state 4; otherwise it defects.

To learn the optimal policy with respect this opponent model, a learning algorithm would simply have to visit all the state-action pairs and estimate the resulting reward for each possible action at each state. Since we assume that the opponent model is Markov, we can use an efficient learning algorithm such as E3.

Note that using this particular model, we can also learn the optimal policy for an opponent that cooperates if we cooperate for some given $n$ consecutive periods, where $n \leq 4$. However, if $n \geq 5$, learning using this model will no longer result in the optimal policy. Whereas choosing the cooperate action from state 4 results in a good reward when $n \leq 4$, when $n \geq 5$ the same action results in a bad reward since the opponent will most likely play defect. The problem is that the 5-state model is no longer sufficient to capture the opponent's state dynamics, and is no longer Markov.

## 5 The Hedged Learner

Since our chosen learning algorithms will sometimes fail to output good policies, we propose to incorporate them as experts inside a hedging algorithm that hedges between a set of experts that includes our learners. This allows the hedging algorithm to switch to using the other experts if a particular learning algorithm fails. It might fail due to incorrect opponent assumptions, such as in the previous section's example, or the learning algorithm may simply be ill-suited for the particular domain, or it may fail for any other reason. The point is that we have a backup plan, and the hedging algorithm will eventually switch to using these other options.

We study two methods for adding learning experts into a regret-minimization algorithm such as Auer et al.'s EXP3. It is straightforward to extend our results to other variants of EXP3 such as EXP3.P.1, which guarantees similar bounds that hold uniformly over time and with probability one.

We are given $N$ fixed experts, to which we must add $M$ learning experts. We assume that $\lambda_i = 1$ for all $i \in N$ and refer to these experts as *static experts*. These static experts are essentially the pure action strategies of the game. For all $i \in M$, we assume $\lambda_i > 1$ and note that $M$ can also include behavioral strategies. When it is clear from context, we will often write $N$ and $M$ as the number of experts in the sets $N$ and $M$, respectively.

- **Naive approach:** Let $\lambda_{max} = \max_i \lambda_i$. Once an expert is chosen to be followed, follow that expert for a $\lambda_{max}$-length commitment phase. At the end of each phase, scale the accumulated reward by $\frac{1}{\lambda_{max}}$ since EXP3 requires rewards to fall in the interval [0,1] and update the weights as in EXP3.
- **Hierarchical hedging:** Let $E_0$ denote the top-level hedging algorithm. Construct a second-level hedging algorithm $E_1$ composed of all the original $N$ static strategies. Use $E_1$ and the learning algorithms as the $M+1$ experts that $E_0$ hedges between.

## 5.1 Naive approach

The Naive approach may seem like an obvious first method to try. However, we will show that it is distinctly inferior to hierarchical hedging.

**Theorem 5** *Suppose we have a set $N$ of static experts, and a set $M$ of learning experts with time horizons $\lambda_i$. Using a naive approach, we can construct an algorithm with regret bound*

$$2\sqrt{e-1}\sqrt{\lambda_{\max}T(N+M)\ln(N+M)}.$$

**Proof.** We run EXP3 with the $M+N$ experts, with a modification such that every expert, when chosen, is followed for a commitment phase of length $\lambda_{\max}$ before we choose a new expert. We consider each phase as one time period in the original EXP3 algorithm, and note that the accumulated rewards for an expert over a given phase falls in the interval $[0, \lambda_{\max}]$. Thus, the regret bound over $\frac{T}{\lambda_{\max}}$ phases is $2\lambda_{\max}\sqrt{e-1}\sqrt{\frac{T}{\lambda_{\max}}(N+M)\ln(N+M)}$, and the result follows immediately. □

## 5.2 Hierarchical hedging

The Naive Approach suffers from two main drawbacks, both stemming from the same issue. Because the Naive Approach follows all experts for $\lambda_{\max}$ periods, it follows the static experts for longer than necessary. Intuitively, this slows down the algorithm's adaptation rate. Furthermore, we also lose out on much of the safety benefit that comes from hedging between the pure actions. Whereas a hedging algorithm over the set of pure actions is able to guarantee that we attain at least the safety (minimax) value of the game, this is no longer true with the Naive approach since we have not included all possible $\lambda_{\max}$-length behavioral experts. Thus, each expert available to us may incur high loss when it is run for $\lambda_{\max}$ periods. Hierarchical Hedging addresses these issues.

**Theorem 6** *Suppose we have a set $N$ of static experts, and a set $M$ of learning experts with time horizons $\lambda_i$, $\max_i \lambda_i > |N|$. We can devise an algorithm with regret bound:*

$$\begin{array}{ll} & 2\sqrt{e-1}\sqrt{TN\ln N} \\ + & 2\sqrt{e-1}\sqrt{\lambda_{\max}T(M+1)\ln(M+1)} \end{array} .$$

This upper bound on the expected regret improves upon the Naive Approach bound as long as

$$\lambda_{\max} \geq \frac{\sqrt{\ln N}}{\sqrt{\ln(M+N)} - \sqrt{\ln(M+1)}}.$$

In practice, we will often use only one or two learning algorithms as experts, so $M$ is small. For $M = 1$, the bound would thus look like:

$$2.63\sqrt{TN\ln N} + 3.10\sqrt{\lambda_{\max}T}.$$

However, we note that these are simply upper bounds on regret. In Section 5.3, we will compare actual performance of these two methods in a some test domains.

**Proof.** Using the bounds shown to be achieved by EXP3, our top-level hedging algorithm $E_0$ achieves performance

$$R_{E_0} \geq \max_{i \in M+\{E_1\}} R_i - 2\sqrt{e-1}\sqrt{T(M+1)\ln(M+1)}.$$

Now consider each of the $|M|+1$ experts. The $|M|$ learning experts do not suffer additional regret since they are not running another copy of EXP3. The expert $E_1$ is running a hedging algorithm over $|N|$ static experts, and thus achieves performance bounded by

$$R_{E_1} \geq \max_{j \in N} R_j - 2\sqrt{e-1}\sqrt{\lambda_{\max}TN\ln N}.$$

Combining this with the above, we see that

$$\begin{array}{rcl} R_{E_0} & \geq & \max_{i \in M+N} R_i \\ & & -2\sqrt{e-1}\sqrt{TN\ln N} \\ & & -2\sqrt{e-1}\sqrt{\lambda_{\max}T(M+1)\ln(M+1)}. \quad \square \end{array}$$

**Proposition 7** *The Hierarchical Hedging algorithm will attain at least close to the safety value of the single-shot game.*

**Proof.** From an argument similar to Freund and Schapire (1999), we know that the second-level expert $E_1$ will attain at least the safety value (or minimax) value of the single-shot game. Since the performance of the overall algorithm $E_0$ is bounded close to the performance of any of the experts, including $E_1$, the Hierarchical Hedger $E_0$ must also attain close to the safety value of the game. □

As desired, hierarchical hedging is an improvement over the naive approach since: (1) it no longer needs to play every expert for $\lambda_{\max}$-length commitment phases and thus should adapt faster, and (2) it preserves the original comparison class by avoiding modifications to the original experts, allowing us to achieve at least the safety value of the game.

**Remark.** It is also possible to speed up the adaptation of these hedged learners by playing each expert $i$ for only

Table 1: Comparison of the performance of the different methods for structuring the hedged learner.

| | Regret Bound | Actual Expected Regret | Actual Performance |
|---|---|---|---|
| Naive | 125,801 | 34,761 | -96,154 |
| Hierarchical | 36,609 | 29,661 | -8,996 |

$\lambda_i$ time periods, weighting the cumulative rewards received during this phase by $1/\lambda_i$, and using this average reward to update the weights. Applied to the hierarchical hedger, we would play each learning algorithm $i$ for $\lambda_i$-length phases and the second-level hedging algorithm $E_1$ for $N$-length phases. In practice, this often results is some performance gains.

### 5.3 Practical comparisons

We can verify the practical benefit of hierarchical hedging with a simple example. We consider the repeated game of Matching Pennies, shown in Figure 1. Assume that the opponent is playing a hedging algorithm that hedges between playing "Heads" and "Tails" every time period. This is close to a worst-case scenario since the opponent will be adapting to us very quickly.

We run each method for 200,000 time periods. The Hierarchical Hedger consists of 9 single-period experts grouped inside $E_1$ and one 500-period expert. The Naive Hedger runs all the experts for 500 periods each. The results are given in Table 1, along with the expected regret upper bounds we derived in the previous section. As expected, the hierarchical hedger achieves much better actual performance in terms of cumulative reward over time, and also achieves a lower expected regret. However, the regret for the naive approach is surprisingly low given that its performance is so poor. This is due to a difference in the comparison classes that the methods use. In the naive approach, our performance is compared to experts that choose to play a single action for 500 time periods, rather than for a single time period. Any single action, played for a long enough interval against an adaptive opponent, is a poor choice in the game of matching pennies. The opponent simply has to adapt and play its best response to our action, which we are then stuck with for the rest of the interval. Thus the expected rewards for any of the experts in the naive approach's comparison class is rather poor. For example, the expected reward for the "Heads" expert is -98,582. This explains why our expected regret is small, even though we have such high cumulative losses; we are comparing our performance against a set of poor strategies!

## 6 Experimental Results

Since the worst-case bounds we derived in the previous section may actually be quite loose, we now present some experimental results using this approach of hedged learning. We consider the repeated Prisoner's Dilemma game, and we first assume that the unknown opponent is a "Tit-for-Ten-Tats" opponent. That is, the opponent will only cooperate once we have cooperated for ten time periods in a row.

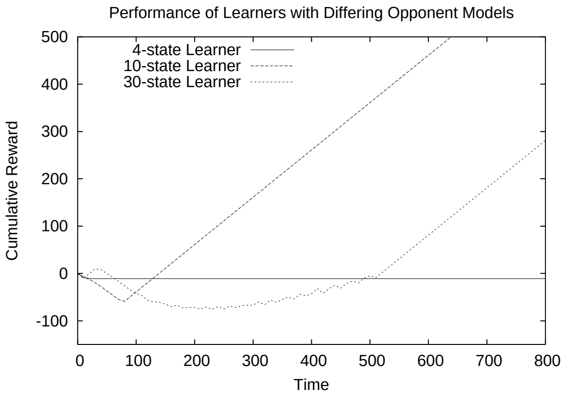


Figure 3: This graph shows the performance of learning algorithms against a Tit-for-Ten-Tats opponent. As the opponent model grows in size, it takes longer for the learning algorithm to decide on an optimal policy.

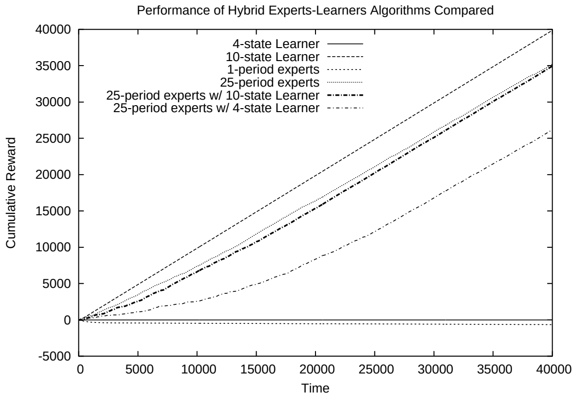


Figure 4: This chart shows the performance of different learning, hedging, and hedging learning algorithms in a game of repeated prisoner's dilemma against a Tit-for-Ten-Tats opponent.

We use a variety of different opponent models with simple learning algorithms, pure hedging algorithms that only hedge between static experts, and hedged learning algorithms that combine learning algorithms with static experts. First, we note that larger opponent models are able to capture a larger number of potential opponent state dynamics, but require both a longer commitment phase $\lambda$ and a larger number of iterations before a learning algorithm can estimate the model parameters and solve for the optimal policy. For example, Figure 3 shows the performance of three different $n$-state learners, with $n = 4, 10, 30$. As discussed earlier in Section 4, the 4-state learner is unable to capture the opponent's state dynamics and thus learns an "optimal" policy of defecting at every state. This results in an average reward of zero per time step. On the other hand, the 10-state and 30-state learners lose some rewards while they are exploring and learning the parameters of their opponent models, but then gain an average reward of 1 after they have found the optimal policy of always cooperating.

Figure 4 shows the performance of various learning and

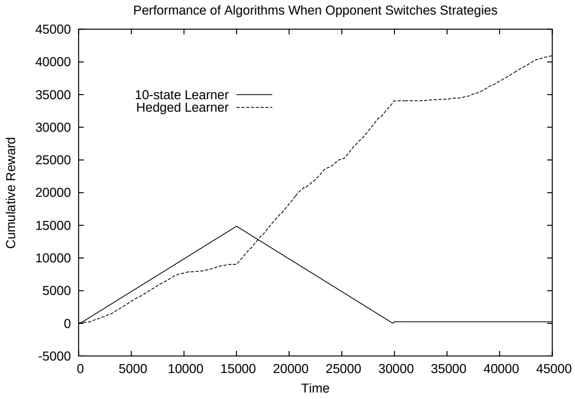


Figure 5: In these trials, the opponent switches strategy every 15,000 time periods. It switches between playing Tit-for-Ten-Tats ("Cooperate for 10 Cooperates") and "Cooperate for 10 Defects". While the modeler becomes confused with each switch, the hedging learner is able to adapt as the opponent changes and gain higher cumulative rewards.

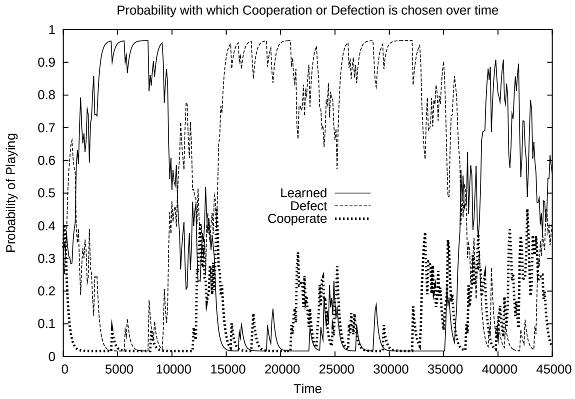


Figure 6: Graph showing the probability with which the weighted hedger plays either a cooperating strategy or a defecting strategy against the switching opponent over time.

hedging algorithms. The "1-period experts" hedging algorithm hedges between single periods of cooperating and defecting. This myopic algorithm is unable to learn the cooperative outcome and thus ends up achieving the single-shot Nash equilibrium value of 0. It assigns a very high weight to the Defect expert. On the other hand, the "25-period experts" hedging algorithm switches between two experts which either cooperate or defect for all possible 25-period histories. This algorithm realizes that the "always cooperate" expert attains higher reward and thus eventually plays Cooperate with probability approaching 1. The hedged 10-state learner is also able to achieve the cooperative outcome. It achieves cumulative reward only slightly lower than the unhedged 10-state learner, since it quickly realizes that the "always cooperate" policy and the learned optimal policy both return higher rewards than the "always defect" policy.

One main benefit of the hedged learning approach becomes evident when we observe the performance of the hedged 4-state learner. Even though the 4-state model is unable to capture the state dynamics and the learning algorithm thus fails to learn the cooperative policy, the hedged 4-state learner is able to achieve average rewards of 1 as it assigns larger and larger weight to the "always cooperate" expert and learns to ignore the recommendations of the failed learning expert. We have wisely hedged our bets between the available experts and avoided placing all our bets on the learning algorithm.

Another major benefit of using hedged learners occurs when the environment is non-stationary. For example, assume that the opponent switches between playing Tit-for-Ten-Tats ("Cooperate for 10 Cooperates") and "Cooperate for 10 Defects" every 15,000 time periods. While the unhedged learner becomes confused with each switch, the hedged learner is able to adapt as the opponent changes and gains higher cumulative rewards (Figure 5). Note that when the opponent does its first switch, the unhedged learner continues to use its cooperative policy, which was optimal in the first 15,000 periods but now returns negative average reward. In contrast, the hedged learner is able to quickly adapt to the new environment and play a primarily defecting string of actions. Figure 6 shows how the hedging algorithm is able to change the probabilities with which it plays each expert as the environment changes, i.e. when the opponent switches strategies.

## 7 Future Work

We are currently adapting these methods to stochastic games, and gathering data from experiments with simple stochastic games such as grid-world soccer. We hope to also be able to present these results at the workshop.

## References

Auer, P., Cesa-Bianchi, N., Freund, Y., & Schapire, R. E. (1995). Gambling in a rigged casino: the adversarial multi-armed bandit problem. *Proceedings of the 36th Symposium on Foundations of Computer Science*.

Chang, Y., & Kaelbling, L. P. (2001). Playing is believing: The role of beliefs in multi-agent learning. *NIPS*.

de Farias, D. P., & Meggido, N. (2004). How to combine expert (or novice) advice when actions impact the environment. *Proceedings of NIPS*.

Freund, Y., & Schapire, R. E. (1999). Adaptive game playing using multiplicative weights. *Games and Economic Behavior*, *29*, 79–103.

Fudenburg, D., & Levine, D. K. (1995). Consistency and cautious fictitious play. *Journal of Economic Dynamics and Control*, *19*, 1065–1089.

Kearns, M., & Singh, S. (1998). Near-optimal reinforcement learning in polynomial time. *ICML*.

Mannor, S., & Shimkin, N. (2001). Adaptive strategies and regret minimization in arbitrarily varying Markov environments. *Proc. of 14th COLT*.

Nachbar, J., & Zame, W. (1996). Non-computable strategies and discounted repeated games. *Economic Theory*.

# Supervised Learning of Options: A Pilot Study

**Cosmin Paduraru and Vadim Bulitko**
Department of Computing Science, University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
{cosmin, bulitko}@cs.ualberta.ca

## Abstract

Options represent a formal way of adding temporal abstraction to reinforcement learning. They have been shown to be successful in terms of accelerating learning and there is much promise in using them for knowledge representation. In this paper, we will propose supervised learning of option policies as an alternative to existing methods. This provides an easy, intuitive way of transferring knowledge from a human expert to a reinforcement learning agent. Moreover, the agent is not limited to mimicking the expert's behavior (as it would be if supervised learning were to be performed for the whole task), since an overall policy is learned on top of both options and primitive actions. Supervised option learning also has the advantage that the parts of the sensory input irrelevant to the subtask are detected automatically, thus allowing for generalization between similar structures in the same environment.

## 1 Introduction

One very promising approach to scaling up reinforcement learning is to use *options* as a form of temporal abstraction. The concept was introduced by Sutton, Precup and Singh [Sutton *et al.*, 1999], where they formally define an option as an entity consisting of three components: a partial policy $\pi : S' \times A \rightarrow [0,1]$ ($S' \subseteq S$), an initiation set $I \subseteq S$ and a termination condition $\beta : S' \rightarrow [0,1]$. These components can generate behavior as follows: as the agent interacts with the environment, it can initiate an option in any state in $I$, after which it behaves according to $\pi$. At each state $s$ the option terminates with probability $\beta(s)$.

While an important body of recent work has focused on autonomous discovery of options [McGovern, 2002], the more simple view and the one that this paper will focus on is that of options as a way of introducing expert knowledge. Options seem to be the right tool for modeling the concept of skill teaching, a concept so often encountered in human life (e.g., a baby is taught how to walk, a tennis player is taught how to serve, etc.).

One of the simplest examples of this can be found in the original options paper [Sutton *et al.*, 1999], where the authors define options for reaching doorways, knowing that the agent will have to go through a sequence of doorways before reaching the goal. There are also more realistic tasks on which this idea has proven to be successful, such as aerial surveillance [Sutton *et al.*, 1998] or robot soccer [Sutton and Stone, 2001]. More generally, defining sub-policies has been shown to speed up learning in non-trivial domains by using other forms of hierarchical abstraction as well, such as MAXQ [Ghavamzadeh and Mahadevan, 2003].

Besides its intuitive appeal, there are good reasons for which defining sub-policies can improve the learning process. First, a very large state-action space can be considerably reduced by planning or learning only on top of options, as has been shown in [Sutton *et al.*, 1998]. This effectively limits the set of policies that are available to the agent and, while it usually means that the learned overall policy is sub-optimal, the better the designer is at defining sub-policies the better the final policy can be.

Second, good options can help speed up learning by focusing the value function updates on relevant parts of the state space. For instance, in their work with Robocup soccer Sutton and Stone used sarsa($\lambda$) without decaying the traces while an option was executed [Sutton and Stone, 2001]. Generally speaking, this is equivalent to having variable values of $\lambda$, but without options it would have been more difficult to figure out a good strategy for varying $\lambda$.

We will propose in this paper an alternative to hand-coding option policies as a way of introducing prior knowledge: supervised learning of options. In the following, we will discuss the potential that this method has, elaborate on its details and support it with preliminary empirical evidence.

## 2 Reusable Options

There is one class of learning tasks for which options could represent a very prolific approach, yet insufficient investigation has been carried out in this area. These are tasks for which the environment is structured in such a way that it encourages the use of what we will call *reusable sub-policies*, or *reusable options*. These are basically sub-policies that can be used in different, yet similar areas of the state space.

There are a number of suggestive examples from human behavior: for instance, opening a door, writing a letter or closing our eyes represent sub-policies that can be applied in various situations based on some common elements of those situations. In the context of reinforcement learning, an area where such sub-policies should be abundant is robot tasks, which are usually rich in the kind of repetitive structures that we wish to exploit. As the matter of fact, hand-coding reusable sub-policies has been done for quite a while in the robotics community, and it has also been used with the options framework on Robocup soccer [Sutton and Stone, 2001].

Hand-coding of reusable sub-policies exploits regularities in the sensory space by defining sub-policies only in terms of a subset of the sensory signal. For instance, in their Robocup work, Sutton and Stone define their options only in terms of the relative angles and distances and ignore the exact positions of the robots on the field. Dietterich also acknowledges the importance of this kind of subtask-specific state abstraction, and identifies a number of conditions under which it is consistent with the MAXQ hierarchical learning [Dietterich, 2000].

In this context, it is important to move the burden of coming up with the proper state abstraction for a certain sub-policy from the programmer to a learning algorithm. In the following, we will propose a way in which this can be achieved.

## 3 Supervised learning of Options

One way of specifying sub-policies and accomplishing the goals of subtask-specific state abstraction is supervised learning of options. The idea is that a human expert controls the agent and tries to accomplish the subtask, thus generating a set of (sensory information, action) pairs. A supervised learning method is then run on these pairs, using the expert's actions as the classification targets. The option's policy will be represented by the result of applying the corresponding classifier on new observations. The same idea has been previously used for learning overall policies, such as driving an autonomous vehicle [Pomerleau, 1993].

We should note that using this method for acquiring options can lead to making them reusable. For instance, if the observation vector is large yet the expert only takes into account some of its components while controlling the agent, the supervised learning method should learn to "pay attention" to only those components in the classification process. "Paying attention" can mean a number of things in this case, such as that the corresponding weights are non-zero (or significantly greater than zero) in a linear classifier.

Another issue to mention here is that, unlike in supervised policy learning for the overall task, examples of states in which there is no point in even thinking about trying the sub-policy should also be given by the teacher. For instance, there is no point in considering opening a door if there is no door nearby or following a wall in an obstacle-free domain. In the options framework, this is translated into restricting the initiation set of an option.

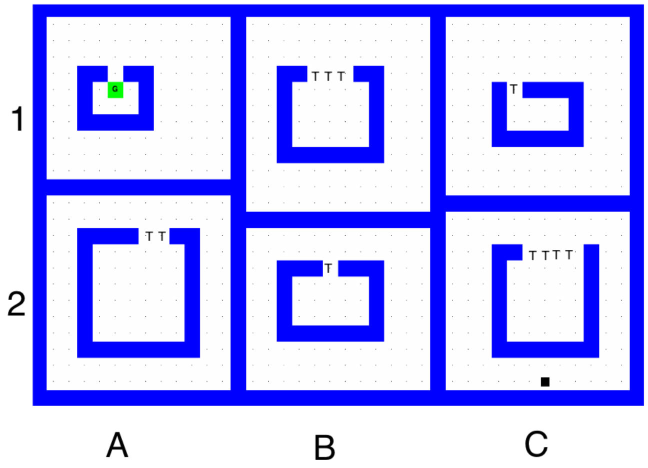


Figure 1: Our experimental "teleporting" gridworld. The agent is moved ("teleported") from each of the states marked with a "T" to the next room, in the order C2-C1-B2-B1-A2-A1. The goal is located in room A1 and is labeled with a "G".

We realize that providing such "negative" examples can be difficult for the teacher. However, the same difficulty is even aggravated when the sub-policy is hand-coded since instead of merely providing negative training examples the programmer would have to explicitly define the initiation set.

These examples have the role of specifying $S'$, the set of states for which the policy and the termination condition are defined. In the case of Markov options (options for which the policy and the termination condition only depend on the current state), it makes sense to assume that the initiation set $I$ is the same as $S'$. In this work we have made both assumptions: that the options are Markov (one reason for this was to be able to use intra-option learning [Sutton *et al.*, 1999]) and that $I = S'$.

Finally, we have to note that supervised policy learning methods have been generally avoided in the context of autonomous agents, mainly because the learned policy was constrained to mimic the teacher's behavior, thus making it sub-optimal in many cases. However, in our approach this disadvantage is compensated for as the agent can identify useless or harmful options while learning the overall policy and learn not to use them or use them in a very restricted part of the environment.

## 4 Preliminary results

We have empirically tested supervised learning of options in a gridworld-style domain, designed in such a way that useful options were easy for a human to come up with. The domain is shown in Figure 1, the task is episodic and the goal is to reach the square marked with a "G". The world is deterministic and the four available actions (up, down, left, right) have the usual outcomes, except when the agent is "teleported" after entering each

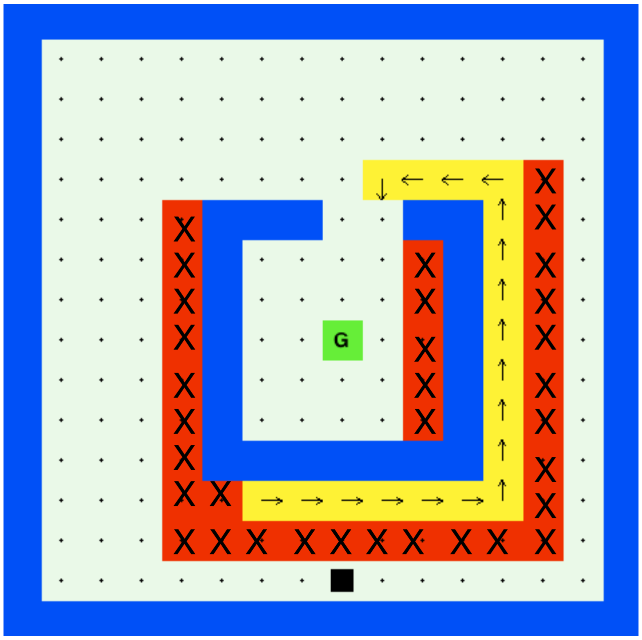


Figure 2: Training data provided for the "get inside the room through the right side" option. The 'X' means that the corresponding state has been labeled as "option inapplicable". For states for which a training action has been provided, that action is indicated by the direction of the arrow.

of the inside rooms. The teleportation is done in a sequence that eventually leads it to the room where the goal is located.

The option that we have taught the agent was to get to the entrance of the inside rooms (where teleporting would then occur) from states next to the inside walls. It should be clear that this option is helpful on our task, since the agent needs to get to the teleporting places in order to reach the goal.

During learning, the observation vector was composed of the unique state label for each square and 8 binary inputs for the 8 directions (up, down, left, right and diagonals). Each of the binary inputs had a value of 1 if there was a wall in the corresponding direction and of 0 otherwise.

We have generated training examples (including examples of places where the option is not initiable) for rooms of different sizes (see Figure 2) and trained a decision tree with these examples. The examples where generated such that the agent would be taken to the teleporting spots by going on the right side of the room.

We have observed that, as expected, the decisions that the decision-tree based classifier made were based on the last 8 components of the input vector rather than the state label. This allows the agent to use the same option for each of the rooms. Indeed, if the policy is defined only in terms of the squares around the agent, it will work for any room, no matter what the exact position of that room is or how long the walls are.

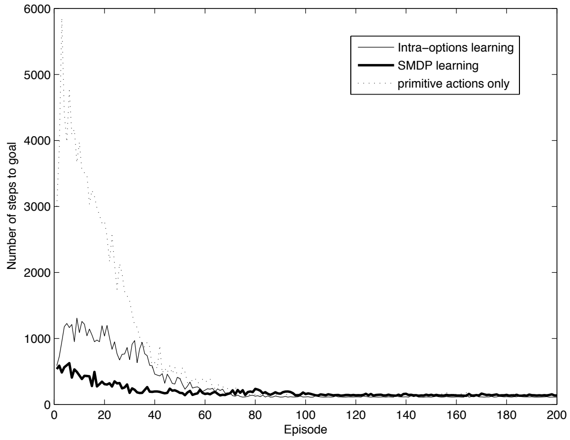


Figure 3: Using options defined by supervised learning can speed up learning. Each data point is averaged over 20 folds.

To test the utility of our "reusable option", we compared learning with primitive actions only against learning with both primitive options and the decision-tree learned option. We used sarsa($\lambda$) as the learning algorithm. Both SMDP and intra-option methods [Sutton *et al.*, 1999] were employed when learning with both actions and options. The rewards were 1 if the goal was reached and 0 for any other transition, and the control parameters were $\gamma = 0.9, \lambda = 0.9, \alpha = 0.2$.

The results, presented in Figure 3, show a considerable improvement in convergence speed when the learning algorithm made use of the option. This shows that the agent was able to generalize from expert-provided training examples and apply the option in every room. Generally speaking, these results suggest that including expert knowledge via supervised learning of options is a viable solution for speeding up reinforcement learning in complex domains.

## 5 Conclusions and Future Work

The following question appears important for temporal abstractions and hierarchical reinforcement learning: What is the proper way to go from our idea of what the agent's sub-policies should be to the agent's representation of these sub-policies? This paper discussed supervised learning of options as a candidate answer to this question. Strengths of the proposed approach include the fact that expert knowledge can be introduced without any programming effort on the user side and that the subset of sensory information relevant to the subpolicy can be automatically selected. The discussion is supported with a pilot empirical study in a toy domain.

The promise of this preliminary work warrants a larger scale investigation, possibly using robot simulation such as the RoboCup competition [Kitano *et al.*, 1997] . A first issue to be investigated is how appropriate different function approximation schemes (such as neural networks, decision trees, linear regression etc.) are for representing option policies.

We also intend to compare different methods of providing a reinforcement learning agent with options (or skills), including our approach, subgoal specification, manual programming, reward system specification, etc. The option policies generated will be used with one of the known option learning or planning algorithms [Sutton *et al.*, 1999], and the relative performances will suggest which of the methods has produced the most robust and useful policies.

We hope that, by doing this, we will get a clear idea of which of these methods are more intuitive, easy and practical. For instance, we may come across subtasks that no expert knows how to solve yet which can be well handled by reinforcement learning algorithms based on subgoal values [Sutton *et al.*, 1999]. Or, perhaps, we can find teaching to be the better choice in practice, because providing examples turns out to be a lot easier than specifying the proper option-specific rewards or subgoal values for each option.

## Acknowledgments

We thank Mark Ring and Rich Sutton for invaluable input and overall wisdom. We are also grateful for the financial support from iCORE and the University of Alberta.

## References

[Dietterich, 2000] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[Ghavamzadeh and Mahadevan, 2003] Mohammad Ghavamzadeh and Sridhar Mahadevan. Hierarchical policy gradient algorithms. In *20th International Conference on Machine Learning (ICML-03)*, Washington, DC, 2003.

[Kitano *et al.*, 1997] Hiroaki Kitano, MilindTambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The robocup synthetic agent challenge. In *15th International Joint Conference on Artificial Intelligence*, San Francisco, CA, 1997.

[McGovern, 2002] Amy McGovern. *Autonomous Discovery of Temporal Abstractions from Interacting with an Environment.* PhD thesis, University of Massachusetts, Amherst, Massachusetts, 2002.

[Pomerleau, 1993] D. Pomerleau. *Neural Network Perception for Mobile Robot Guidance.* Kluwer Academic Publishing, 1993.

[Sutton and Stone, 2001] Richard S. Sutton and Peter Stone. Scaling reinforcement learning toward robocup soccer. In *18th International Conference on Machine Learning (ICML-01)*, 2001.

[Sutton *et al.*, 1998] Richard S. Sutton, Satinder Singh, Doina Precup, and Balaraman Ravindran. Improved switching among temporally abstract actions. In *Neural Information Processing Systems 11 (NIPS-98).* MIT Press, 1998.

[Sutton *et al.*, 1999] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–121, 1999.