MONOGRAFIES DE L'INSTITUT D'INVESTIGACIÓ
EN INTEL·LIGÈNCIA ARTIFICIAL
Number 26

Institut d'Investigació
en Intel·ligència Artificial

Consell Superior
d'Investigacions Científiques

Monografies de l'Institut d'Investigació en Intel·ligència Artificial

Num. 1 J. Puyol, *MILORD II: A Language for Knowledge–Based Systems*

Num. 2 J. Levy, *The Calculus of Refinements, a Formal Specification Model Based on Inclusions*

Num. 3 Ll. Vila, *On Temporal Representation and Reasoning in Knowledge–Based Systems*

Num. 4 M. Domingo, *An Expert System Architecture for Identification in Biology*

Num. 5 E. Armengol, *A Framework for Integrating Learning and Problem Solving*

Num. 6 J. Ll. Arcos, *The Noos Representation Language*

Num. 7 J. Larrosa, *Algorithms and Heuristics for Total and Partial Constraint Satisfaction*

Num. 8 P. Noriega, *Agent Mediated Auctions: The Fishmarket Metaphor*

Num. 9 F. Manyà, *Proof Procedures for Multiple-Valued Propositional Logics*

Num. 10 W. M. Schorlemmer, *On Specifying and Reasoning with Special Relations*

Num. 11 M. López-Sánchez, *Approaches to Map Generation by means of Collaborative Autonomous Robots*

Num. 12 D. Robertson, *Pragmatics in the Synthesis of Logic Programs*

Num. 13 P. Faratin, *Automated Service Negotiation between Autonomous Computational Agents*

Num. 14 J. A. Rodríguez, *On the Design and Construction of Agent-mediated Electronis Institutions*

Num. 15 T. Alsinet, *Logic Programming with Fuzzy Unification and Imprecise Constants: Possibilistic Semantics and Automated Deduction*

Num. 16 A. Zapico, *On Axiomatic Foundations for Qualitative Decision Theory - A Possibilistic Approach*

Num. 17 A. Valls, *ClusDM: A multiple criteria decision method for heterogeneous data sets*

Num. 18 D. Busquets, *A Multiagent Approach to Qualitative Navigation in Robotics*

Num. 19 M. Esteva, *Electronic Institutions: from specification to development*

Num. 20 J. Sabater, *Trust and Reputation for Agent Societies*

Num. 21 J. Cerquides, *Improving Algorithms for Learning Bayesian Network Classifiers*

Num. 22 M. Villaret, *On Some Variants of Second-Order Unification*

Num. 23 M. Gómez, *Open, Reusable and Configurable Multi-Agent Systems: A Knowledge Modelling Approach*

Num. 24 S. Ramchurn *Multi-Agent Negotiation Using Trust and Persuasion*

Num. 25 S. Ontañón *Ensemble Case Based Learning for Multi-Agent Systems*

Num. 26 M. Sánchez *Contributions to Search and Inference Algorithms for CSP and Weighted CSP*

Contributions to Search and Inference Algorithms for CSP and Weighted CSP

Martí Sánchez Fibla

Foreword by
Javier Larrosa Bondia and Pedro Meseguer González

2006 Consell Superior d'Investigacions Científiques
Institut d'Investigació en Intel·ligència Artificial
Bellaterra, Catalonia, Spain.

Series Editor
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Foreword by

Javier Larrosa Bondia
Associate Professor at Universitat Politècnica de Catalunya

Pedro Meseguer González
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Volume Author
Martí Sánchez Fibla
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Institut d'Investigació
en Intel·ligència Artificial

Consell Superior
d'Investigacions Científiques

NIPO: 653-06-043-1
ISBN: 84-00-08433-0
Dip. Legal: B.36352-2006

Ordering Information: Text orders should be addressed to the Library of the IIIA, Institut d'Investigació en Intel·ligència Artificial, Campus de la Universitat Autònoma de Barcelona, 08193 Bellaterra, Barcelona, Spain.

Contents

List of Figures

Foreword

Many important problems can be modelled in terms of constraints and solved reasoning about them. This book deals with CSP and Weighted CSP, which are the satisfaction and the optimization frameworks of constraint reasoning, respectively.

This work presents contributions to search and inference, the two main solving approaches. Search methods traverse the space of possible configurations and their efficiency depends on their ability to make right guesses and to identify and scape from mistakes. Inference methods -in their complete version- perform a sequence of solution-preserving simplifications until obtaining a trivially solvable problem.

Conventional wisdom in constraint reasoning says that the efficiency of search algorithms greatly depends on their ability to perform the right amount of look-ahead after each assignment. Similarly, the efficiency of inference algorithms depends on their ability to exploit the problem structure. The novelty of this thesis is to break such a hard dichotomy: it introduces efficient search algorithms that exploit the problem structure and efficient inference algorithms that perform some sort of look-ahead.

The proposed methods are completely generic, in the sense that they do not assume any particular graph topology and they do not consider any particular constraint type. Since the considered problems are computationally intractable, algorithm evaluation has been done empirically, on random problems and on different benchmarks coming from the real world, such as earth observation satellite management and radio link frequency assignment.

Bellaterra, July 26 2006

Javier Larrosa Bondia
Associate Professor, UPC

Pedro Meseguer González
Researcher of IIIA-CSIC

Acknowledgements

Agradezco tanto al IIIA y a toda su gente que me ha soportado y dado soporte. Me siento incapaz de nombrarlos a todos, con lo que mando un abrazo y una sonrisa enorme a todos los que se sientan identificados. Estar en un despacho solo y encerrado es muy duro y todos ellos lo saben. Agradezco, eso sí, a Pedro la enorme paciencia que ha tenido conmigo y que ha hecho posible que algunas de muchas ideas y más ideas (muy malas, mejores, complicadas o falsas), algunas pocas, se pudieran traducir al papel y ser explicadas paso a paso con la formalidad que requerían. Agradezco a la gente del LSI que me conoce, su soporte con igual ímpetu. Especialmente, doy las gracias a Javier por su siempre preciosa visión global y porque siempre estuvo allí. Agradezco tantísimo a mi familia su apoyo. En las cenas que hacemos cada martes, mi estado de ánimo ha estado estrechamente ligado a las aventuras y desventuras de la investigación llevada a cabo, increíble, pero gracias. A todos mis amigos del alma, por aguantar lo que nadie de nosotros debería tener, cualquier cosa que pudiera parecerse a un monotema: es insoportable. Entre cotas inferiores y cotas superiores, de las cuales esta tesis está llena, allí estuvo en un momento muy importante la única Cota superior a todos y a todas. Y ya desviándome un poco del tronco principal que ha levantado este trabajo agradezco al pasatiempos Sudoku ser capaz de arrancar una sonrisa a la gente cuando uno explica de que va esta tesis. Agradecimientos totalmente insuficientes. Me siento en deuda con todos y todas, un beso a todos, a ti también Dani. Y me recuerdo a mí mismo antes de despedirme que la ciencia, y muchas otras cosas, son tan sólo una minúscula parte, seguramente imprescindible, de todo lo demás, o eso creo. Hasta pronto!

Abstract

This thesis presents a collection of new algorithms for solving *Constraint Satisfaction Problems* (CSP) and *Weighted Constraint Satisfaction Problems* (WCSP). We pursue two main objectives: enhancing solving methods for WCSP, which are of recent development, and narrowing the gap between search and inference methods. The first part of the thesis is devoted to search methods for solving WCSP. In a branch-and-bound context, the lower bound computation in each node of the search is of great importance and has a serious impact in the practical efficiency of algorithms. We start from an algorithm called Russian Doll Search (RDS) that has a costly, yet very powerful lower bound and we develop three enhancements of it: *Specialized RDS (SRDS)*, *Full SRDS* and *Opportunistic SRDS* [Meseguer and Sanchez, 2001] [Meseguer et al., 2002]. We then tackle the problem of exploiting the global structure of the problem inside search. An algorithm exists for CSP that is able to exploit what is called pseudo-tree structure extend it to WCSP, obtaining algorithm *Pseudo Tree Partial Forward Checking (PT-PFC)*. This algorithm has a source of inefficiency mainly related to bad quality local lower and upper bounds. We suggest a solution to this problem by combining pseudo-tree search for WCSP with the RDS techniques that we previously developed obtaining algorithms *PT-RDS* and *PT-SRDS* [**?**] [Larrosa et al., 2002]. In all this first part the aim is enhancing the practical efficiency of existing search algorithms with respect to the time spent in solving several benchmarks. The second part of the thesis is devoted to complete inference methods for solving CSP and WCSP. Complete inference solves the problem by a sequence of transformations that obtain an equivalent problem. In these transformations variable elimination plays an important role. We present some new inference operations that permit us to factorize a constraint into a set of smaller size constraints. We then introduce factorization into variable elimination. The result is algorithm *Adaptive Consistency with negative factorized constraints (ADC$^{\leftarrow}_{factor}$)* [Sanchez et al., 2004b]. With these operations we define also an alternative method to eliminate a binary domain variable. Next, we introduce the idea of *Filtering* which consists in anticipating tuples of constraints that will become inconsistent when joined with other constraints of the problem. One could say that we are doing the equivalent to look-ahead during search but the goal is to reduce memory storage instead of pruning branches and reduce time. We introduce filtering in the main complete inference algo-

rithms for CSP and WCSP producing *Delayed Variable Elimination with Filtering ADC (ADC-DVE-F)* [Sanchez et al., 2004a] [Sanchez et al., 2005a] and *Bucket Elimination-DVE-F* algorithms. Still in the complete inference context, we generalize filtering to tree decomposition methods that yield us to *Cluster Tree Elimination CTE-F* algorithm. We also present an iterative pure inference algorithm that performs a sequence of more accurate approximations to solve the problem, we call it *Iterative Mini Cluster Tree Elimination (IMCTE)* [Sanchez et al., 2005b]. All contributions to the complete inference part of the thesis are devoted to enhance the memory spent by these methods.

Chapter 1

Introduction

This thesis deals with algorithms for *constraint satisfaction* and *weighted constraint satisfaction problems*. Many problems arising in *Artificial Intelligence* and other areas of Computer Science can be naturally expressed as one of these models. As a consequence research in their solving methods is gaining importance as it has great impact in many areas.

A *Constraint Satisfaction Problem* (CSP) [Mackworth, 1977] consists of a set of *variables*, each one taking values in a finite *domain*. Variables are related by *constraints* which impose restrictions to the values that variables can simultaneously take. A constraint involves a certain number of variables and has information on which combinations of values of its variables are permitted and which ones are forbidden. Solutions are assignments of values to variables that satisfy all constraints. A constraint may be represented by different means:

- mathematical expressions (e.g. $x_1 < x_2$).
- computing procedures (e.g. $isPrime?(x_1)$).
- a specific semantic (e.g. the all-different constraint, $\texttt{all-diff}(x_1, x_2, ...)$ which is equivalent to saying that variables it involves must take different values).

In all these cases a constraint is defined implicitly. A constraint can also be defined explicitly, by giving the set of permitted combinations ({yes if $\langle x_1, x_2 \rangle \in \{\langle a, a \rangle, \langle a, b \rangle, \langle b, a \rangle\}$, no in any other case}). CSP problems are classified under the NP-complete category (for more details see [R.M.Haralick and L.G.Shapiro, 1979]).

Often it happens that a CSP instance has no solution. In that case, the question of which assignment best respects all constraints arises. The CSP model is not sufficient to answer this question and it has to be augmented by the so called *soft* constraint framework ([Schiex et al., 1995, Bistarelli et al., 1995]). *Soft* constraints are constraints that give a degree of acceptability of a combination of values of its variables, so they can be seen as cost functions that assign a cost

to the combinations of values of the variables they relate. Permitted tuples are assigned a zero cost, forbidden tuples are assigned an unacceptable cost and partially permitted tuples are assigned an intermediate cost. To maintain coherence with the CSP model, constraints are called *hard* when they assign an unacceptable cost to forbidden tuples and a zero cost to permitted ones. Therefore, hard constraints must be mandatorily satisfied. We have chosen as representant of this class of problems the *Weighted* CSP model (WCSP) [Larrosa and Schiex, 2003]. In WCSP the cost of an assignment of values to variables is computed as the sum of the costs of constraints that have all their variables assigned. A solution to a WCSP is an assignment that satisfies all hard constraints. The optimal solution is the solution with minimum cost. WCSP problems are classified under the NP-hard category.

Typical (W)CSP problems can be found in many areas: *machine vision*, *belief maintenance*, *scheduling* [Minton et al., 1992], *propositional reasoning* [Selman et al., 1992], *temporal reasoning*, *VLSI circuit design*, *combinatorial problems*. Specific well-known examples of (W)CSPs are: *n-Queens problem*, *crossword puzzles*, *sudoku puzzles*, *timetabling* [Schaerf, 1999], *car sequencing*, *configuration*, *frequency assignment* [Cabon et al., 1999], *combinatorial auctions* [Sandholm, 1999] [Cramton et al., 2006], *graph coloring*, etc. Consider as example the graph coloring problem. The aim is to color (from a finite set of colors) each region of a map such that no adjacent regions have the same color. A possible CSP modeling of this problem has a variable per region. The domain of each variable is the set of possible colors. Constraints involve all pairs of adjacent regions and force them to take a different color. Consider now that we have an instance with no solution. We can convert the graph coloring problem to an optimization problem and model it as a WCSP in the following way. Constraints are now cost functions that assign a cost of one when two regions share the same color and zero otherwise. The objective is now to minimize the total cost which is equivalent to minimize the number of adjacent regions that share the same color.

(W)CSP are in close relation to other models that have important research communities. For instance a problem formulation in integer linear programming (ILP), propositional logic (SAT) and bayesian inference can be trivially translated to (W)CSP and vice versa. The advantage of (W)CSP it is commonly assumed to be its compact constraint representation. The generality of the definition of a constraint gives to the (W)CSP models a great deal of expressiveness. But this fact has also its drawbacks, its solving methods are less specific and sometimes less powerful than specialized ones for particular domains.

There exist two families of solving methods for (W)CSP: *Search* and *Inference*. In between these two classes we find hybrids procedures combining both approaches.

Search consists in searching in a space of states. The search space can be represented by a depth-bounded tree that is a representation of all possible combinations of values that variables can take. Search is called *complete* if the

exploration of the search space is systematic and it is conducted until a solution is found or it is proven not to exist (CSP). For the WCSP case complete search is able to prove the optimality of a solution. In both cases search explores this search space exploiting the fact that parts of the unexplored search space can be avoided if it can be proven that it contain no solution (CSP) or no optimal solution (WCSP). To do this, a fruitful strategy is to propagate previous decisions in the current branch. Search has an exponential time complexity in the number of variables of the problem, but it is usually the preferred option as some specific types of search have polynomial space complexity. Examples of search algorithms among the different mentioned models are *Backtrack* (CSP), *Branch and Bound* (ILP [Doig and Land, 1960] and WCSP) and *Backtrack-based Davis-Putnam-Logemann-Loveland* [Davis et al., 1962] (SAT). A survey on WCSP search methods can be found at [Meseguer et al., 2003]. Search can also be *incomplete* also called *local*. Local search partially explores the search space and cannot prove the non existence of a solution (CSP) and neither its optimality (WCSP). Examples of local search algorithms are *tabu search*, *simulated annealing*, *hill climbing*, *genetic algorithms*, ...

Inference consists in transforming the problem into an equivalent one that is supposed to be easier to solve. A transformation operates with constraints to deduce new explicit constraints (that were implicit in the original problem formulation) and reduce the size or the complexity of the problem. Inference is called *complete* when it can solve the problem without the use of search. We call *incomplete* inference (also named local) when it cannot find by itself solutions and has to be combined with search.

Given a particular (W)CSP instance, the global interaction of its variables and constraints is commonly represented by the graph that variables and constraints define. The structure refers to the kind of constraint graph that the problem has. There exist parameters in graph theory that measure the cyclicity of the constraint graph (that measure how far from being acyclic is the graph). An example of such a parameter is the *induced width*. A problem has a low induced width structure if it is close to being acyclic. Complete inference algorithms have exponential space and time complexities in the induced width of the constraint graph. Examples of Complete Inference algorithms among the different mentioned models are *Adaptive Consistency* [Dechter and Pearl, 1987] (CSP), *Bucket Elimination* [Dechter, 1999] (Bayesian Inference and WCSP) and *Directional Resolution* [Davis and Putnam, 1960] (SAT).

The objective of this thesis is to contribute in both CSP and WCSP solving methods. Both models have common features that can be exploited for algorithmic development. Moreover recent advances in WCSP make the effort of maintaining the coherence with CSP, that is intend to be the logic extension from CSP to WCSP. Also the fact that we present contributions in both, search and inference solving methods is not casual. There is in all this work the objective of narrowing the gap between both families of methods, bringing to search the advantages of inference and vice versa. A well known way of combining search and inference is by using a form of local incomplete inference inside

search. In this direction arc consistency, which is a form of incomplete inference, is of main importance. Soft Arc Consistency (SAC) [Larrosa and Schiex, 2004] is the recent extension of arc consistency to the WCSP model and is of recent development. SAC has appeared in parallel to our work. We found SAC of major importance but we followed another line of research. On the search side we focus on *Russian Doll Search* [Verfaillie et al., 1996] another method for lower bound computation and on the inference side we focus on complete inference. Ways of combing SAC with our contributions are sketched.

1.1 Motivation

As CSP is classified NP-complete and WCSP NP-hard, all algorithms for these problems will present an exponential worst case behavior. In this situation, and considering the practical importance of constraint satisfaction, developing algorithms able to solve in practice the considered problems, using a reasonable amount of resources on average, is of obvious interest. As better algorithms are developed, larger and more difficult problems instances can be successfully considered. As (W)CSP can model problems in many areas, developments in the applicability of its solving methods have widespread repercussion.

Algorithms for the Soft CSP framework are of recent development [Shapiro and Haralick, 1981, Rosenfeld et al., 1976]. The latter reference it is often considered to be the seminal paper on the topic of fuzzy constraints. Partial Forward Checking (PFC) [Freuder and Wallace, 1992] can be considered the first algorithm for Soft CSP. PFC is an extension of a CSP algorithm and since then the effort of extending CSP methods to WCSP and also producing new efficient algorithms continues. The first motivation of the thesis was to contribute in this new area and started within an European project of the same subject called ECSPLAIN.

The other big motivation was born in the relation between the two families of solving methods: search and inference. What we call the structure of the problem plays an important role in this relation. All the search algorithms that have been developed in the state-of-the-art are in fact a combination of search and a form of local inference that is performed in each node of the search. This particular hybridization of both methods has a weakness that is its blindness to the global structure of the problem. They are not able to exploit low induced width problems. The only guide these algorithms can use to be aware of the global structure is heuristics, that is, rules of thumb to help in the selection of the next variable to assign or the next value to select. There is experimental evidence that search algorithms can perform badly on large problems with low induced width. For example [Givry et al., 2003] show a bad performance on MAX-SAT dubois instances that have a low induced width. Complete Inference methods on the other hand, make use of the structure of the problem as a main step and perform efficiently in low induced width problems but usually are not considered of practical use in problems that have a high induced width.

Thus a motivation was born from this observation and it was to explore ways of exploiting the global structure in search algorithms, and vice versa, to explore how could complete inference make use of the search advantages.

1.2 Scope and Orientation

In this section we state the decisions that we have taken in our research that define the limits of our approach.

- *Practical Constraint Solving.* All work is devoted to improve the practical applicability of (W)CSP algorithms. Our contributions are new algorithms that have been implemented and that we prove to be more efficient that existing state of the art algorithms, at least in specific kind of problems. The final aim is to apply these algorithms to real problems.

- *General Constraint Solving.* CSP and WCSP solving are computationally untractable due to the fact that they belong to NP-complete and NP-hard classes, respectively. One way to circumvent this intrinsic drawback is to characterize subclasses of (W)CSP that can be efficiently solved (either we suppose an specific semantic of constraints, or we suppose a specific problem structure). A lot of effort has been recently devoted to increase the library of implicit constraints and to enhance its specific algorithms to prune variable domains during search. However, our research does not fall into this line of work. We do not make any assumption about the problems that we attempt to solve. In practice, it means that our algorithms consider a (W)CSP in its explicit form, where constraints are given explicitly and we do not assume any specific constraint semantics.

- *Exact constraint solving.* Another approach to circumvent the computational intractability of constraint solving is to use approximation approaches: incomplete for CSP and suboptimal for WCSP. These approaches typically try to solve the problem using local optimization or making some relaxation of the problem statement. However in our work we are concerned with algorithms that can find all solutions (for CSP) and all optimal solutions (for WCSP). In Section 7.3.1 we make use of an approximate inference method but it is only used to help in finding all optimal solutions.

- *Empirical evaluation.* Because of the practical orientation of our work and the recognized exponential worst-case behavior of our algorithms, the assessment of our contributions is mainly supported by empirical methods. In our experimental evaluation we use a variety of benchmarks widely used in the CSP community. These benchmarks are described in Appendix A.

Search

All algorithms developed in the search part are for WCSP. We are concerned with backtrack based algorithms. These algorithms do a depth-first traversal of the search tree such that each level corresponds to a different variable and tree nodes correspond to the different values that the variable of each level can take. We restrict ourselves to *systematic* search algorithms, meaning that the optimal solution (WCSP) has to be found. The search tree has to be completely explored and if a subtree is skipped the algorithm has to be able to prove that it contains no optimal solution. Search algorithms for WCSP keep track of the cost of the best solution found so far (called upper bound) and compute at each node an underestimation of the cost of the unexplored subtree underneath the node (called lower bound). The process of computing the lower bound and pruning future domain values that cannot belong to an optimal solution is called look-ahead. The efficiency of algorithms largely depends on this lower bound computation.

Inference

In the inference part we are concerned with complete inference. Complete inference methods are able by itself to find all the solutions of the problem without doing any subsequent search. They perform a sequence of transformations of the problem. At each transformation they operate with constraints and eliminate variables when possible to obtain a smaller equivalent subproblem. When no variables are left the solution can be trivially obtained.

1.3 Contributions

Search

- In the new framework of soft constraints we started from an algorithm that has a powerful lower bound computation. It is called Russian Doll Search (RDS, [Verfaillie et al., 1996]) and its principle is to solve the problem by several searches, adding one variable at each time, and reusing the obtained optimal cost at each resolution for improving the lower bound computation in latter searches. One could say that RDS solves the problem by first solving a simplified version of it and then reuses this resolution to solve the whole problem. In this sense it reminds of Dynamic Programming techniques. We developed several extensions of RDS following the principle of obtaining more information of each resolution for improving the lower bound computation of latter searches. The developed algorithms are more efficient in certain kinds of problems. These ideas were published in

  – "Specializing Russian Doll Search" by Pedro Meseguer, Marti Sanchez. In *Proceedings of Principles and Practice of Constraint Programming*, CP 2001. LNCS 2239.

– "Opportunistic Specialization in Russian Doll Search" by Pedro Meseguer, Marti Sanchez and Gerard Verfaillie. In *Proceedings of Principles and Practice of Constraint Programming*, CP 2002. LNCS 2470.

- How can we make use of the global structure of the problem to improve search? An algorithm exists for CSP that first decomposes the problem, identifies independent subproblems during search, as it assigns variables, and solves the subproblems independently. This algorithm is called *Pseudo-tree search* [Freuder and Quinn, 1985, Bayardo and Miranker, 1995]. Our starting point was this algorithm. We develop several extensions of it to WCSP. The main advantage is that the theoretical complexity of the new algorithms is exponentially bounded by the decomposition parameter, so it theoretically improves the time and space complexities of all other search algorithms for WCSP up to the moment. We observed a major difficulty of the extension to WCSP that caused the algorithm to be inefficient in certain cases. We developed a solution to this problem by extending the RDS techniques developed to the WCSP pseudo-tree search. These ideas are gathered in,

  – "A tree-based Russian Doll Search" by Pedro Meseguer and Marti Sanchez in *Workshop on Soft Constraints* held in CP 2000.

  – "Pseudo-tree search with soft constraints" by Javier Larrosa, Pedro Meseguer and Marti Sanchez. In *European Conference on Artificial Intelligence* ECAI 2002.

Inference

- Complete inference main drawback is that it has an exponential space complexity with respect to the induced width. As we mentioned the induced width is a measure of the cyclicity of the constraint graph so captures the global interaction of variables and constraints. If the constraint graph does not have a low induced width, complete inference can quickly exhaust memory resources. The second part of the thesis presents several algorithms for CSP and WCSP complete inference methods that decrease the memory consumption of the existing algorithms. The first idea consists in factorizing a constraint into an equivalent set of smaller constraints. We introduce a new formalism to deal with factorization. Incorporating this idea into complete inference algorithms we are capable of reducing significantly the total memory used in its execution. Using this same formalism we introduce a new complete inference operation that is able to eliminate a binary domain variable from the problem. These ideas can be found in,

  – "Using constraints with memory to implement variable elimination" by Marti Sanchez, Pedro Meseguer and Javier Larrosa. In *Proceedings of the European Conference on Artificial Intelligence* , ECAI 2004.

- Another idea is to take advantage of the whole problem when operating with constraints, to detect that a combination of values will not be allowed when extended to other parts of the problem. These combinations can then be deleted from constraints which reduces memory usage. In a sense we are making use of a kind of look-ahead in a complete inference context. The developed techniques are proven to be very powerful and greatly reduce the memory spent on average. We finally present an iterative algorithm that performs successive approximations of the problem and reuses previous iterations to reduce the memory spent on subsequent iterations. Three publications develop these ideas,

  - "Improving the Applicability of Adaptive Consistency: Preliminary Results" by Marti Sanchez, Pedro Meseguer and Javier Larrosa. Poster in *Principles and Practice of Constraint Programming* CP 2004.
  - "Improving Tree Decomposition Methods With Function Filtering" by Marti Sanchez, Javier Larrosa and Pedro Meseguer. Poster in *International Join Conference on Artificial Intelligence* IJCAI 2005.
  - "Tree Decomposition with Function Filtering" by Marti Sanchez, Javier Larrosa and Pedro Meseguer. In *Principles and Practice of Constraint Programming* CP 2005.

- Two publications accompany these contributions: one is the participation in a survey of Soft CSP techniques, the other one is the extension of the PFC algorithm to non binary constraints.

  - "Current approaches for solving overconstrained problems" by Pedro Meseguer, N. Bouhmala, T. Bouzoubaa, M. Irgens, Marti Sanchez. In Constraints Journal 2003.
  - "Lower Bounds for Non-binary Constraint Optimization" by Pedro Meseguer, Javier Larrosa and Marti Sanchez. In *Proceedings of Principles and Practice of Constraint Programming*, CP 2001. LNCS 2239.

1.4 Overview

The thesis contains eight Chapters and two Appendix. The work is divided in two parts: Search and Inference. Both collect the contributions made in each family of solving methods. Chapter 2 contains all the necessary terminology and the algorithms needed to understand the subsequent Chapters. It is not intended to be an exhaustive state-of-the-art. It introduces some basic concepts for CSP and WCSP. Regarding CSP, it covers depth-first backtrack forward checking in the search side and adaptive consistency in the complete inference

side. Regarding WCSP, it covers branch and bound, partial forward checking in the search side and bucket elimination in the complete inference side. When more specific concepts are used, they are introduced in the corresponding Chapters.

Chapter 3 is devoted to enhance lower bound computation in branch and bound WCSP search. We take as starting point an existing algorithm called Russian Doll Search (RDS) that has a powerful lower bound computation. We then develop three enhancements of it that we call *Specialized* RDS, *Full Specialized* RDS and *Opportunistic* RDS. We finally prove their performance using Random Problems and Frequency Assignment benchmarks. The Chapter ends by putting RDS techniques in the context of other recent algorithms of the kind and by sketching some possible wider applications that RDS may have.

Chapter 4 is about exploiting the global structure of the problem (the interaction among variables and constraints) in WSCP search. We extend an existing algorithm called pseudo-tree search to the WCSP framework. We first present a basic extension of it and an enhanced one that still has a source of inefficiency. To tackle this problem we combine pseudo-tree search for WCSP with the RDS algorithms of previous Chapter and obtain algorithm PT-PFC-SRDS. Then, experimentation of the obtained algorithms in random problems is done. We introduce at the end of the Chapter recent similar and alternative decomposition algorithms (AND/OR search, BTD) which we found of main importance. We end by describing some prospects of future work.

We then enter in the inference part. Chapter 5 first introduces the basic algorithm for complete inference, Adaptive Consistency (ADC) which is modified to work with negative information. Then, a formalism for factorizing constraints in negative smaller constraints is developed. Factorization is then included in ADC working with negative information. The produced algorithm is called ADC^{-}_{factor}. Additionally an operation that is able to eliminate binary domain variables by factorization is explained. We end by putting factorization in context with the state-of-the-art and we suggest some lines of further development of the work done in the Chapter.

Chapter 6 introduces the filtering operation for CSP. It first starts introducing an idea that lead us to filtering and is it called delayed variable elimination (DVE). It then defines the filtering operation: it is a way of using constraints of some parts of the problem to reduce the size of other constraints. Then filtering is used inside ADC, producing a new algorithm ADC-DVE-F. An experimental evaluation on the N-Queens problem and the Shur-lemma is done.

Chapter 7 is devoted to the extension of filtering into various complete inference algorithms for WCSP. We first extend filtering to Bucket Elimination (which is the direct extension of ADC to WCSP). We then introduce CTE an algorithm which is a generalization of the explained complete inference algorithms. We add filtering to CTE and develop an iterative algorithm called IMCTE. Experimental evaluation is done on MAX-SAT instances and earth satellite management benchmark. The Chapter ends introducing related work and prospects of future work.

In the first Appendix we show how search algorithms using propagation, like

Partial Forward Checking, can be extended to non-binary constraints. In the second Appendix we describe the set of benchmarks used in the experimental sections of every chapter.

Chapter 2

Preliminaries

2.1 CSP

Many combinatorial problems can be naturally expressed as a *Constraint Satisfaction Problem* (CSP). A CSP involves a set of variables, each one taking values in a finite domain. Variables are related by constraints which impose restrictions on the values that variables can take simultaneously. A solution to a CSP is a selection of one value per variable such that all constraints of the problem are satisfied. Formally,

Definition 2.1 **[CSP]** *A CSP is a triple* $\wp = \langle X, D, C \rangle$ *where:*

- $X = \{x_1, ..., x_n\}$ *is a set of* n *variables,*
- $D = \{D_1, ..., D_n\}$ *is a collection of finite domains such that each variable* x_i *takes values in* D_i*,*
- C *is the set of constraints. Each constraint* $c \in C$ *relates (involves) some variables* $var(c) = \{x_{i_1}, ..., x_{i_r}\}$ *and specifies those combinations of values that the variables can simultaneously take. We call* $var(c)$ *the scope of* c *and* $|var(c)|$ *its arity. There are two usual ways to define constraints: as a relation,* c *is defined as a subset of the cartesian product of the domains of the variables in its scope* $D_{i_1} \times ... \times D_{i_r}$*. As a function* c *maps each combination in the cartesian to* $true$ *or* $false$ *depending on wether it is permitted by the constraint or not. Both forms are equivalent and in this thesis we will use them indistinctly. Thus abusing notation,* $t \in c$ *will be equivalent to* $c(t) = true$

An assignment t (we will call it also a tuple) is a set of pairs ⟨variable, value⟩ in which variables are assigned a value of its domain. The variables assigned by t are noted $var(t)$. *Projecting* tuple t over variables $\{x_{i_1}...x_{i_k}\}$, noted $t[x_{i_1}...x_{i_k}]$, returns the tuple where only the pairs of these variables are kept. We define the *concatenation* of two tuples t and t' noted $t \cdot t'$ as the union of its pairs of

assignments and it is only defined if common variables have the same assigned value.

For clarity, we assume that $c(t)$ with $var(c) \subseteq var(t)$ always means $c(t[var(c)])$ so that we select from tuple t the assignments that mention variables of c and ignore the others.

We denote by c_{ij} a constraint involving variables x_i and x_j (namely $var(c_{ij}) = \{x_i, x_j\}$). Similarly c_i denotes the unary constraint on variable x_i. A constraint c that, independently of its arity, does not permit any tuple is denoted by the empty set $c = \emptyset$. $c = \emptyset$ cannot be satisfied.

An assignment is *complete* if it involves all the variables of the problem, otherwise it is *partial*. An assignment is *consistent* if satisfies all the constraints such that all of their variables are assigned, otherwise it is *inconsistent*.

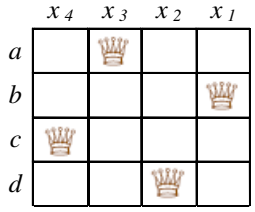
Definition 2.2 **[consistent assignment]** *Consider a CSP $\wp = \langle X, D, C \rangle$ and an assignment t. We say that t is consistent if and only if it is permitted by all constraints $c \in C$ such that $var(c) \subseteq var(t)$, that is:*

$$\texttt{consistent}(t) = \bigwedge_{\forall c \in C \ s.t. \ var(c) \subseteq var(t)} c(t)$$

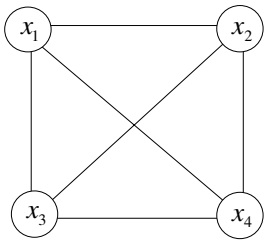
A *solution* to a CSP is a complete consistent assignment. The problem of finding a solution is NP-complete.

Definition 2.3 **[primal constraint graph]** *Given a CSP, its constraint graph is the undirected graph having as set of nodes the variables of the problem. Two nodes are connected if they belong to the scope of at least one constraint of the problem.*

Example 2.1 *A classic example of CSP is the 4-Queens problem. The goal is to place 4 queens in a 4 by 4 chessboard in such a way that they do not attack each other.*



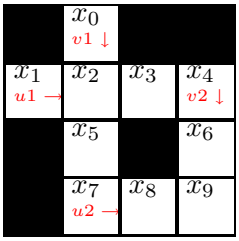
The $n-$Queens has many CSP modelings. One possibility is to associate a variable per column as we know that each queen must be placed in a different column, $X = \{x_1, x_2, x_3, x_4\}$. Each queen has to be placed in a row so each variable takes values in the domain $D_i = \{a, b, c, d\}$. The tuple $t = \{\langle x_1, b\rangle, \langle x_2, d\rangle, \langle x_3, a\rangle, \langle x_4, c\rangle\}$ is a feasible solution. There is a constraint on each pair of variables to express that two queens must not share a row or a diagonal. $C = \{c_{12}, c_{13}, c_{14}, c_{23}, c_{24}, c_{34}\}$ is the set of binary constraints. Find below the constraint graph and the set of constraints associated to the 4-Queens problem,



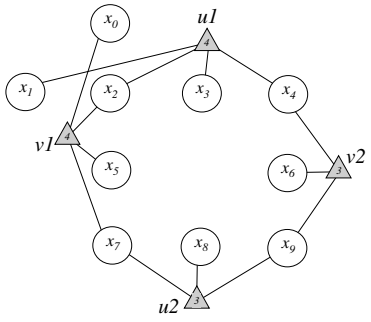
$$c_{12} = c_{23} = c_{34} = \{ac, ad, bd, ca, da, db\}$$
$$c_{13} = c_{24} = \{ab, ad, ba, bc, cb, cd, da, dc\}$$
$$c_{14} = \{ab, ac, ba, bc, bd, ca, cb, cd, db, dc\}$$

Definition 2.4 [**constraint hypergraph**] *Given a CSP, its constraint hypergraph is the hypergraph having as set of nodes the variables of the problem. There is an hyperedge for every constraint of the CSP and connects all nodes that belong to the scope of the constraint. In binary problems the constraint hypergraph is equivalent to the primal constraint graph.*

Example 2.2 *Let us consider an example of CSP inspired in crossword puzzles. Find below the grid of cells and the goal is to assign to each vertical and horizontal slot a word that is a correct written number. The problem can be modeled as a CSP with a variable per cell ($X = \{x_0, ..., x_9\}$) and a constraint per each horizontal and vertical slot ($C = \{u1, u2, v1, v2\}$).*



Each variable takes values in all possible letters so each variable x_i has $|D_i| = 26$ values. Constraints impose slots to take values in valid written numbers. The constraint hyper-graph associated to this problem is shown below. Triangles represent n-ary constraints and are the hyperedges of the hypergraph. For instance, hyperedge $u1$ is associated with the first horizontal slot and links variables $var(u1) = \{x_1, x_2, x_3, x_4\}$.



2.2 CSP Solving Methods

There are two main algorithmic approaches for solving CSP: search and inference.

2.2.1 Search

Search explores the search space defined by all possible assignments looking for a complete consistent assignment. The search space is developed as a tree, called the *search tree*. In Fig. 2.1 we show the search tree of the 4-Queens problem. Each node of the search tree represents an assignment defined by the path from this node to the root. The search tree can be generated with the following procedure assuming a fixed ordering among variables and domain values. Consider the first variable and its d possible values; for each value consider the second variable and its d possible values (it produces d^2 possibilities); if we proceed with the third variable we produce d^3 possibilities; and so on. If we continue this process for the n variables it generates a tree such that its leaves are the set of all possible complete assignments. Clearly the size of the tree is exponential $O(d^n)$).

The search tree can be traversed in many ways. The usual choice is *depth-first* because it has polynomial space complexity and also because the depth of the tree is bounded by the number of variables.

Backtracking

The simplest algorithm to traverse the search tree is *Backtracking* (BT). BT performs a depth-first traversal of the search tree. At each node, BT checks whether the assignment associated with the current node is consistent or not. If it is, BT takes the next variable and sequentially assigns its domain values and the same process is recursively applied. If the current assignment is inconsistent, the detected inconsistency invalidates any assignment in the subtree rooted at the current node and BT backtracks. This step is called backtracking because it involves reconsidering assignments made in previous levels of the tree. In Fig. 2.1 we show a BT tree traversal.

BT algorithm is presented in Fig. 2.2. It has two input parameters: the current assignment t, and the set of the remaining unassigned variables F (called *future variables*). The different components of the CSP instance $\langle X, D, C\rangle$ are accessible as global variables. The algorithm is called initially $\texttt{BT}(\{\}, X)$. BT returns $true$ if a consistent complete assignment exists, $false$ otherwise. Every time BT reaches a leaf of the search tree, the current assignment is a complete assignment (line 1). At an internal node of the search tree, BT chooses a variable and iterates over all its possible values (lines 2,3). The consistency of t is tested (line 4). If t is consistent then we assign another variable calling recursively BT (line 5). If one of the recursive calls finds a solution the current call returns $true$ (line 6). If no value for the variable leads to a solution, it returns $false$ (line 7). In the worst case BT has to traverse all the nodes of the tree, thus it has

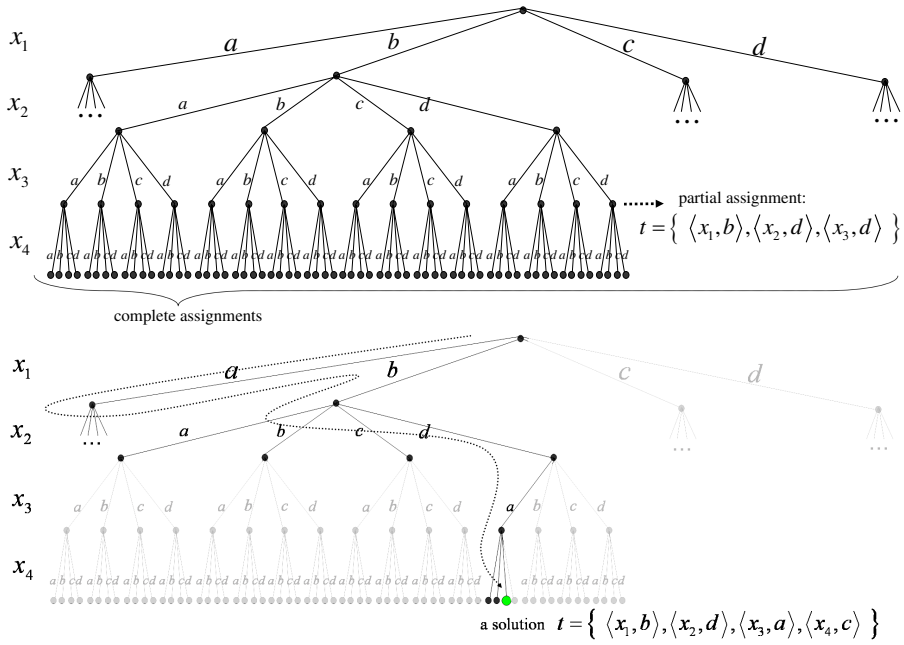


Figure 2.1: Top: The search tree for the 4-Queens problem. Bottom: The BT traversal of this search tree.

exponential time complexity $O(d^n)$, being d the maximum domain of a variable, and n the number of variables. The space complexity of BT is polynomial.

Forward Checking

BT algorithm traverses the search tree checking the consistency of the current assignment. *Forward Checking* (FC) [Harlick, 1980] algorithm adds *look-ahead* to BT. Look-ahead is the process of removing values of the domains of future variables that are not consistent with the current assignment. In a way look-ahead anticipates the detection of branches of the search tree with no solution. Fig. 2.3 presents FC. It has three input parameters: the current assignment t, the set of future variables F and the collection of current domains. It chooses a variable and iterates over its values. After selecting a value it performs look-ahead (line 4). Look-ahead function iterates over all future domains deleting the values that are inconsistent when added to the current assignment and returns the new domains. Now an assigned value can be skipped if after look-ahead an empty domain is detected (line 5). If there is not an empty domain, FC is called recursively with the updated collection of domains (line 6). As BT, FC has to traverse all the nodes of the tree in the worst case, it has exponential time complexity $O(d^n)$ and polynomial space complexity.

```
function BT(t, F)
1   if F = ∅ then return true
2   x_i ← choose-variable(F)
3   for each a ∈ D_i do
4     if consistent(t · ⟨x_i, a⟩) then
5       sol ← BT(t · ⟨x_i, a⟩, F − {x_i})
6       if sol then return true
7   return false
```

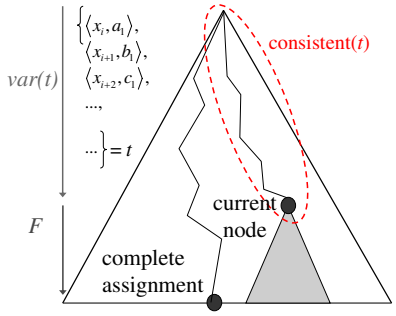


Figure 2.2: Left: Backtracking algorithm. Right: a representation of the tree search.

There are more sophisticated forms of look-ahead. FC look-ahead is only taking into account constraints that link past to future variables. It is possible to prune domains of future variables also taking into account constraints that link exclusively future variables. This is the case of *Maintaining Arc Consistency* (MAC) algorithm [Sabin and Freuder, 1994]. To do so, MAC has an extra computational effort. However MAC is considered the most efficient algorithm on hard problems. There are also more sophisticated ways of backtracking. Instead of reconsidering the last assignment made when an inconsistency is found, it is possible to jump to the last variable that caused the conflict. This strategy is called *Backjumping* (BJ) [Gaschnig, 1978]. *Graph Based* BJ [Dechter, 1990] computes the level of the search where to backtrack based on the constraint graph. *Conflict directed* BJ improves BJ by following a more sophisticated jumping strategy that is based on the conflicts between variables [Prosser, 1993].

2.2.2 Inference

Inference algorithms deduce new constraints from original constraints. These new constraints were implicit in the original problem and are made explicit. When inference can, by itself, obtain all the solutions of the problem it is called *Complete Inference*. Inference can also be incomplete, meaning that it deduces

```
function FC(t, F, D)
1 if F = ∅ then return true
2 x_i ← choose-variable(F)
3 for each a ∈ D_i do
4   D' ← look-ahead(t · ⟨x_i, a⟩, F − {x_i}, D)
5   if ∅ ∉ D' then
6     sol ← FC(t · ⟨x_i, a⟩, F − {x_i}, D')
7     if sol then return true
8 return false
```

```
function look-ahead(t, F, D)
1 for each x_j ∈ F do
2   for each b ∈ D_j do
3     if not consistent(t · ⟨x_j, b⟩) then
4       D_j ← D_j − {b}
5 return D
```

Figure 2.3: Forward Checking algorithm.

implicit constraints but deductions are not sufficient to obtain solutions. In this case it must be combined with search. We define two operations on constraints,

Definition 2.5 [Project Out] *Given a constraint c such that $x_i \in var(c)$, projecting out a variable x_i from c, noted $c \Downarrow x_i$, is a new constraint with scope $var(c) - \{x_i\}$ whose permitted tuples are*

$$c \Downarrow x_i = \{t \mid t \cdot \langle x_i, a \rangle \in c\}$$

Consider a constraint c and a set of variables X. We denote by $c \Downarrow X$ the operation that projects out all variables in X from c one after the other. The result of projecting out a variable from a unary constraint, $c_i \Downarrow x_i$, is a 0-arity constraint. If the original constraint c_i permitted at least one tuple we must be able to express that the resulting 0-arity constraint is not empty. For this purpose we introduce the *empty tuple* that we denote λ.

Example 2.3 *Consider constraint c_{12} of the 4-Queens problem,*

$$c_{12}(x_1, x_2) = \{ac, ad, bd, ca, da, db\}$$

If we project out variable x_1 from c_{12} we obtain a unary constraint c_2 that permits all the values of the domain,

$$c_{12} \Downarrow x_1 = c_2 = \{a, b, c, d\}$$

If we then project out x_2 we obtain,

$$c_2 \Downarrow x_2 = \{\lambda\}$$

Definition 2.6 [Join] *Given two constraints c and c', their join $c \bowtie c'$ is a new relation with scope $var(c) \cup var(c')$ containing all the possible tuples permitted by both constraints,*

$$c \bowtie c' = \{t \cdot t' \mid t \in c, t' \in c'\}$$

Example 2.4 *Find on the left of the drawing below constraints c_{12} and c_{13} of the 4-Queens problem. On the right we show the constraint resulting of their join $c_{123} = c_{12} \bowtie c_{13}$.*

$$c_{12} = \{ac, ad, bd, ca, da, db\}$$

$$c_{13} = \{ab, ad, ba, bc, cb, cd, da, dc\}$$

$$\bowtie \rightarrow c_{123} = \{acb, acd, adb, add, bda, bdc, cab, cad, daa, dac, dba, dbc\}$$

Joining all the constraints of the problem ($\bowtie_{c \in C}\ c$) we obtain a constraint with the whole set of solutions as permitted tuples. Computing this global constraint as a sequence of joins of all the constraints is a brute force approach exponential in time and space with respect to the number of variables of the problem.

Definition 2.7 [**stronger than**] *Consider two constraints c and c'. Let $V = var(c') \cap var(c)$ be their set of common variables. Then c is* stronger than *constraint c', denoted $c' \preccurlyeq c$, if*

$$\{t[V] \mid t \in c'\} \supseteq \{t[V] \mid t \in c\}$$

In words a constraint c is stronger than another constraint c' if, projecting all tuples in its common set of variables, c' permits all the tuples of c and possibly more. We say that a set of constraints C is *stronger than* a set of constraints C' iff $(\bowtie_{c' \in C'}\ c') \preccurlyeq (\bowtie_{c \in C}\ c)$. Note that for any constraint c and a variable $x_i \in var(c)$, $(c \Downarrow x_i) \preccurlyeq c$ and also $c \preccurlyeq (c \Downarrow x_i)$.

Definition 2.8 [**equivalence in the common set of variables**] *Consider two sets of constraints C and C'. C and C' are said to be equivalent in their common set of variables, noted $C \approx C'$ if*

$$C \preccurlyeq C' \text{ and } C' \preccurlyeq C$$

In words two sets of constraints are equivalent if they have the same set of solutions in their common set of variables. Two CSPs $\wp = \langle X, D, C \rangle$ and $\wp' = \langle X', D', C' \rangle$ are said to be equivalent, noted $\wp = \wp'$ if their respective sets of constraints are equivalent in their common set of variables. An inference operation transforms the problem $\wp$ into $\wp'$ by operating with constraints while preserving its equivalence. $\wp'$ it is presumably easier to solve than $\wp$ as more constraints are made explicit. Replacing two constraints c and c' from a CSP $\wp = \langle X, D, C \rangle$ by its join $c \bowtie c'$ is an inference transformation, thus preserves equivalence. It is possible to eliminate a variable from a CSP and obtain an equivalent CSP in the remaining set of variables. This process is called *Variable Elimination* [Bertele and Brioschi, 1972, Dechter, 1999].

Definition 2.9 [**variable elimination**] *The elimination of variable x_i from the set of constraints C is an inference operation that obtains a set of constraints C' that does not mention x_i and is equivalent to C. The new C' can be computed as follows,*

$$C' \leftarrow C - \{c \in C \mid x_i \in var(c)\} \ \cup\ \{(\bowtie_{c \in C \ s.t.\ x_i \in var(c)} c) \Downarrow x_i\}$$

In Fig. 2.4 we show the algorithm associated to variable elimination (function Var-Elim). It receives as input a variable and a set of constraints. The process of eliminating variable x_i from this set of constraint C can be described in three elementary operations : i) gathering of the constraints in which x_i participates

(line 1). This set of constraints is often called *bucket* of the variable. *ii*) join all these constraints and project out x_i (line 2). *iii*) Finally, it substitutes the initial bucket of constraints by the new obtained constraint in the set C (line 3). The obtained set of constraints does not mention x_i and is equivalent to the initial set.

Graph concepts

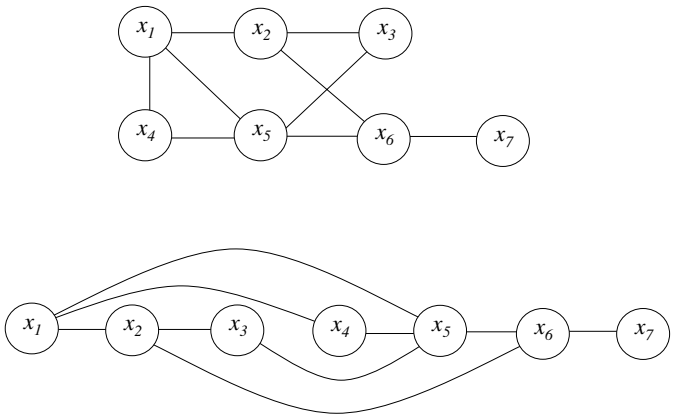
Consider a CSP instance and its constraint graph. Let $O = \{x_{i_1}, ..., x_{i_n}\}$ be an ordering of its variables X. The *induced ordered graph* associated to the constraint graph and the ordering O is obtained as follows: variables are processed from last to first; when variable x_i is processed, all its preceding neighbors are connected.

The *width* of a variable is the number of neighbors that precede the variable in the ordering. The width of an ordering O, denoted $w(O)$, is the maximum width over all variables. The *induced width* of a graph and an ordering O, is the width of the induced ordered graph.

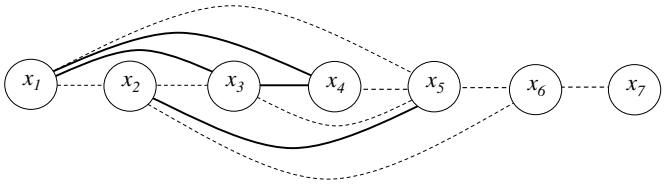
Definition 2.10 [**induced width**] *The induced width, noted w^{opt}, of a constraint graph is the minimal induced width over all its orderings.*

Computing the induced width of a graph is NP-complete. Heuristics exist to compute orders with low induced width.

Example 2.5 *Find below the constraint graph of a CSP and the same constraint graph arranged in the order imposed by its variable indexes.*



Let's now build the induced graph with respect to this order. With respect to the original graph a connection between variables x_2 and x_5 has to be added because they are both preceding neighbors of x_6. x_5 has three preceding neighbors x_4, x_3 and x_1. They have all three to be connected. The corresponding induced graph is shown below. Strong lines are the new added connections in this process.



The induced width of this ordering is 4 as variable x_5 has 4 preceding neighbors.

Adaptive Consistency

Adaptive Consistency (ADC) [Dechter and Pearl, 1987] is the basic Complete Inference algorithm for solving CSP. ADC performs a sequence of problem transformations eliminating one variable at each step and obtaining an equivalent CSP on the remaining set of variables. When there are no variables left, the solution can be trivially obtained. ADC appears in Fig. 2.4. It receives as input a CSP and it returns *true* if it has solution and $false$ otherwise. It first computes an elimination order (line 1). It then performs a sequence of variable eliminations (lines 2 and 3 of ADC). After calling variable elimination ADC checks if there is an empty constraint in the returned set C (line 4). If it is the case it returns $false$ as no solution exists. After eliminating all variables, the empty tuple is the only present in the set of constraints $C = \{\lambda\}$ and we can then return *true*.

If we store all the generated constraints c', before projecting out each variable (line 2), the set of all solutions can be trivially obtained. One possibility is to join all c' constraints in the inverse order of how they where obtained, joining the c' obtained in the first elimination at the end. The resulting constraint of this join is the set of all solutions (see example 2.6). For the sake of clarity we skip in the algorithms this step needed to retrieve the set of all solutions. The interested reader may address [Dechter, 2003].

function `Var-Elim`(x_i, C)
1 $B \leftarrow \{c \in C \mid x_i \in var(c)\}$
2 $c' \leftarrow (\bowtie_{c \in B} c) \Downarrow x_i$
3 $C \leftarrow C \cup \{c'\} - B$
4 **return** C

function `ADC`$(\wp)$
1 $X \leftarrow$ compute-order(X)
2 **for each** $x_i \in X$ **do**
3 $\quad C \leftarrow$ `Var-Elim`(x_i, C)
4 $\quad$ **if** $\emptyset \in C$ **then return** $false$
5 **return** *true*

Figure 2.4: Adaptive Consistency pseudo-code.

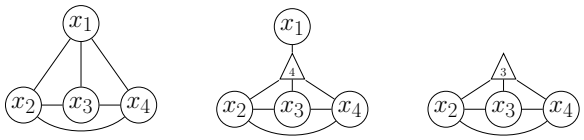
Example 2.6 *Consider the 4-Queens problem of example 2.1. Its constraint graph is a clique thus its induced width is $w^{opt} = n - 1 = 3$. We eliminate variables in the ordering imposed by variable indexes. ADC starts with the elimination of variable x_1. It first joins all constraints linked to x_1, which are $B = \{c_{12}, c_{13}, c_{14}\}$. Let's recall that ,*

$$c_{12} = \{ac, ad, bd, ca, da, db\}$$
$$c_{13} = \{ab, ad, ba, bc, cb, cd, da, dc\}$$
$$c_{14} = \{ab, ac, ba, bc, bd, ca, cb, cd, db, dc\}$$

Joining all constraints in bucket, we obtain (line 2 of Var-Elim in Fig. 2.4),

$$c''_{1234} = \begin{array}{l}\{acbb, acbc, acdb, acdc, adbb, adbc, addb, addc, bdaa, bdac,\\ bdad, bdca, bdcc, bdcd, caba, cabb, cabd, cada, cadb, cadd,\\ daab, daac, dacb, dacc, dbab, dbac, dbcb, dbcc\}\end{array}$$

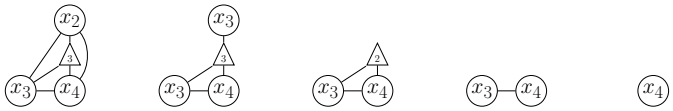
Find below on the left the original constraint graph with x_1 on top. The second constraint graph shows the result after joining all constraints linked to x_1, that is c''_{1234}. ADC finally projects out x_1 from $c''_{1234} \Downarrow x_1 = c'_{234}$ (last constraint graph) and the variable is eliminated.



ADC continues with the elimination of variable x_2 by joining all the constraints it is linked to; $B = \{c'_{234}, c_{23}, c_{24}\}$. It obtains,

$$c''_{234} = \begin{array}{l}\{dbc, daa, dac,\\ adb, add, acb\}\end{array}$$

Find on the drawing below the last constraint graph rearranged with x_2 on top. The second constraint graph shows c''_{234}. ADC then projects out x_2 and obtains c'_{34} (shown in the third constraint graph below). For the x_3 elimination we join the two binary constraints $B = \{c'_{34}, c_{34}\}$ and obtain $c''_{34} = \{ac, db\}$. When projecting out x_3 we obtain $c'_4 = \{c, b\}$.



Only one variable is left. Now $B = \{c'_4\}$. Then x_4 is projected out and ADC obtains the empty tuple $c'_4 \Downarrow x_4 = \{\lambda\}$, meaning that a solution exists.

To recover all possible solutions ADC proceeds backwards joining the obtained c'' constraints on each elimination: $c''_4 \bowtie c''_{34} = \{ac, db\} = c''$, then $c'' \bowtie c''_{234} = \{dac, adb\} = c'''$ and finally $c''' \bowtie c''_{1234} = \{bdac, cadb\}$ which is the set of all possible solutions.

ADC has exponential time and space complexity with respect to the induced width $O(exp(w))$ of the elimination order.

2.3 WCSP

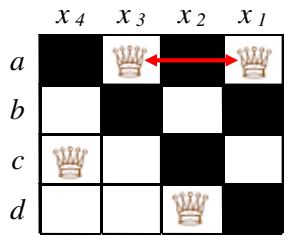
Some CSP instances do not have any solution. In this case we may want to find the best assignment close to a solution among all assignments with respect to some preference criterion. A CSP instance may also have many solutions. Similarly it would be also useful in this case to look for the best solution with respect to a preference criteria. In both cases the CSP model is not sufficient because a CSP constraint completely accepts or forbids a tuple, constraints are then called *hard* for this reason. The CSP model is augmented with the introduction of the so called *soft* constraints. The introduction of soft constraints has originated different class of models, see for example [Bistarelli et al., 1995], [Schiex et al., 1995]. For our purposes we choose as representive of this class of models the *Weighted* CSP framework (WCSP) [Larrosa and Schiex, 2004]. In WCSP a constraint assigns a cost to each tuple. To maintain coherence with CSP model we consider inconsistent an assignment with cost ∞. Constraints are now functions which assign a cost in $\mathbb{N} \cup \infty$ to every tuple. Formally,

Definition 2.11 [WCSP] *A WCSP is triple $\wp = \langle X, D, C \rangle$ where X and D are variables and domains as in CSP. Constraints in the set C are now cost functions. Each function $f \in C$ involves a number of variables $var(f) = \{x_{i_1}, \ldots, x_{i_r}\}$ and assigns costs to tuples in the cartesian product $\prod_{i \in var(f)} D_i$. Assigned costs are non negative integers in the interval $[0..\infty]$. The meaning of costs is:*

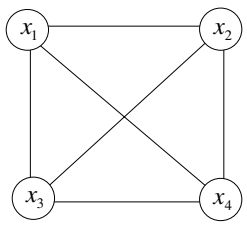
$$f(t) = \begin{cases} 0 & \text{if } t \text{ is completely allowed} \\ u \in \mathbf{N}^+ & \text{if } t \text{ is partially allowed} \\ \infty & \text{if } t \text{ is totally forbidden} \end{cases}$$

A solution to a WCSP is a complete consistent assignment with cost $< \infty$. An optimal solution is a solution with minimum cost. Finding an optimal solution to a WCSP is NP-hard. As a consequence of this fact, if the optimal cost is unknown, given a complete assignment t it is not possible to test if it is an optimal solution or not in polynomial time.

Example 2.7 *We convert the 4-Queens problem into an optimization problem that we call the 4-WQueens. The task is to place one queen per column in a board in such a way that the number of mutual attacks is minimized. Some cells are forbidden. Find on the drawing below the board where black cells denote forbidden cells and a solution of cost* 1 *is shown. An arrow indicates the only attack.*



Black cells are forbidden so functions assign cost ∞ to tuples that contain them. Find below the constraint graph and the set of functions associated to the 4-WQueens problem.



$$f_{12} = \begin{cases} 0 & ad \\ 1 & ab, cb, cd \\ \infty & b\_, d\_, \_a, \_c \end{cases} \qquad f_{13} = \begin{cases} 0 & ad, cd \\ 1 & aa, ac, cc, ca \\ \infty & b\_, d\_, \_b \end{cases}$$

$$f_{23} = \begin{cases} 0 & bd, da \\ 1 & ba, bc, dc, dd \\ \infty & a\_, c\_, \_b \end{cases} \qquad f_{24} = \begin{cases} 0 & bc, dc \\ 1 & bb, bd, db, dd \\ \infty & a\_, c\_, \_a \end{cases}$$

$$f_{34} = \begin{cases} 0 & ac, ad, db \\ 1 & ab, cb, cc, cd \\ & dc, dd \\ \infty & b\_, \_a \end{cases} \qquad f_{14} = \begin{cases} 0 & ab, ac, cb, cd \\ 1 & ad, cc \\ \infty & b\_, d\_, \_a \end{cases}$$

A mutual attack has cost one. The tuple $t = \{\langle x_1, a\rangle, \langle x_2, d\rangle, \langle x_3, a\rangle, \langle x_4, c\rangle\}$ is an optimal solution of cost 1.

2.4 WCSP Solving Methods

Analogously to CSP there exist two main algorithmic approaches for solving WCSP: search and inference.

2.4.1 Search

For clarity reasons, search algorithms for WCSP usually make the assumption of dealing exclusively with *binary* functions, that is, functions that only have two variables in their scope [Freuder and Wallace, 1992, Larrosa et al., 1999, Larrosa and Schiex, 2004]. Generalization of certain WCSP search algorithms to n-ary functions have been studied, see for example [Meseguer, 2000, Meseguer et al., 2001]. The latter is summarized in Appendix B. The binary assumption is a consequence of the difficulty of dealing with explicit higher arity functions, as the number of tuples to store grows exponentially with the arity of the function. However other approaches exist that deal with implicit functions. [Regin et al., 2000a] explores the idea of introducing meta-constraints for counting the number of unsatisfied hard constraints. The idea of implicit soft constraints first appears in [Petit et al., 2001]. It deals with implicit constraints and the cost of each tuple is obtained via a computing procedure. Our choice in all this thesis is to work exclusively with explicit constraints and functions, the reason being its generality and abstraction. The counterpart of this choice is that we loose the possibility of exploiting the semantics of the constraint. The semantic of every specific pre-defined type of constraint can be exploited to derive filtering algorithms that can be used during search to prune the domain of the variables involved in the constraint. Filtering algorithms that have been studied for exploiting the semantics of the *all-different* constraint are one of the most famous examples of this kind [Régin, 1994].

Branch and Bound

The *Branch and Bound* (BB) algorithm traverses the search tree depth-first. At a node of the tree, the current assignment is t and $var(t)$ is the set of assigned variables, we call them *past* variables. Unassigned variables are called *future* and noted F.

BB computes at every node of the search tree an underestimation of the cost of any leaf node descendent from the current node, the *lower bound* at that node (`lb`). The *upper bound* (`ub`) is the cost of the best complete assignment found so far. When $\texttt{lb} \geq \texttt{ub}$ we know that the current best solution cannot be improved below the current node, so we prune the subtree rooted at the current node. Thus $\texttt{lb} \geq \texttt{ub}$ is the pruning condition. In that case, BB performs backtracking as in classical CSP. In its simplest version BB computes as lower bound the sum of costs returned by completely instantiated constraints, that is all constraints that link past variables. It is the cheapest lower bound one can compute.

In Fig. 2.5 we show the search tree with some additional concepts from BB. The BB lower bound expression, $LB^{BB}(t)$, is also called $\texttt{cost}(t)$.

Definition 2.12 [BB lower bound] *Consider a search state such that t is the current assignment (namely var(t) is the set of past variables) the lower bound that BB computes is:*

$$LB^{BB}(t) = \texttt{cost}(t) = \sum_{f \in C, var(f) \subseteq var(t)} f(t)$$

function $\texttt{BB}(t, F, \texttt{ub})$
1 **if** $F = \emptyset$ **then return** $\texttt{cost}(t)$
2 $x_i \leftarrow \texttt{choose-variable}(F)$
3 **for each** $a \in D_i$ **do**
4 **if** $\texttt{LB}^{\texttt{BB}}(t \cup \langle x_i, a \rangle) < \texttt{ub}$ **then**
5 $\texttt{ub}' \leftarrow \texttt{BB}(t \cup \langle x_i, a \rangle, F - \{x_i\}, \texttt{ub})$
6 **if** $\texttt{ub}' < \texttt{ub}$ **then** $\texttt{ub} \leftarrow \texttt{ub}'$
7 **return** ub

function $\texttt{LB}^{\texttt{BB}}(t)$
1 **return** $\sum_{f \in C, var(f) \subseteq var(t)} f(t)$

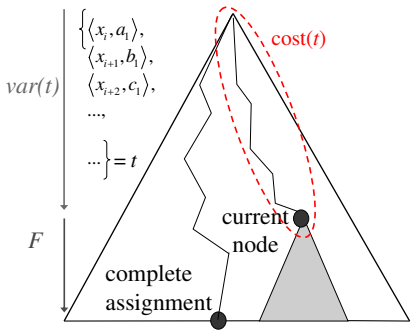


Figure 2.5: Left: Branch and Bound algorithm. Right: a representation of the tree search.

BB is presented in Fig. 2.5. It has three input parameters: the current assignment t, the set of future variables F, and the upper bound `ub`. The different components of the WCSP instance $\langle X, D, C \rangle$ are accessible as global variables. The algorithm is called initially $\texttt{BB}(\{\}, X, \infty)$. BB returns the optimal cost of the problem if it was initially called with an upper bound greater than

the optimal cost. If the initial upper bound `ub` cannot be improved the same value of `ub` is returned, meaning that it is a lower bound of the cost of solving the problem. Every time BB reaches a leaf of the search tree, the current assignment is a complete assignment (line 1) and its cost is returned [1]. At an internal node of the search tree, BB chooses a variable and iterates over all its possible values (lines 2,3). The corresponding lower bound is computed after each assignment and the pruning condition is tested (line 4). If the cost of past variables does not reach the upper bound it assign another variable calling recursively BB (line 5). After a BB call returns, it tests if the solution has been improved in the nodes below (line 6). Finally the best `ub` found is returned.

Partial Forward Checking

The *Partial Forward Checking* (PFC) [Freuder and Wallace, 1992] enhances BB lower bound by taking into account the costs that will certainly occur because of constraints that link past variables with future variables. The cost of these constraints can be handled with the so-called *inconsistency costs*.

Definition 2.13 [inconsistency cost] *Given a WCSP $\mathcal{W} = \langle X, D, C \rangle$, the current assignment t and the set of future variables F, the inconsistency cost of future variable x_j and value b is:*

$$ic_{jb} = \sum_{f_{ij} \in C \mid \langle x_i, a \rangle \in t} f_{ij}(a, b)$$

In words, the inconsistency cost ic_{jb} contains the cost of assigning value b caused by constraints that link variable x_j with past variables. Observe that $\min_{b \in D_j}\{ic_{jb}\}$ is the cost that at least will be payed when assigning future variable x_j independently of the value that it will be assigned to x_j. Then, the sum $\sum_{x_j \in F} \min_{b \in D_j}\{ic_{jb}\}$ is a lower bound of cost that will necessarily occur if the current partial assignment is extended into a total one. This term can be added to the cost of past variables to compute the lower bound of the current partial assignment, because both terms record different costs from different constraints. The PFC lower bound is defined as follows,

Definition 2.14 [PFC lower bound] *Consider a search state such that t is the current assignment. The lower bound that PFC computes is:*

$$LB^{PFC}(t, F) = \texttt{cost}(t) + \sum_{x_j \in F} \min_{b \in D_j}\{ic_{jb}\}$$

This lower bound can be specialized to a certain value to test if it can not belong to the optimal solution. In that case it can be pruned. The specialization for value b of variable x_j is done by substituting the minimum ic_{jb} of variable x_j by the particular ic_{jb} as follows,

[1] At this point the current assignment t should be stored as the best solution found so far although we omit it for the sake of code simplicity.

$$LB_{jb}^{PFC}(t,F) = \texttt{cost}(t) + ic_{jb} + \sum_{x_k \in F, k \neq j} \min_{c \in D_k} \{ic_{kc}\}$$

PFC appears in Fig. 2.6. With respect to BB we add the function look-ahead which is in charge of updating the inconsistency costs and function prune which deletes values of future variables that have a PFC lower bound greater than the current upper bound. Prune returns the new collection of domains. As values can be deleted PFC also has as input parameter the current domains D. The algorithm is called initially PFC($\{\}, X, D, \infty$). In line 4 of PFC look-ahead is called. Function look-ahead updates inconsistency costs ic_{jb} of every value b of future variable j by adding the cost of the binary constraint f_{ij} to it. Function prune calls the specialized PFC lower bound for every value of every future variable. Prune may produce an empty domain. The ics have to be restored to its previous values. This operation is called context restoration and it is omitted for simplicity.

function PFC($t, F, D,$ ub)
1 **if** $F = \emptyset$ **then return** cost(t)
2 $x_i \leftarrow$ choose-variable(F)
3 **for each** $a \in D_i$ **do**
4 look-ahead($i, a, t, F, D,$ ub)
5 **if** $\text{LB}^{\text{PFC}}(t \cdot \langle x_i, a \rangle, F, D') <$ ub **then**
6 $D' \leftarrow$ prune($i, a, t, F, D,$ ub)
7 **if** $\emptyset \notin D'$ **then**
8 ub$' \leftarrow$ PFC($t \cdot \langle x_i, a \rangle, F - \{x_i\}, D',$ ub)
9 **if** ub$' <$ ub **then** ub $\leftarrow$ ub$'$
10 **return** ub

function $\text{LB}^{\text{PFC}}(t, F, D)$
1 **return** cost(t) $+ \sum_{x_j \in F} \min_{b \in D_j} \{ic_{jb}\}$

function $\text{LB}_{jb}^{\text{PFC}}(t, F, D)$
1 **return** cost(t) $+ ic_{jb} + \sum_{x_k \in F, k \neq j} \min_{c \in D_k} \{ic_{kc}\}$

function look-ahead($i, a, t, F, D,$ ub)
1 **for each** $x_j \in F - \{x_i\}$ **do**
2 **for each** $b \in D_j$ **do** $ic_{jb} \leftarrow ic_{jb} + f_{ij}(a, b)$

function prune($i, a, t, F, D,$ ub)
1 **for each** $x_j \in F - \{x_i\}$ **do**
2 **for each** $b \in D_j$ **do**
3 **if** $\text{LB}_{jb}^{\text{PFC}}(t \cdot \langle x_i, a \rangle, F, D) \geq$ ub **then**
4 $D_j \leftarrow D_j - \{b\}$
5 **return** D

Figure 2.6: Partial Forward Checking algorithm.

There are forms of more sophisticated look-ahead for WCSP algorithms. Analogously to FC (see Section 2.2.1), PFC only takes into account the functions that link past to future variables. Chapter 3 presents more advanced techniques to take into account constraints that link exclusively future variables.

Alternative ways of doing so are also pointed out in Section 3.6 and also Section B.1.

2.4.2 Inference

Inference CSP algorithms (see section 2.2.2) can be generalized to WCSP. The generalization is obtained by substituting project out and join operations from CSP with their generalization to WCSP.

Definition 2.15 [Project out by minimization]. *Given a function f, projecting out variable $x_i \in var(f)$, denoted $f \Downarrow x_i$, is a new function with scope $var(f) - \{x_i\}$ such that,*

$$(f \Downarrow x)(t) = \min_{a \in D_i} \{f(\langle x_i, a\rangle \cdot t)\}$$

In words the cost of a tuple in $f \Downarrow x_i$ is the minimum among the costs of all possible extensions of x_i. Projecting out the variable of a unary function produces a constant. Any constant can be considered an empty scope function.

Definition 2.16 [Sum] *Given two functions f and g, its sum $f + g$ is a new function with scope $var(f) \cup var(g)$ such that,*

$$(f + g)(t) = f(t) + g(t)$$

Example 2.8 *Find on the left of the drawing below functions f_{12} and f_{13} of the 4-WQueens. On the right we show the constraint resulting of its sum $f_{123} = f_{12} + f_{13}$.*

$$f_{12} = \begin{cases} 0 & ad \\ 1 & ab, cb, cd \\ \infty & b\_, d\_, \_a, \_c \end{cases} \qquad f_{13} = \begin{cases} 0 & ad, cd \\ 1 & aa, ac, cc, ca \\ \infty & b\_, d\_, \_b \end{cases} \quad + \rightarrow \quad f_{123} = \begin{cases} 0 & add \\ 1 & abd, ada, cbd \\ 2 & adc, cda, cdd, aba, \\ & cba, abc, cdc, cbc \\ \infty & b\_\_, d\_\_, \_a\_, \_c\_, \_\_b \end{cases}$$

Summing all functions of the problem, $\sum_{f \in C} f$, we obtain a function involving all variables of the problem with the whole set of solutions as consistent tuples with their associated cost.

We now extend the relation *stronger than* defined for CSP in Def. 2.7 to WCSP. In this context we call it lower bound function.

Definition 2.17 [lower bound function] *Consider two functions g and f. Let $V = var(g) \cap var(f)$ be their set of common variables. Let $g' = g \Downarrow (var(g) - V)$ a functions where we have projected out all variables that don't appear in the scope of f. Let $f' = f \Downarrow (var(f) - V)$. Then g is* a lower bound functions *of f, denoted $g \preceq f$, if*

$$g'(t) \leq f'(t) \quad \forall t \in \prod_{i|x_i \in V} D_i$$

We say that a set of functions G is a lower bound of a set of functions F if $(\sum_{g \in G} g) \preceq (\sum_{f \in F} f)$. Note that for any function f and a variable $x_i \in var(f)$, $(f \Downarrow x_i) \preceq f$, $\sum_{f \in F}(f \Downarrow x_i) \preceq (\sum_{f \in F} f) \Downarrow x_i$ holds.

Analogously to CSP, two sets of functions C and C' are equivalent in their common set of variables, noted $C \approx C'$ if $C \preceq C'$ and $C' \preceq C$. Two WCSP are equivalent if they have the same set of solutions with equal cost in their common set of variables. Two WCSPs $\mathcal{W} = \langle X, D, C \rangle$ and $\mathcal{W}' = \langle X', D', C' \rangle$ are said to be equivalent, noted $\mathcal{W} = \mathcal{W}'$, if $C \approx C'$. Variable elimination can be extended to WCSP.

Definition 2.18 [variable elimination] *Let $\mathcal{W} = \langle X, D, C \rangle$ be a WCSP. The elimination of variable x_i from the set of functions C is an inference operation that obtains a set of functions C' that does not mention x_i and is equivalent to C. The new C' can be computed as follows,*

$$C' \leftarrow C - \{f \in C \mid x_i \in var(f)\} \ \cup \ \{(\sum_{f \in C \ s.t. \ x_i \in var(f)} f) \Downarrow x_i\}$$

In Fig. 2.7 we show the algorithm associated to variable elimination (function Var-Elim) for WCSP. It receives as input a variable and a set of constraints. The process of eliminating variable x_i from this set of constraint C can be depicted in three elementary operations: (i) gathering of the constraints in which x_i participates, also called the *bucket* of x_i (line 1). (ii) sum of all constraints in the bucket and projection out of x_i from the obtained constraint (line 2). (iii) then the constraint in the original bucket must be replaced with thew obtained constraint in C (line 3).

Bucket Elimination

Bucket Elimination (BE) [Dechter, 1999] is the basic Complete Inference algorithm for solving WCSP. BE appears in Fig. 2.7. Similarly to ADC BE first computes an elimination order which is intended to have low induced width (line 1). Then BE performs a sequence of problem transformations eliminating a variable at each step and obtaining an equivalent WCSP (line 2,3). When all variables have been eliminated if the remaining set of constraints contains the infinite cost constant means the the problem has no solution. Thus it returns $C \neq \{\infty\}$ (line 4).

Example 2.9 *Consider the 4-WQueens problem introduced in example 2.7. Its constraint graph is a clique so its induced width is $w^{opt} = 3$ We eliminate variables in the order $O = \{x_1, x_2, x_3, x_4\}$. BE starts with the elimination of variable x_1. It first sums all functions linked to x_1, which are $B = \{f_{12}, f_{13}, f_{14}\}$ and projects out variable x_1. It obtains,*

function `Var-Elim`(x_i, C)
1 $B \leftarrow \{f \in C \mid x_i \in var(f)\}$
2 $f' \leftarrow (\sum_{f \in B} f) \Downarrow x_i$
3 $C \leftarrow C \cup \{f'\} - B$
4 **return** C

function `BE`$(\mathcal{W})$
1 $X \leftarrow$ compute-order(X)
2 **for each** $x_i \in X$ **do**
3 $\quad C \leftarrow$ `Var-Elim`(x_i, C)
4 **return** $C \neq \{\infty\}$

Figure 2.7: Bucket Elimination algorithm.

$$f'_{234} = \begin{cases} 0 & ddb, ddc \\ 1 & bdb, bdc, bdd, dab, dac, dcb, dcc, ddd \\ 2 & bab, bac, bad, bcb, bcc, bcd, dad, dcd \\ \infty & a\_\_, c\_\_, \_b\_, \_\_a \end{cases}$$

BE then proceeds eliminating x_2. Now $B = \{f'_{234}, f_{23}, f_{24}\}$. It sums functions in B (left function shown below) and projects out x_2 (right function shown below). Obtains,

$$f'_{234} = \begin{cases} 1 & ddc, bdc, dac, \\ 2 & ddb, bdb, bdd, dab, dcc, \\ 3 & dcb, ddd, bac, bcc, dad \\ 4 & bab, bad, bcb, bcd, dcd \\ \infty & a\_\_, c\_\_, \_b\_, \_\_a \end{cases} \qquad f'_{234} \Downarrow x_2 = f'_{34} = \begin{cases} 1 & dc, ac \\ 2 & db, dd, ab, cc \\ 3 & cb, ad \\ 4 & cd \\ \infty & b\_, \_a \end{cases}$$

It continues with the elimination of x_3. Now $B = \{f'_{34}, f_{34}\}$. It sums functions in B (left function shown below) and projects out x_3 (right function shown below).

$$f'_{34} = \begin{cases} 1 & ac \\ 2 & db, dc \\ 3 & dd, ab, cc, ad \\ 4 & cb \\ 5 & cd \\ \infty & b\_, \_a \end{cases} \qquad f'_{34} \Downarrow x_3 = f''_4 = \begin{cases} 1 & c \\ 2 & b \\ 3 & d \\ \infty & a \end{cases}$$

There is one variable left. Now $B = \{f'_4\}$. The final projection returns a constant that is the optimal cost of the problem, $f'_4 \Downarrow x_4 = 1$.

To recover the solutions one must join all the obtained f' functions starting by the last obtained. From f'_4 we only keep tuples with cost one which is the optimal cost, then $f'_4 + f'_{34} = \{ac\} = f''$. We continue by summing $f'' + f'_{234} = \{dac\} = f'''$. And finally $f''' + f'_{1234} = \{adac\}$, the single optimal solution.

Part I

Systematic Search

Chapter 3

Russian Doll Search

Depth First Branch and Bound (BB) is the basic search algorithm for solving combinatorial optimization problems such as WCSP. BB assigns variables one after the other until it detects that it cannot improve the best solution found so far. Then it backtracks reconsidering systematically all the previous variable assignments. BB has an exponential time complexity because, in the worst case, has to visit all the nodes of the search tree. It has a polynomial space complexity because only explores one particular branch of the tree at the same time. An assignment can be discarded in a particular moment of the search if its under estimated solution cost (we call it lower bound) reaches the cost of the best solution found so far (we call it upper bound). In its simplest version BB computes the simplest lower bound: the sum of costs returned by the constraints that all its variables are assigned so far. We call past variables those variables already assigned. So the lower bound that BB computes only takes into account the contribution of past variables. The practical applicability of BB methods mainly relies on discarding assignments as soon as possible and this happens when the lower bound reaches the upper bound. So the lower bound computation made at each node of the search is a fundamental issue. BB has been enhanced with more accurate lower bounds, which improve solving time on average by pruning branches earlier. As the lower bound is an approximation of the unknown optimal, by accurate we mean a better approximation, closest to the optimum. A lower bound should be as accurate and as cheap to compute as possible.

Partial Forward Checking (PFC) [Freuder and Wallace, 1992], improves BB lower bound by taking into account the costs that will certainly occur because of the constraints that link past variables with variables that haven't been assigned yet, we call them future variables. So PFC lower bound also takes into account the contribution of constraints that link past with future variables.

Russian Doll Search (RDS) [Verfaillie et al., 1996] enhances PFC lower bound by adding also a contribution of the constraints that link only future variables. RDS decomposes the resolution in a sequence of resolutions of subproblems starting from the subproblem containing one single variable and adding

a variable at each time until the whole problem is solved. Each subproblem is solved using information from the smaller subproblems and its optimal cost is reused for larger subproblems resolutions. The main idea is that at an arbitrary search state of an arbitrary subproblem the optimal cost of subproblems that we already solved can be added to the lower bound as a contribution of future variables. This idea recalls *Dynamic Programming* in the sense that it reuses previous computed information for successive resolutions.

RDS only uses the optimal cost of each subproblem in the resolution of larger subproblems. In this Chapter we show that it is possible to extract more information of each subproblem. First, we compute the optimal cost of including every value of the new included variable and reuse it in other larger subproblems (we call the resulting algorithm *Specialized* RDS, SRDS). As subproblems size grow, we loose information of values of the first included variables, so we decide to compute the optimal cost of every value for every subproblem. We call this novel idea *Full Specialized* RDS (FSRDS). FSRDS may spend too much time solving subproblems so we propose a version called *Opportunistic* RDS (ORDS) that optimally solves the problem for every included value only if it is found promising (namely, when there is a possibility of increasing the lower bound). [Meseguer and Sanchez, 2001, Meseguer et al., 2002]

Figure 3.1: Left: Consider a problem instance with 6 variables, this is the sequence of subproblems that RDS solves from the smallest one (with one variable) to the complete one with 6 variables. Right: Analogy with the real Russian dolls.

3.1 Preliminaries

Branch and Bound

Branch and Bound (BB) was presented in Section 2.4.1. BB computes at every node of the search tree an underestimation of the cost of any leaf node descendent form the current node, the *lower bound* at that node (`lb`).

Definition 3.1 [BB lower bound] *Consider a search state such that t is the current assignment (namely var(t) is the set of past variables) the lower bound that BB computes is:*

$$LB^{BB}(t) = \texttt{cost}(t) = \sum_{f \in C, var(f) \subseteq var(t)} f(t)$$

Partial Forward Checking

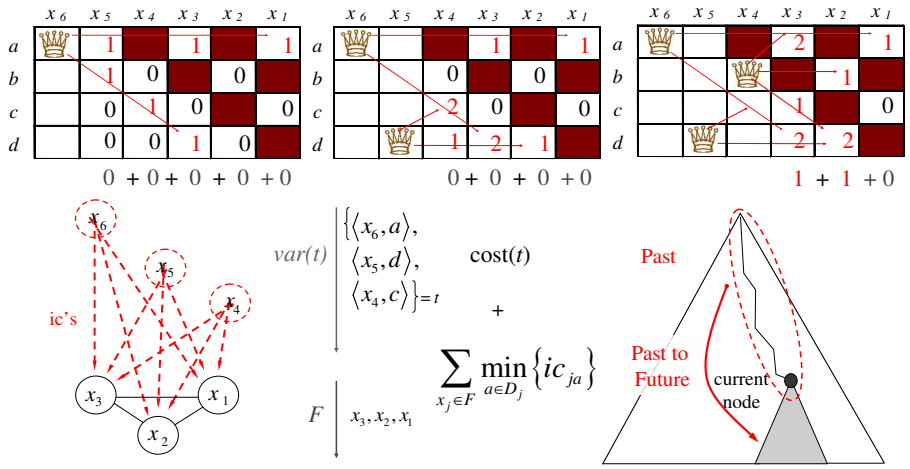
The *Partial Forward Checking* (PFC) was presented in Section 2.4.1. PFC enhances BB lower bound by taking into account the costs that will certainly occur because of constraints that link past variables with future variables. The cost of these constraints can be handled with the *inconsistency costs*.

The PFC lower bound 2.14 adds the sum of minimum inconsistency cost of every variable to the cost of past variables to compute the under estimated cost of the current partial assignment, see Definition 2.14.

This lower bound can be specialized to a certain value to test if it does not belong to the optimal solution and can be pruned. The specialization for value b of variable x_j is done by substituting the minimum ic_{jb} of variable x_j by the particular ic_{jb} as follows,

$$LB^{PFC}_{jb}(t,F) = \texttt{cost}(t) + ic_{jb} + \sum_{x_k \in F, k \neq j} \min_{c \in D_k} \{ic_{kc}\}$$

Example 3.1 *Consider the 6-WQueens problem introduced in example 2.7 with two more variables of domain $D_i = \{a,b,c,d\}$. Find below three tables of inconsistency costs that PFC computes after three assignments. In each cell $\langle x_i, a\rangle$ we show the inconsistency cost for that value ic_{ia}. Below each table we show the minimum ic for each column. PFC lower bound is the sum of the minimum of all columns. First table: $LB^{PFC}(\{\langle x_6,a\rangle\},\{x_5,x_4,x_3,x_2,x_1\}) = 0$. Second table: $LB^{PFC}(\{\langle x_6,a\rangle\,\langle x_5,d\rangle\},\{x_4,x_3,x_2,x_1\}) = 0$. Third table: $LB^{PFC}(\{\langle x_6,a\rangle\,\langle x_5,d\rangle,\langle x_4,b\rangle\},\{x_3,x_2,x_1\}) = 2$.*



Above we show the situation after these three assignments x_6, x_5, x_4. Dashed constraints in the constraint graph are the ones that have been propagated as inconsistency costs. The current assignment t is shown in the middle and PFC lower bound is depicted in two: the past (cost(t)) and past to future (the sum of minimum ic's) contributions.

3.1.1 Russian Doll Search

Russian Doll Search (RDS) enhances PFC by adding to the lower bound some cost that will certainly occur in constraints that link future variables. The idea consists in performing n subproblem resolutions (n is the total number of variables).

Definition 3.2 [subproblem] *Let $\mathcal{W} = \langle X, D, C\rangle$ be a WCSP and $\{x_1, x_2, ..., x_n\}$ a static variable ordering of its variables. $\mathcal{W}^i = \langle X^i, D^i, C^i\rangle$ is the subproblem having $X^i = \{x_i, ..., x_1\}$ as set of variables, $D^i = \{D_i, ..., D_1\}$ as set of domains and $C^i = \{f | f \in C,\ var(f) \subseteq X^i\}$ as set of constraints.*

RDS solves the sequence of subproblems $\mathcal{W}^1, \mathcal{W}^2, ..., \mathcal{W}^n$. Each subproblem is equal to the previous one plus one more variable. Each resolution returns the optimal cost of the corresponding subproblem.

Definition 3.3 *The optimal cost of subproblem $\mathcal{W}^i$ is noted rds_i.*

When RDS solves subproblem $\mathcal{W}^{i+1}$ all $\{rds_1, ..., rds_i\}$ are known. RDS assigns variables in the inverse order of their inclusion. Then, if we just assigned x_j, rds_{j-1} can be added to the lower bound. Remember that PFC lower bound sums the cost coming from constraints that link past variables with constraints that link past and future variables. The rds_{j-1} cost comes exclusively from constraints that link future variables, so no cost is counted twice.

Definition 3.4 [RDS lower bound] *Consider a search state such that t is the current assignment and $var(t) = \{x_n, ..., x_{i+1}\}$. The lower bound that RDS computes is:*

$$LB^{RDS}(t, F = \{x_i, ..., x_1\}) = \texttt{cost}(t) + \sum_{x_j \in F} \min_{b \in D_j} \{ic_{jb}\} + rds_i$$

RDS lower bound can be specialized for a particular value b of variable x_j in the same way as we did for PFC:

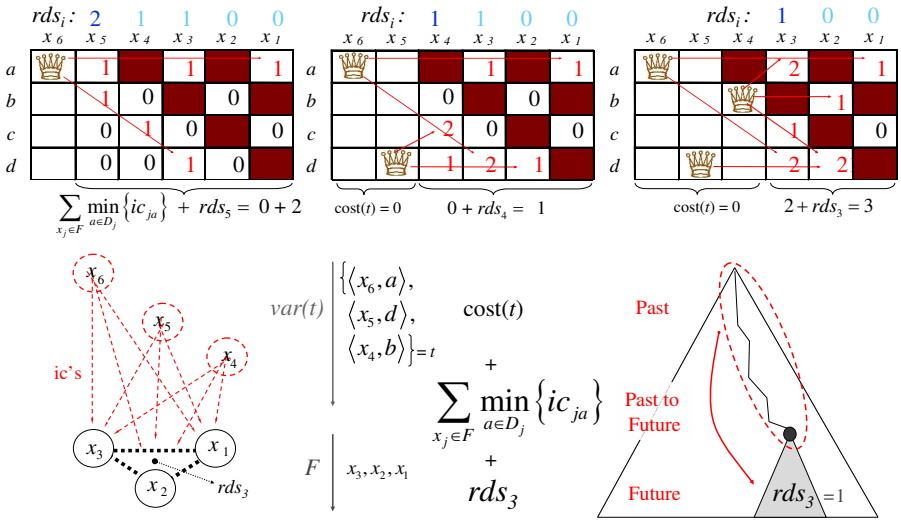
$$LB^{RDS}_{jb}(t, F = \{x_i, ..., x_1\}) = \texttt{cost}(t) + ic_{jb} + \sum_{x_k \in F, k \neq j} \min_{c \in D_k} \{ic_{kc}\} + rds_i$$

Property 3.1 *rds_i increases monotonically, that is $rds_{i+1} \geq rds_i$.*

Proof. The subproblem $\mathcal{W}^{i+1}$ may have more constraints than $\mathcal{W}^i$ because the constraints linking x_{i+1} and variables in $x_i, ..., x_1$ have been added. Constraints only assign positive costs. Thus the optimal cost of solving $\mathcal{W}^{i+1}$ is the same or greater. □

Example 3.2 *Find in the drawing below the three tables of inconsistency costs of example 3.1. Subproblems $\mathcal{W}^1, ..., \mathcal{W}^5$ have been already solved obtaining the corresponding $rds_1, ..., rds_5$ sequence shown on top of the tables.*

We are currently in the last resolution of $\mathcal{W}^6$. The RDS lower bounds computed after the three consecutive assignments are: $LB(\{\langle x_6, a\rangle\}, \{x_5, ..., x_1\}) = 0 + 0 + 2 = 2$, $LB(\{\langle x_6, a\rangle, \langle x_5, d\rangle\}, \{x_4, ..., x_1\}) = 0 + 0 + 1 = 1$, $LB(\{\langle x_6, a\rangle, \langle x_5, d\rangle, \langle x_4, b\rangle\}, \{x_3, x_2, x_1\}) = 0 + 2 + 1 = 3$.



Above in the constraint graph constraints belonging to $\mathcal{W}^3$ are shown thick and dashed. In the middle RDS lower bound is depicted in three parts: the past (cost(t)), the past to future (sum of minimum ic's) and the future (rds_3) contributions.

In Fig. 3.2 we presents the RDS algorithm. The main function RDS is in charge of performing the $n-1$ successive resolutions. For this purpose it calls `PFC-RDS` which is essentially equivalent to PFC presented in previous Section 2.4.1. The only difference is that in order to obtain `PFC-RDS` one must substitute the call to function LB^{PFC} in pseudo-code of `PFC` (Fig. 2.6) with the new lower bound computation LB^{RDS}.

3.2 Specialized RDS

In this and the following Sections we present improvements of the RDS lower bound. The idea is to solve more subproblems in order to obtain their optimal costs and include them in the lower bound computation of subsequent resolutions. In *Specialized* RDS (SRDS) every subproblem including a single value of the new variable is solved. So SRDS disposes of an optimal cost for every value of the new included variable. An SRDS subproblem is defined as follows,

Definition 3.5 [SRDS subproblem] *Let $\mathcal{W} = \langle X, D, C\rangle$ be a WCSP and $\{x_1, x_2, ..., x_n\}$ a static variable ordering of its variables. $\mathcal{W}^{ia}$ is the subproblem $\mathcal{W}^i$ where the domain of variable x_i has been reduced to the singleton $\{a\}$.*

function PFC-RDS$(t, F, D, \texttt{ub})$
1 **if** $F = \emptyset$ **then return** cost(t)
2 $x_i \leftarrow$ get-var(F)
3 **for each** $a \in D_i$ **do**
4 look-ahead$(i, a, t, F - \{x_i\}, D^{i-1}, \texttt{ub})$
5 **if** $\text{LB}^{\text{RDS}}(t \cdot \langle x_i, a\rangle, F - \{x_i\}, D^{i-1}) < \texttt{ub}$ **then**
6 $nD^{i-1} \leftarrow$ prune$(i, a, t, F - \{x_i\}, D^{i-1}, \texttt{ub})$
7 **if** $\emptyset \notin nD^{i-1}$ **then**
8 $\texttt{ub}' \leftarrow$ PFC-RDS$(t \cdot \langle x_i, a\rangle, F - \{x_i\}, nD^{i-1}, \texttt{ub})$
9 **if** $\texttt{ub}' < \texttt{ub}$ **then** $\texttt{ub} \leftarrow \texttt{ub}'$
10 **return** ub

function RDS$(\mathcal{W})$
1 $rds_1 \leftarrow 0$
2 **for each** $i = 2$ **to** n **do**
3 $rds_i \leftarrow$ PFC-RDS$(\emptyset, X^i, D^i, \infty)$
4 **return** rds_n

function look-ahead$(i, a, t, F, D, \texttt{ub})$
1 **for each** $x_j \in F$ **do**
2 **for each** $b \in D_j$ **do**
3 $ic_{jb} \leftarrow ic_{jb} + f_{ij}(a, b)$

function prune$(i, a, t, F, D, \texttt{ub})$
1 **for each** $x_j \in F$ **do**
2 **for each** $b \in D_j$ **do**
3 **if** $\text{LB}^{\text{RDS}}_{jb}(t \cup \langle x_i, a\rangle, F, D) \geq \texttt{ub}$ **then**
4 $D_j \leftarrow D_j - \{b\}$
5 **return** D

function $\text{LB}^{\text{RDS}}(t, F, D^{i-1})$ **return** cost$(t) + rds_{i-1} + \sum_{x_j \in F} \min_{b \in D_j} \{ic_{jb}\}$

function $\text{LB}^{\text{RDS}}_{jb}(t, F, D^{i-1})$ **return** cost$(t) + ic_{jb} + rds_{i-1} + \sum_{x_k \in F, k \neq j} \min_{c \in D_k} \{ic_{kc}\}$

Figure 3.2: Russian Doll Search algorithm.

We enumerate domains $D_i = \{a_{i_1}, ..., a_{i_d}\}$. SRDS solves the sequence of subproblems in Fig. 3.3. Subproblems are solved from beginning in the right column from top to bottom and from right to left. SRDS solves optimally every subproblem for every value of the new included variable and the cost of including each value is stored. Thus solves $n \times d$ subproblems.

$$\begin{array}{cccc} \mathcal{W}^{n,a_{n_1}} & \ldots & \mathcal{W}^{2,a_{2_1}} & \mathcal{W}^{1,a_{1_1}} \\ \mathcal{W}^{n,a_{n_2}} & \ldots & \mathcal{W}^{2,a_{2_2}} & \mathcal{W}^{1,a_{1_2}} \\ \ldots & \ldots & \ldots & \ldots \\ \mathcal{W}^{n,a_{n_d}} & \ldots & \mathcal{W}^{2,a_{2_d}} & \mathcal{W}^{1,a_{1_d}} \end{array}$$

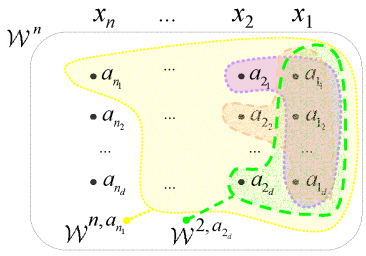


Figure 3.3: Left: Sequence of subproblems solved by SRDS. Right: In columns variables and their domains. We show for some SRDS subproblems which variables and values include.

Definition 3.6 *The optimal cost of subproblem* $\mathcal{W}^{ia}$ *is* rds_{ia}.

Property 3.2 $rds_i = \min_{a \in D_i} \{rds_{ia}\}$

Proof. Variable i has at least one value that belongs to the optimal solution of $\mathcal{W}^i$. This value has the minimum rds_{ia} which is equal to rds_i. All values of D_i that do not belong to one optimal solution have a cost greater than or equal to rds_i. □

Property 3.3 *rds_{ia} increases monotonically. That is, $\forall_a rds_{ia} \geq rds_{i-1}$.*

Proof. All the values a of the new included variable i have a cost greater than or equal to the optimal cost of solving subproblem $\mathcal{W}^{i-1}$ which is rds_{i-1}. □

One motivation for solving subproblems for every value comes from the fact that a variable domain can be very heterogeneous: a variable can have values with low cost and values with high cost. As subproblems grow in size, SRDS computes the optimal cost for every value of the new added variable. In subsequent resolutions SRDS may delete values earlier than RDS.

Example 3.3 *SRDS obtains this table of rds_{ia} when solving the 6-WQueens. We can observe the heterogeneity of optimal costs in the different values of each variable. RDS only computes the minimum optimal cost per column.*

	x_6	x_5	x_4	x_3	x_2	x_1
a	4	3		1		0
b	3	2	2		1	
c	4	3	1	2		0
d	4	3	3	1	0	

rds_{3d}

3.2.1 A new lower bound

SRDS combines rds_{ia} with inconsistency costs. We have that $\min_{a \in D_i}\{ic_{ia} + rds_{ia}\}$ is the cost that will certainly occur when assigning future variable x_i. This combination can only be done in one future variable because we are only allowed to use the rds_{ia} cost once as we could repeat costs coming from the same constraints. For the rest of future variables we can take its ic's contribution.

Definition 3.7 [SRDS lower bound] *Consider a search state such that t is the current assignment. F is the set of future variables and x_j an arbitrary future variable. The lower bound that SRDS computes is:*

$$LB^{SRDS}(t, F, j) = \texttt{cost}(t) + \min_{a \in D_j}\{ic_{ja} + rds_{ja}\} + \sum_{x_k \in F, k \neq j} \min_{b \in D_j}\{ic_{jb}\}$$

Since every future variable produces a valid lower bound, SRDS takes the best one.

$$LB^{SRDS}(t, F) = \max_{x_j \in F}\{LB^{SRDS}(t, F, j)\}$$

Property 3.4 *$LB^{SRDS}(t, F, j)$ is a lower bound.*

Proof. $\texttt{cost}(t)$ comes from the cost of constraints that link past assigned variables. The rds_{ja} contribution comes from the cost of constraints between future variables. The ic_{ja} contribution comes from a cost of constraints that link past with future variables. Both contributions rds_{ja} and ic_{ja} come from different constraints so we can safely add them taking the minimum of the sum of both in only one variable x_j, which is the minimum cost of extending the partial assignment to that variable no matter which value is assigned. Finally we can safely add the sum of minimum inconsistency costs for the rest of the variables different form x_j. □

The following property shows that SRDS lower bound is never worse than the RDS lower bound,

Property 3.5 *Consider a search state such that t is the current assignment. Let $F = \{x_i, ..., x_1\}$. For all $rds_{1a}, ..., rds_{ia}$ that have been computed, we have:*

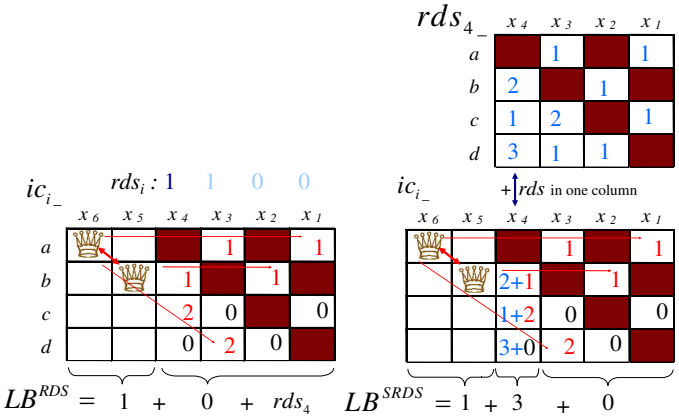
$$LB^{SRDS}(t, F) \geq LB^{RDS}(t, F)$$

Proof.

$$\begin{aligned}
LB^{SRDS}(t,F) &= \max_{x_j \in F} \{LB^{SRDS}(t,F,j)\} \\
&\geq LB^{SRDS}(t,F,i) \\
&= \texttt{cost}(t) + \min_{a \in D_i} \{ic_{ia} + rds_{ia}\} + \sum_{x_k \in F, k \neq i} \min_{a \in D_k} \{ic_{ka}\} \\
&\geq \texttt{cost}(t) + \min_{a \in D_i} \{ic_{ia}\} + \min_{a \in D_i} \{rds_{ia}\} + \sum_{x_k \in F, k \neq i} \min_{a \in D_k} \{ic_{ka}\} \\
&= \texttt{cost}(t) + \sum_{x_k \in F} \min_{a \in D_k} \{ic_{ka}\} + \min_{a \in D_i} \{rds_{ia}\} \\
&= \texttt{cost}(t) + \sum_{x_k \in F} \min_{a \in D_k} \{ic_{ka}\} + rds_i \\
&= LB^{RDS}(t,F)
\end{aligned}$$

Realizing that $\min_{a \in D_i}\{rds_{ia}\}$ is referred to the first variable in F, it is clear that $\min_{a \in D_i}\{rds_{ia}\} = rds_i$. □

Example 3.4 *Consider a search state when solving the 6-WQueens where $t = \{\langle x_6, a\rangle, \langle x_5, b\rangle\}$ and all subproblems $\mathcal{W}^{4\_}, ..., \mathcal{W}^{1\_}$ have been solved for all values (all $rds_{4\_}$ have been solved, where a value is noted by an under dash). In the drawing that follows next we show the lower bounds that RDS (first column) and SRDS (second column) compute.*



The table in the first column contains the ic's distribution after the assignment t. In the second column we have the lower bound table that SRDS has and we show superposed with the ic's distribution how the $rds_{4\_}$ can be combined with it. In the bottom of both columns we show the lower bounds computed by RDS ($rds_4 = 1$ so $LB^{RDS} = 2$) and SRDS ($LB^{SRDS} = 4$).

3.2.2 Future Value Pruning

Computing a specialized RDS cost for every value is very powerful for pruning values. A value b of a future variable x_j can be pruned when the lower bound specialized for that value is greater than or equal to the current `ub`. In SRDS there is a family of lower bounds, $LB(t, F, k)$, which can be specialized for value b of future variable j as follows,

$$LB_{jb}^{SRDS}(t,F,k) = \begin{cases} \texttt{cost}(t) + ic_{jb} + \min\limits_{a \in D_k}\{ic_{ka} + rds_{ka}\} + \sum\limits_{x_l \in F, l \neq j,k} \min\limits_{a \in D_k}\{ic_{ka}\} & k \neq j \\ \texttt{cost}(t) + ic_{jb} + rds_{jb} + \sum\limits_{x_l \in F, l \neq j} \min\limits_{a \in D_l}\{ic_{la}\} & k = j \end{cases}$$

The SRDS lower bound specialized for value b of future variable x_j is,

$$LB_{jb}^{SRDS}(t,F) = \max_{x_k \in F}\{LB_{jb}^{SRDS}(t,F,k)\}$$

which is always better than the specialized lower bound of RDS as we proof in the following property,

Property 3.6 *Consider a search state such that t is the current assignment. Let the set of future variables be $F = \{x_i, ..., x_1\}$. Therefore, all $rds_{1a}, ..., rds_{ia}$ have been computed. We have: $LB_{jb}^{SRDS}(t,F) \geq LB_{jb}^{RDS}(t,F)$.*

Proof.

If $j \neq i$, we have

$$\begin{aligned}
LB_{jb}^{SRDS}(t,F) &= \max_{k\in F}\{LB_{jb}^{SRDS}(t,F,k)\} \\
&\geq LB_{jb}^{SRDS}(t,F,i) \\
&= \texttt{cost}(t) + ic_{jb} + \min_{a\in D_i}\{ic_{ia} + rds_{ia}\} + \sum_{x_l\in F, l\neq i,j} \min_{a\in D_l}\{ic_{la}\} \\
&\geq \texttt{cost}(t) + ic_{jb} + \min_{a\in D_i}\{ic_{ia}\} + \min_{a\in D_i}\{rds_{ia}\} + \sum_{x_l\in F, l\neq i,j} \min_{a}\{ic_{la}\} \\
&= \texttt{cost}(t) + ic_{jb} + \sum_{x_l\in F, l\neq j} \min_{a\in D_l}\{ic_{la}\} + \min_{a\in D_i}\{rds_{ia}\} \\
&= LB_{jb}^{RDS}(t,F)
\end{aligned}$$

If $j = i$, we have

$$\begin{aligned}
LB_{jb}^{SRDS}(t,F) &= \max_{k} LB_{jb}^{SRDS}(t,F,k) \\
&\geq LB_{jb}^{SRDS}(t,F,i) \\
&= \texttt{cost}(t) + ic_{jb} + rds_{jb} + \sum_{x_l\in F, l\neq j} \min_{a\in D_l}\{ic_{la}\} \\
&\geq \texttt{cost}(t) + ic_{jb} + \sum_{x_l\in F, l\neq j} \min_{a\in D_l}\{ic_{la}\} + \min_{a\in D_j}\{rds_{ja}\} \\
&= LB_{jb}^{RDS}(t,F)
\end{aligned}$$

□

3.2.3 SRDS upper bound

As any other branch-and-bound based algorithm, SRDS has an upper and a lower bound at every search node. When SRDS solves subproblem $\mathcal{W}^{ia}$, it takes advantage of previously solved $\mathcal{W}^{jb}$, $(j < i)$. In the previous Section we showed how it uses the resolution of $\mathcal{W}^{jb}$ to improve its lower bound. Next, we show how it can also use it to improve the upper bound. SRDS maintains during the solving process a table of upper bounds. Each subproblem $\mathcal{W}^{ia}$ has an associated entry $\texttt{ub}_{ia}$. Before SRDS starts, all $\texttt{ub}_{ia}$ entries are set to infinity. When SRDS solves subproblem $\mathcal{W}^{ia}$, every time it finds an improving solution t, it updates its upper bound $\texttt{ub}^{ia}$. Besides, the solution t may also be a good starting solution for the next subproblem. So it also checks if $t \cup \langle x_{i+1}, b\rangle$ is an improving solution for each b of subproblem $\mathcal{W}^{i+1,b}$. If so, it also updates $\texttt{ub}^{i+1,b}$. As a result, when it is the time for solving $\mathcal{W}^{i+1,b}$, we will presumably have an initially good upper bound that will enhance the pruning. The solution t may also be a good solution for subproblem $\mathcal{W}^{ib}$. So it checks also if $(t - \langle x_i, a\rangle) \cup \langle x_i, b\rangle$ is an improving solution of $\mathcal{W}^{ib}$. If so it updates $\texttt{ub}^{i,b}$. SRDS algorithm of Fig. 3.4 incorporates

these upper bound adjustments inside function `adjustUB()` which is called every time a solution is improved.

A particularly advantageous situation occurs when, at the time of solving $\mathcal{W}^{ia}$, we have that $\mathtt{ub}^{ia} = \min_{b \in D_{i-1}}\{rds_{i-1,b}\}$. In that case, SRDS can skip $\mathcal{W}^{ia}$ resolution as it is proved that it is already the optimal cost, the addition of value a of variable x_i does not increment the optimal cost of subproblem $\mathcal{W}^i$.

The upper bounding techniques of this section can also be applied to RDS. The main difference is that instead of storing a lower bound for every value, RDS will only keep the minimal of all values.

3.2.4 The algorithm

Fig. 3.4 shows the pseudo-code of SRDS. Function SRDS is in charge of doing the sequence of nested subproblem resolutions. For this purpose PFC-SRDS is called for each value a of every variable x_i. At each resolution PFC-SRDS has four parameters: the current assignment t, the set of future variable F, the collection of domains D and the upper bound `ub`. PFC-SRDS is essentially the same algorithm than PFC-RDS replacing the lower bound functions with the new SRDS ones. When the set of future variables is empty we also apply the upper bounding techniques explained in previous Section 3.2.3. Line 7 PFC-SRDS checks a pruning condition that is valid before propagating ic's. Before propagating the effect of the current selected value a of variable x_i (that is before propagating its inconsistency cost done in function look-ahead) we can sum the specialized rds cost as a valid lower bound to the PFC lower bound. After look-ahead (that is the propagation of inconsistency cost of the current value in line 8 of `PFC-SRDS`) the previous pruning condition cannot be applied because inconsistencies coming from the same constraints could be counted twice.

In line 9 a safe approximation of the lower bound LB^{SRDS} is computed. Instead of computing the variable $x_j \in F$ that provides the highest contribution, x_{i-1} the first variable of F is taken. Something similar occurs in function LB^{SRDS}_{jb}, where no complete maximization is performed on the set F. Instead, the best specialized lower bound for x_{jb} is selected from two candidates, which differ in the variable that provides the rds contribution, x_{i-1} or x_j. These approximations have been done to reduce overhead without causing a serious decrement in the lower bound.

Limited SRDS

Consider a state of the search where RDS begins the resolution of subproblem $\mathcal{W}^i$ with `ub` as initial upper bound. At the end it obtains rds_i. At the same state, SRDS solves instead $\mathcal{W}^{i,a_1}, ..., \mathcal{W}^{i,a_d}$ and at each subproblem $\mathcal{W}^{i,a_j}$ resolution uses $\mathtt{ub}_{i,a_j}$ as initial upper bound. Consider now that we start solving $\mathcal{W}^{i,a_1}$. When the resolution finishes the optimal cost rds_{i,a_1}, is known. Now we can start solving $\mathcal{W}^{i,a_2}$, but instead of using $\mathtt{ub}_{i,a_2}$ as initial upper bound we use the previous computed optimal cost $\mathtt{ub} \leftarrow rds_{i,a_1}$. This is exactly what RDS does. RDS explores all the values of the first variable and as the search progresses the

function PFC-SRDS($t, F, D, \mathtt{ub}$)
1 **if** $F = \emptyset$ **then**
2 adjustUB(t)
3 **return** cost(t)
4 $x_i \leftarrow$ get-var(F)
5 **for each** $a \in D_i$ **do**
7 **if** $\mathrm{LB}^{\mathrm{PFC}}(t \cdot \langle x_i, a\rangle, F, D) + rds_{ia} < \mathtt{ub}$ **then**
8 look-ahead($i, a, t, F - \{x_i\}, D^{i-1}, \mathtt{ub}$)
9 **if** $\mathrm{LB}^{\mathrm{SRDS}}(t \cdot \langle x_i, a\rangle, F - \{x_i\}, D^{i-1}) < \mathtt{ub}$ **then**
10 $nD^{i-1} \leftarrow$ prune($i, a, t, F - \{x_i\}, D^{i-1}, \mathtt{ub}$)
11 **if** $\emptyset \notin nD^{i-1}$ **then**
12 $\mathtt{ub}' \leftarrow$ PFC-SRDS($t \cdot \langle x_i, a\rangle, F - \{x_i\}, nD^{i-1}, \mathtt{ub}$)
13 **if** $\mathtt{ub}' < \mathtt{ub}$ **then** $\mathtt{ub} \leftarrow \mathtt{ub}'$
14 **return** ub

function SRDS($\mathcal{W}$)
1 **for each** $i = 1$ **to** n **do**
2 **for each** $a \in D_i$ **do**
3 $rds_{ia} \leftarrow$ PFC-SRDS($\emptyset, X^i, \{a\} \cup D^{i-1}, \mathtt{ub}_{ia}$)
4 **return** $\min_{a \in D_n}\{rds_{na}\}$

function $\mathrm{LB}^{\mathrm{SRDS}}(t, F, D^{i-1})$
1 **return** $\mathtt{cost}(t \cup \langle x_i, a\rangle) + \min_{b \in D_{i-1}} \{ic_{i-1,b} + rds_{i-1,b}\} + \sum_{x_j \in F, j \neq i-1} \min_{b \in D_j} \{ic_{jb}\}$

function $\mathrm{LB}^{\mathrm{SRDS}}_{jb}(t, F, D^{i-1})$
1 $lb_1 \leftarrow \mathtt{cost}(t) + ic_{jb} + + \min_{c \in D_{i-1}} \{ic_{i-1,c} + rds_{i-1,c}\} + \sum_{x_k \in F, k \neq j, i-1} \min_c\{ic_{kc}\}$
2 $lb_2 \leftarrow \mathtt{cost}(t) + ic_{jb} + rds_{jb} + \sum_{x_k \in F, k \neq j} \min_{c \in D_k} \{ic_{kc}\}$
3 **if** $(lb_1 > lb_2)$ **return** lb_1 **else return** lb_2

function adjustUB($t = \{..., \langle x_i, a\rangle\}$)
1 $\forall_{b \in D_i} \mathtt{ub}_{i,b} \leftarrow \min\{\mathtt{ub}_{i,b}, \mathtt{cost}(t - \langle x_i, a\rangle \cup \langle x_i, b\rangle)\}$
2 $\forall_{b \in D_{i+1}} \mathtt{ub}_{i+1,b} \leftarrow \min\{\mathtt{ub}_{i+1,b}, \mathtt{cost}(t\rangle \cup \langle x_{i+1}, b\rangle)\}$

Figure 3.4: Specialized Russian Doll Search algorithm.

ub decreases monotonically. We call this algorithm *Limited* SRDS (LSRDS) and its pseudo-code is shown in Fig. 3.5. The LSRDS function is essentially equal to SRDS but when the PFC-SRDS call finishes, we set the upper bound to the just obtained optimal cost (line 5).

Property 3.7 *LSRDS has the same computational effort than RDS but has a better pruning capacity.*

function LSRDS($\mathcal{W}$)
1 ub $\leftarrow \infty$
2 **for each** $i = 1$ **to** n **do**
3 **for each** $a \in D_i$ **do**
4 $rds_{ia} \leftarrow$ PFC-SRDS$(\emptyset, X^i, \{a\} \cup D^{i-1}$, ub$)$
5 ub $\leftarrow rds_{ia}$
6 **return** ub

Figure 3.5: Limited Specialized Russian Doll Search algorithm.

Proof. LSRDS starts with ub $\leftarrow \infty$ and then passes to each resolution of $\mathcal{W}^{ia}$ the best known upper bound up to the moment (line 5). So it still computes the rds_{ia} contribution until it finds the minimum for that variable. This minimum is stored as the rds_{ia} contribution for the values which were not still processed when that minimum was found. LSRDS requires exactly the same search effort as RDS (and both compute the minimum rds_{ia} contribution of a subproblem), $LB^{SRDS}(t, F, D)$ combines ic and rds counters in one future variable, while $LB^{RDS}(t, F, D)$ always takes the rds contribution in isolation. Because of that, LSRDS has a higher pruning capacity than RDS with the same effort. □

In cases where SRDS is too costly one can think of switching to LSRDS. For example one can think of specializing the first included variables and switch to LSRDS for including the other variables. In this case, we say that we restrict SRDS up to a certain subproblem. After this subproblem we use LSRDS. We take into account that after limiting SRDS the costs stored in the lower bound tables may not be optimal any more. For that reason we use an alternative name for the rds tables:

Definition 3.8 *Let* $\mathtt{lb}_{ia}$ *be a lower bound on the optimal cost of including value* a *of variable* i *in subproblem* $\mathcal{W}^{i-1}$*, that is* $\mathtt{lb}_{ia} \leq rds_{ia}$*.*

One way of knowing if rds tables entries are optimal is to check if $\mathtt{ub}_{ia} = \mathtt{lb}_{ia}$ which is a sufficient condition for optimality.

Example 3.5 *LSRDS obtains this table of* $\mathtt{lb}_{ia}$ *costs when solving the 6-WQueens. Values are assigned from top to bottom. Observe that once the optimal cost is found the remaining values get this optimal* rds_{ia}*. First included values may have a higher cost than* $rds_i = \min_{a \in D_i}\{rds_{ia}\}$*.*

	x_6	x_5	x_4	x_3	x_2	x_1
a	4	3		1		0
b	3	2	2		1	
c	3	2	1	1		0
d	3	2	1	1	0	

3.3 Full SRDS

When we solve each $\mathcal{W}^{ia}$ subproblem we observe that optimal costs for values of the first included variables are low. The reason being that optimal costs increase as more variables and constraints are included. Because of this fact, the lower bound that we compute when we are in a deep level of the search tree, or when we want to prune values of the first included variables is poor. To improve the lower bound in these cases we propose to some more resolutions than SRDS by also specializing the values of the first included variables. We call this new RDS lower bound *Full Specialized* RDS (FSRDS).

Definition 3.9 [FSRDS subproblem] *Let $\mathcal{W} = \langle X, D, C \rangle$ be a WCSP and $\{x_1, x_2, ..., x_n\}$ a static variable ordering of its variables. $\mathcal{W}^i_{ja}$ is the subproblem $\mathcal{W}^i$ where the domain of variable x_j has been reduced to the singleton a.*

We enumerate domains $D_i = \{a_{i_1}, ..., a_{i_d}\}$. FSRDS solves the sequence of subproblems that appears in Fig. 3.6. Problems are solved column by column from left to right.

$$\begin{array}{cccccc}
\mathcal{W}^1_{1,a_{1_1}} & \mathcal{W}^2_{1,a_{1_1}}\mathcal{W}^2_{2,a_{2_1}} & \mathcal{W}^3_{1,a_{1_1}}\mathcal{W}^3_{2,a_{2_1}}\mathcal{W}^3_{3,a_{3_1}} & & \mathcal{W}^n_{1,a_{1_1}}...\mathcal{W}^n_{n,a_{n_1}} \\
\mathcal{W}^1_{1,a_{1_2}} & \mathcal{W}^2_{1,a_{1_2}}\mathcal{W}^2_{2,a_{2_2}} & \mathcal{W}^3_{1,a_{1_1}}\mathcal{W}^3_{2,a_{2_1}}\mathcal{W}^3_{3,a_{3_1}} & & \mathcal{W}^n_{1,a_{1_2}}...\mathcal{W}^n_{n,a_{n_2}} \\
... & ... & ... & ... & ... \\
\mathcal{W}^1_{1,a_{1_d}} & \mathcal{W}^2_{1,a_{1_d}}\mathcal{W}^2_{2,a_{2_d}} & \mathcal{W}^3_{1,a_{1_1}}\mathcal{W}^3_{2,a_{2_1}}\mathcal{W}^3_{3,a_{3_1}} & & \mathcal{W}^n_{1,a_{1_d}}...\mathcal{W}^n_{n,a_{n_d}}
\end{array}$$

Figure 3.6: Sequence of subproblems solved by FSRDS.

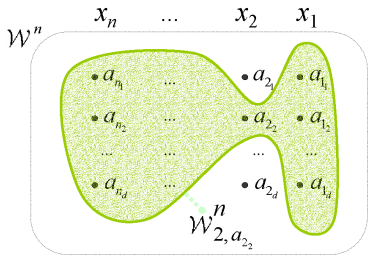


Figure 3.7: The filled area includes all values of variables of the FSRDS subproblem $W^n_{2,a_{2_2}}$ which was not previously consider by RDS neither SRDS.

Definition 3.10 *The optimal cost of solving $\mathcal{W}^j_{ia}$ is rds^j_{ia}.*

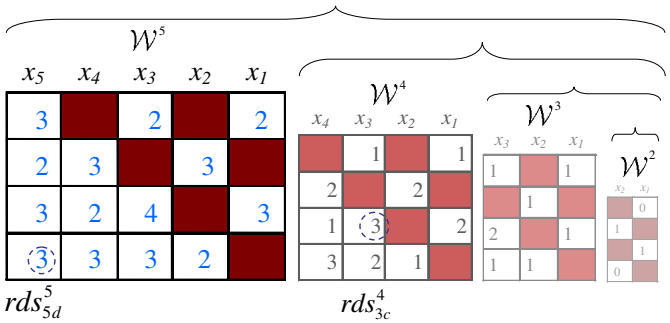
The essential difference between SRDS and FSRDS is that the latter computes the cost of including every value of all variables in the current subproblem. Consequently FSRDS solves $n \times n \times d$ subproblems. SRDS computes rds^i_{ia} the optimal cost of including every value of the i highest variable. FSRDS also

computes $rds_{ia}^{i-1}, ..., rds_{ia}^{1}$. By doing so, the smallest subproblems also have good lower bounds for every value. It is worth noting that the specialized lower bound for a particular value a of variable i increases monotonically as we solve more subproblems $\mathcal{W}^j$, that is:

Property 3.8 $\forall_{i,a}(rds_{ia}^{j+1} \geq rds_{ia}^{j})$.

Proof. The optimal cost of including value a of variable x_i in subproblem $\mathcal{W}^{j-1}$ is the same or higher than when this value of x_i was included in $\mathcal{W}^j$. □

Example 3.6 *Consider the four initial FSRDS resolutions when solving the 6-WQueens. Find below the four lower bound tables that this algorithm computes solving subproblems* $\mathcal{W}_{ia}^2, ..., \mathcal{W}_{ia}^5$.



3.3.1 A new lower bound

Definition 3.11 [FSRDS lower bound] *Consider a search state such that t is the current assignment. F is the set of future variables and k an arbitrary future variable. The lower bound that FSRDS computes is:*

$$LB^{FSRDS}(t, F = \{x_i, ..., x_1\}, k) = \texttt{cost}(t) + \min_{a \in D_k}\{ic_{ka} + rds_{ka}^{i}\} + \sum_{x_l \in F, l \neq k} \min_{a \in D_l}\{ic_{la}\} \quad \forall x_k \in F$$

The best lower bound of this family is:

$$LB^{FSRDS}(t, F) = \max_{x_k \in F}\{LB^{FSRDS}(t, F, k)\}$$

Property 3.9 $\forall x_k \in F$, $LB^{FSRDS}(t, F, k)$ *is a lower bound.*

Proof. Similar to the proof of property 3.4. □

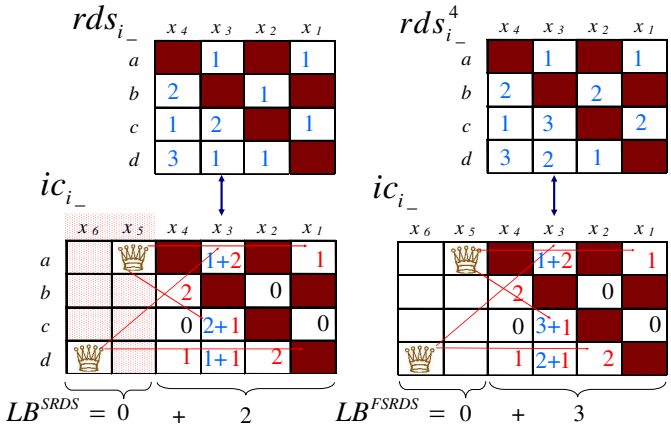
Property 3.10 *All values a of all variables x_i of subproblem $\mathcal{W}^j$ that have an equal minimal rds_{ia}^{j} belong to an optimal solution of this subproblem.*

Proof. If values are optimally specialized the minimal cost of including all values a of variable x_i in subproblem $\mathcal{W}^{j-1}$ is the optimal cost of solving $\mathcal{W}^j$. If there are several minimal rds^j_{ia} then these values belong to different optimal solutions. □

Property 3.11 *The values of every variable that have a minimal rds, that is* $\min_{a \in D_i}\{rds^j_{ia}\}$ *represent a super set of all the optimal solutions of the subproblem* $\mathcal{W}^j$*.*

Proof. A value a with minimal cost in variable i belongs for sure to an optimal solution if all the values have been optimally specialized. It may happen that two minimal values of different variables belong to different optimal solutions, so its in fact a super set. □

Example 3.7 *Consider a search state where* $t = \{\langle x_6, d\rangle, \langle x_5, a\rangle\}$ *and all subproblems* $\mathcal{W}^4_{ia}, ..., \mathcal{W}^1_{ia}$, $i = 1..4$ *have been solved for all a values. The tables in the first column of the drawing below correspond respectively to the SRDS lower bound and the inconsistency cost distribution after the assignment t.*



The second column correspond to the same tables for FSRDS. The combination of ic's and rds^j_{ia} *is done in variable* x_3*.*

3.3.2 Future Value Pruning

A value b of a future variable x_j can be pruned when the lower bound specialized for that value is greater than or equal to the current `ub`. In FSRDS there is a family of lower bounds, $LB^{FSRDS}(t, F, k)$, which can be specialized for each value b of each future variable x_j as follows,

$$LB^{FSRDS}_{jb}(t,F,k) = \begin{cases} cost(t) + ic_{jb} + \min\limits_{a \in D_k}\{ic_{ka} + rds^i_{ka}\} + \sum\limits_{x_l \in F, l \neq k,j} \min\limits_{a}\{ic_{la}\} & j \neq k \\ cost(t) + ic_{jb} + rds^i_{jb} + \sum\limits_{x_l \in F, l \neq j} \min\limits_{a \in D_l}\{ic_{la}\} & j = k \end{cases}$$

function FSRDS($\mathcal{W}$)
1 **for each** $i = 1$ **to** n **do**
2 **for each** $j = i$ **down to** 1 **do**
3 **for each** $a \in D_j$ **do**
4 $rds^i_{ja} \leftarrow$ PFC-FSRDS$(\emptyset, X^i, \{D_1, ..., D_{j-1}, \{a\}, D_{j+1}, ..., D_i\}, \mathtt{ub}^i_{ja})$
5 **return** $\min_{a \in D_n}\{rds^n_{na}\}$

function $\mathtt{LB}^{\mathtt{FSRDS}}(t, F, D^{i-1})$
1 **return** $\max_{x_k \in F}\{\mathtt{cost}(t) + \min_{a \in D_k}\{ic_{ka} + rds^{i-1}_{ka}\} + \sum_{x_l \in F, l \neq k} \min_{a \in D_l}\{ic_{la}\}\}$

function $\mathtt{LB}^{\mathtt{FSRDS}}_{jb}(t, F, D^{i-1})$
1 **return** $\mathtt{cost}(t) + rds^{i-1}_{jb} + \sum_{x_k \in F, k \neq j} \min_{c \in D_k}\{ic_{kc}\}$

Figure 3.8: Full Specialized Russian Doll Search algorithm.

The FSRDS lower bound specialized for value b of future variable x_j is,

$$LB^{FSRDS}_{jb}(t, F) = \max_{x_k \in F}\{LB^{FSRDS}_{jb}(t, F, k)\}$$

which is always better than the specialized lower bound of SRDS.

Property 3.12 *Using the same static variable ordering* $LB^{FSRDS}_{jb}(t, F) \geq LB^{SRDS}_{jb}(t, F)$.

Proof. Similar to the proof of property 3.5. □

3.3.3 The algorithm

FSRDS has to specialize values of variables that are not necessarily the last included variable. The strategy to specialize a value will be to fix the domain of the variable to this particular value. Doing so FSRDS obtains the optimal cost of including that value a of variable x_j in the subproblem $\mathcal{W}^{i-1}$.

The pseudo code of FSRDS is shown in Fig. 3.8. The main function FSRDS is in charge of doing all subproblems resolutions. It includes variables one by one (line 1) and all values of every variable of the current subproblem (lines 2 and 3). When solving $\mathcal{W}^i_{ja}$ FSRDS starts with the best known upper bound for it and fixes the domain of variable x_j to value a (line 4). PFC-FSRDS is essentially the same as PFC-SRDS substituting the corresponding lower bound functions with the new functions appearing in Fig. 3.8.

3.3.4 FSRDS upper bound

The upper bounding techniques of Section 3.2.3 can also be applied to FSRDS. FSRDS maintains during the solving process a table of upper bounds $\mathtt{ub}^i_{ja}$. Before FSRDS starts, all $\mathtt{ub}^i_{ja}$ entries are set to infinity. When FSRDS solves subproblem $\mathcal{W}^i_{ja}$, every time it finds a better solution $t = \{\langle x_1, a_1\rangle, ..., \langle x_{i-2}, a_{i-2}\rangle, \langle x_{i-1}, a_{i-1}\rangle, \langle x_i, a_i\rangle\}$, it can update $\mathtt{ub}^i_{i,a_i}$ but also $\mathtt{ub}^i_{i,a_{i-1}}$, ..., $\mathtt{ub}^i_{1,a_1}$. FSRDS can update all upper bounds of the values of the found solution. The solution t found may also be a good solution for the next subproblem. So it also checks if $t \cup \langle x_{i+1}, b\rangle$ is an improving solution for each b of subproblem $\mathcal{W}^{i+1}_{i+1,b}$. If so, it also updates $\mathtt{ub}^{i+1,b}$.

Limited FSRDS

As for SRDS (3.2.4) we describe a *limited* version of FSRDS. The idea is to allow skipping resolutions of subproblems but still using the FSRDS lower bound which is better than its predecessor SRDS. Thus the cost stored in the lower bound tables may not be optimal any more.

Definition 3.12 *Let* $\mathtt{lb}^j_{ia}$ *be a lower bound of* $rds^j_{i,a}$*, that is* $\mathtt{lb}^j_{ia} \leq rds^j_{i,a}$

$\mathtt{lb}^j_{ia}$ are used instead of rds^j_{ia} when the obtained lower bound is not proven to be optimal but is instead an underestimation.

Consider we just solved subproblem $\mathcal{W}^{i-1}$.

1) We can update the table of the next subproblem: $\forall_{j \in X^{i-1}} \forall_{a \in D_j} \mathtt{lb}^i_{ja} \leftarrow \mathtt{lb}^{i-1}_{ja}$.

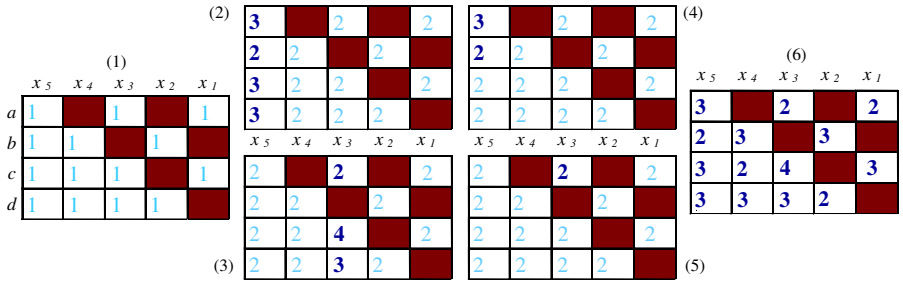
2) We can set all the values of the new included variable x_i to the optimal cost rds_{i-1}: $\forall_{a \in D_j} \mathtt{lb}^i_{ja} \leftarrow rds_{i-1}$. Now we need to find the optimal cost of $\mathcal{W}^i$. This implies specialization of one variable x_k of $\mathcal{W}^i$, in two possible ways:

1. Specialize all values of variable x_k.

2. Specialize the values of variable x_k until the minimum cost of solving subproblem i has been found.

Any variable k is suitable for this. The following example illustrates this.

Example 3.8 *Consider we have already solved* $\mathcal{W}^4, ..., \mathcal{W}^1$ *of the 6-WQueens using RDS and obtained* $rds_4 = 1$*. Before start solving* $\mathcal{W}^5$ *we can set all values of table* $\mathtt{lb}^j$ *to* 1 *(table* 1 *in the drawing that follows next).*



Then we can specialize all values of the first variable x_5 as SRDS does, we obtain table 2*. Optimal values are indicated in bold. In table* 2 *we have set all values of the other variables to the optimal cost of subproblem $\mathcal{W}^5$ which is known after the specialization of one variable. We can also specialize all values of variable x_3, we then obtain table* 3*. We can also use LSRDS to specialize all values of variable x_5 (table* 4*). If we use LSRDS starting in variable x_3 we obtain table* 5*. We can also specialize all values obtaining table* 6*.*

3.4 Opportunistic RDS

In the previous Section we showed that it is not necessary to specialize all values, we can dispose of a table of underestimations of *rds* costs. We loose time specializing a value, but in turn we then have a high lower bound for it. In this section we are going to study the possibility of skipping value specializations and identifying a posteriori which of the skipped specializations is promising to increment the lower bound.

In a particular node of the search, only one variable x_j and one value b (the one with minimum sum of *ic*'s plus rds^i_{kc}) is chosen to contribute to the lower bound. In general, unless all rds^i_{kc} are computed, it is impossible to determine the pair $\langle x_j, b\rangle$ that offers the best combination between *ic* and *rds* in a given tree branch when solving subproblem $\mathcal{W}^{i+1}$. However, once variable x_{i+1} has been assigned and its *ic*'s have been propagated, it is possible to identify some $\langle variable, value\rangle$ pairs which look promising to increase the lower bound on the basis of the current *ic* distribution.

We propose a new algorithm that we call *Opportunistic* RDS (ORDS). The main idea consists in first solving subproblem $\mathcal{W}^i$ using RDS (or the LSRDS version which has the same computational effort) and then start solving subproblem $\mathcal{W}^{i+1}$ identifying advantageous specializations. After assigning variable x_{i+1} and *ic*'s have been propagated we look for the values that, if they are removed, the lower bound is increased. For doing so we compute,

$$w = \max_{x_j \in F} \{\min_{b \in D_j} \{ic_{jb} + \mathtt{lb}^i_{jb}\} + \sum_{x_k \in F, k \neq j} \min_{a \in D_k} \{ic_{ka}\}\}$$

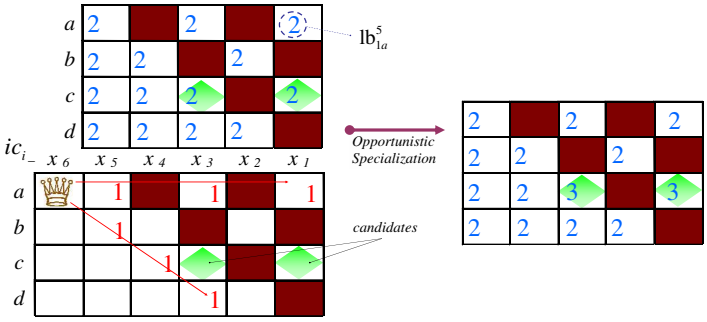
w is the maximum lower bound that we can obtain. To compute w, the sum

of ic's and rds is combined in variable x_j summing the contributions of value b. If the next minimum is different from w we have found a unique minimum $\langle$variable, value$\rangle$ pair. If this pair could be specialized and its optimal cost raised, the lower bound will be increased for sure.
All pairs $\langle x_j, b\rangle$ such that,

1. $w = \min_{b \in D_j} \{ic_{jb} + \mathtt{lb}^i_{jb}\} + \sum_{x_k \in F, k \neq j} \min_{a \in D_k} \{ic_{ka}\}$
2. $w < ic_{ja} + \mathtt{lb}^i_{ja} + \sum_{x_k \in F, k \neq j} \min_{a \in D_k} \{ic_{ka}\}, \quad \forall a \in D_j - \{b\}$
3. $\mathtt{lb}^i_{jb} < \mathtt{ub}^i_{jb}$

are the candidates to increase the lower bound if their exact rds^i_{jb} costs where known. In words, condition 1) states that the pair $\langle x_j, b\rangle$ is the one with minimum combination of ic's plus rds. Condition 2) states that if we remove value b of variable x_j the lower bound increases. Condition 3) assures that the value is not already optimal.

Example 3.9 *Consider we have already solved $\mathcal{W}^5, ..., \mathcal{W}^1$ of the 6-WQueens using RDS. We start solving $\mathcal{W}^6$ with ORDS. x_6 has been assigned value a. The tables that follow next illustrate an example of specialization of values.*



The top table in the first column shows the unspecialized lower bounds $\mathtt{lb}^5_{ia}$. The table of inconsistency costs is shown above (an ic is shown if it is greater than 0). The possible candidates for specialization are higlighted. These are the unique minimums $ic_{jc} + \mathtt{lb}^i_{jc}$ in every variable. Δ associated to variable x_3 is equal to 1, which is the maximum increment that we can get. Δ associated to variable x_1 is also 1. On the right we show the resulting lower bound table after the specialization of both candidates. After this operation the lower bound increases one unit.

Consider that $\langle x_j, b\rangle$ is a candidate for specialization which satisfies all three conditions. How much can we improve the lower bound by specializing $\langle x_j, b\rangle$? After specialization $\mathtt{lb}^i_{jc}$ hopefully increases and becomes optimal. But is it

necessary to compute the optimal. The lower bound is now determined now by the next minimum $ic_{jc} + \mathtt{lb}^i_{jc}$ in variable x_j. Let $c \in D_j$ be the value with the next minimum combination of ic's and rds. The maximum increment that we can get by specializing $\langle x_j, b\rangle$ is the difference with the lower bound computed with the next minimum. This difference is $\Delta = ic_{jc} + \mathtt{lb}^i_{jc} - ic_{jb} - \mathtt{lb}^i_{jb}$. Δ is the maximum increment we can get no matter how high rds^i_{jb} could be. Therefore, when a value is specialized opportunistically, we do not ask for the optimum value of the subproblem $\mathcal{W}^i$ with value $\langle x_j, b\rangle$. Instead, we pass to FSRDS the maximum achievable increment of the lower bound as a parameter, and as soon as this increment has been achieved, the specialization process stops. This is done by calling FSRDS with a fake upper bound $\mathtt{ub} = \Delta + \mathtt{lb}^i_{jb}$. If no solution is found then this upper bound becomes automatically a lower bound.

When solving subproblem $\mathcal{W}^{i+1}$ and pair $\langle x_j, b\rangle$ has to be specialized, it is not mandatory to do it in the previous subproblem $\mathcal{W}^i$. Pair $\langle x_j, b\rangle$ can be specialized in any previous subproblem $\mathcal{W}^k$ including x_j with $i \leq k \leq j$. Obviously, subproblem $\mathcal{W}^k$ has to allow enough room for improvement, that is, $\mathtt{ub}^k_{jb} \geq \Delta + \mathtt{lb}^i_{jb}$.

In Fig. 3.9 we show the opportunistic specialization function. `OSpecialize` must be called after `look-ahead` in PFC-RDS function of Fig. 3.2. The set of candidates is computed following the previous specifications (line 1). Then we specialize them one by one. First we compute the next minimum (line 3), it exists by construction of the candidates set. Then Δ cost is computed (line 4). `OSpecialize` looks for a subproblem $\mathcal{W}^k$ candidate for the specialization (line 5). It checks if rds is not already optimal (line 6). If it is not it finally specializes the pair $\langle x_j, b\rangle$ by calling PFC-FSRDS and updating the rds cost (line 7).

function `OSpecialize`(i,D)
1 $candidates \leftarrow \{\langle x_j, b\rangle \mid$ satisfy conditions 1,2 and 3$\}$
2 **for each** $\langle x_j, b\rangle \in candidates$ **do**
3 $\quad c \leftarrow \min_{a \in D_j - \{b\}}\{ic_{ja} + \mathtt{lb}^i_{ja}\}$
4 $\quad \Delta \leftarrow ic_{jc} + \mathtt{lb}^i_{jc} - ic_{jb} - \mathtt{lb}^i_{jb}$
5 $\quad$ find k such that $\mathtt{ub}^k_{jb} \geq \Delta + \mathtt{lb}^k_{jb},\ j \geq k$
6 $\quad$ **if** $\mathtt{lb}^k_{jb} < \mathtt{ub}^k_{jb}$ **then**
7 $\quad\quad \mathtt{lb}^k_{jb} \leftarrow$ `PFC-FSRDS`$(\{\}, \{x_k, ..., x_1\}, \{D_k, ..., D_j = \{b\}, ..., D_1\}, \Delta + \mathtt{lb}^k_{jb})$

Figure 3.9: Opportunistic Specialization function.

3.5 Experimental Evaluation

We have tested RDS, LSRDS, SRDS, FSRDS and ORDS in four benchmarks: random problems, frequency assignment problems (FAP), earth satellite management (Spot) and combinatorial auctions.

All specialized versions can be limited to a certain number of variables, meaning that the specialization (independently of being SRDS, FSRDS, or ORDS) is only done up to a certain number of variables. So for example ORDS(lim=15) means that we use the opportunistic specialization but only specialize the cost of a value when we reach a subproblem of size less than 15 variables. If the subproblem is bigger than that we proceed as LSRDS. If the limited parameter is not indicated, it is assumed that algorithms are executed with no limitation. This fact allows for many variants of the tested algorithms.

3.5.1 Random Problems

In the Appendix A.2.1 we describe the weighted version of random problems benchmark. Random problems do not show the entire advantages of specializing RDS: all values of a variable have a similar expected cost because of the homogeneity of the constraint tightness in the whole problem. Nevertheless, we use this benchmark to evaluate in detail all the variants of the algorithms in addition to its limited versions, because random problems are easier to manipulate. We extract from the results general conclusions. We focus in two main points:

- We quantify how better LSRDS is with respect to RDS.
- We evaluate SRDS, FSRDS and ORDS w.r.t. their limited versions.

In Section 3.2.3 we described some upper bounding techniques that can be applied in RDS and all specialized versions. In our implementation all versions have this option enabled. We have observed that turning off the upper bound adjustment decreases the performance of all algorithms by a factor of 2 approximately. So from now on we suppose that all algorithms always use it.

LSRDS versus RDS

LSRDS is SRDS limited to problems of size 0, meaning that we never specialize a value, but we record its cost even if it is not optimal. As it was proven in Section 3.2.4, LSRDS has the same computational effort as RDS but it has a more powerful lower bound. The advantage of LSRDS is that the ic's can be combined with the rds contribution and also it has of a better pruning lower bounds for every value. When LSRDS has finished assigning a value of the first included variable of the subproblem, it stores its cost even if it is not optimal. RDS discards this information as it only records the minimal cost. So for example if, during search, values with high cost are selected first, LSRDS is able to use its cost in the lower bound computation. LSRDS may be more advantageous when there are few solutions of minimal cost and many values with cost higher than the optimal. In Fig. 3.10 we show comparative results of RDS and LSRDS on the random problem class $\langle n = 20, m = 5, p1 = 0.9 \rangle$. LSRDS is clearly more competitive than RDS. For tightness around 0.85 is 3 times faster than RDS. Another important fact is that the variance in the CPU time is highly reduced.

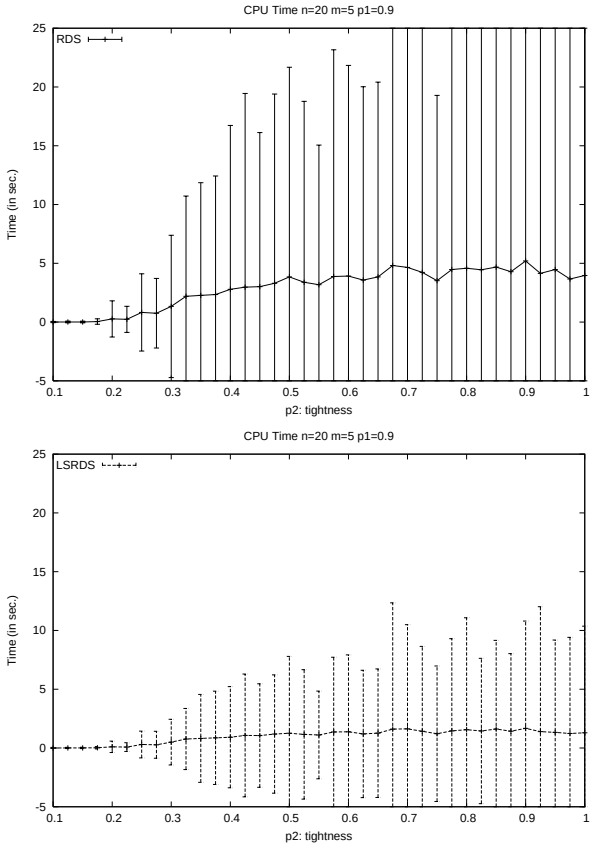


Figure 3.10: Average cpu time of algorithms RDS (left) and LSRDS (right) for random problem class $\langle n = 20, m = 5, p1 = 0.9\rangle$ with varying tightness $p2$ (horizontal axis). Error bars show the variance in the CPU time.

SRDS

In Fig. 3.10 we show comparative results of SRDS($lim = n/2$) and SRDS on the random problem class $\langle n = 20, m = 5, p1 = 0.9\rangle$. Both executions have very similar results. This may be due to the fact that for SRDS, the time spent specializing more values compensates and at the end the pay-off is the same. Of course it is always better to have the lower bound tables that SRDS computes. This is because even if it is only the first included variable the one that is specialized, the combination of ic's and rds contribution can be done in any future variable. With respect to implementation issues, it is possible to compute the variable that has a maximal combination of ic's and rds during the look-ahead with no extra overhead.

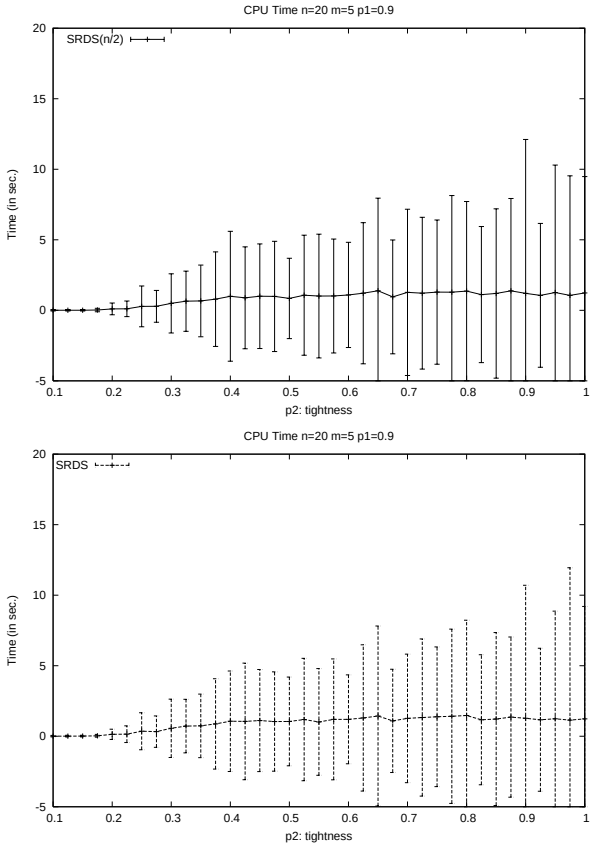


Figure 3.11: Average cpu time of algorithms SRDS($lim = n/2$) (left) and SRDS (right) for random problem class $\langle n = 20, m = 5, p1 = 0.9\rangle$ with varying tightness $p2$ (horizontal axis). Error bars show the variance in the CPU time.

FSRDS

In Fig. 3.12 we show comparative results of FSRDS($lim = n/2$) and FSRDS on the random problem class $\langle n = 20, m = 5, p1 = 0.9\rangle$. The full specialization of FSRDS is more costly than FSRDS($lim = n/2$).

We have observed a degenerated behavior when the size of subproblems to be full specialized increases. The variance in both cases is very high. FSRDS is the worst performing algorithm in average. The specialization in FSRDS does not pay-off. This fact does not discard its use in practice as we show in the FAP benchmark Section 3.5.2. The table of lower bounds that offers FSRDS is superior to SRDS. FSRDS can be useful to be combined with SRDS and LSRDS to dispose of more accurate lower bounds of the first small subproblems.

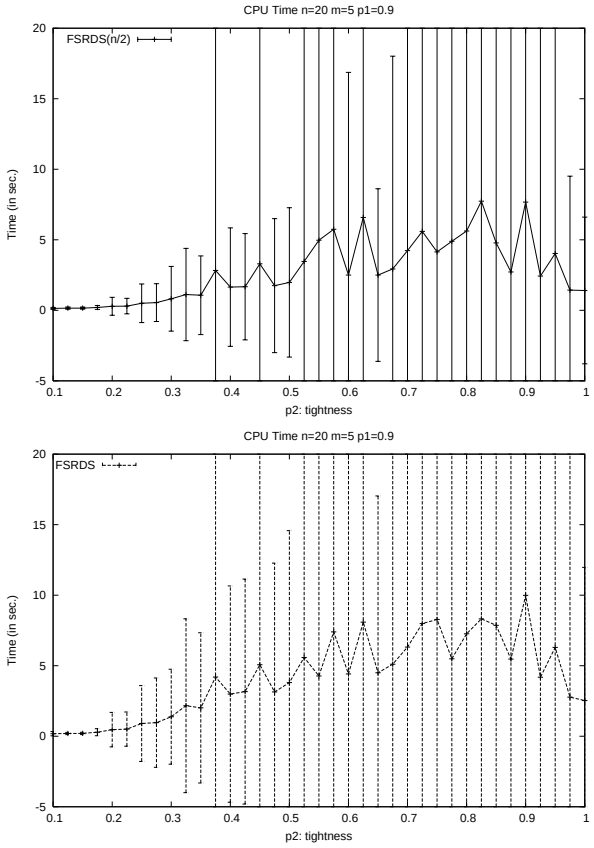


Figure 3.12: Average cpu time of algorithms FSRDS($lim = n/2$) (left) and FSRDS (right) for random problem class $\langle n = 20, m = 5, p1 = 0.9\rangle$ with varying tightness $p2$ (horizontal axis). Error bars show the variance in the CPU time.

ORDS

In Fig. 3.13 we show comparative results of ORDS($lim = n/2$) and ORDS on the random problem class $\langle n = 20, m = 5, p1 = 0.9\rangle$. ORDS can in theory specialize the same subproblems as FSRDS but the specialization is guided by the ic's distribution and by the fact that any specialization must be able to increase the lower bound. So the expected behavior is that only those values that seem promising to increase the lower bound are specialized. This fact is observed in Fig. 3.13. The limited version ORDS($lim = n/2$) has a behavior that is similar to LSRDS. The algorithm is stable with low variance and a low number of values is specialized. ORDS, on the contrary, specializes a higher number of values and we start to observe a degenerated behavior just as in FSRDS.

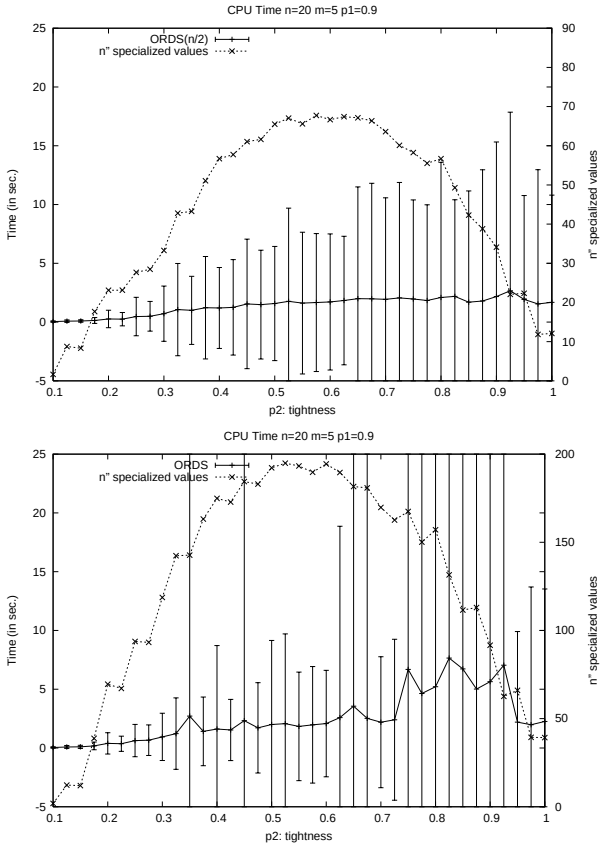


Figure 3.13: Average cpu time of algorithms ORDS($lim = n/2$) (left) and ORDS (right) for random problem class $\langle n = 20, m = 5, p1 = 0.9\rangle$ with varying tightness $p2$ (horizontal axis). Error bars show the variance in the CPU time. We show in an additional curve the number of specialized values.

Integrating RDS versions

In the previous experiments we observed that: in two cases, FSRDS and ORDS, the execution of the algorithms with no limitation can led to a degenerated behavior. SRDS seem to be more robust than the rest of algorithms.

In Fig. 3.14 we show average number of nodes and constraint checks for all the limited versions of the algorithms of the previous sections for the same class of random problem. The more efficient algorithm in terms of number of checks and nodes is ORDS($lim = n/2$) followed by LSRDS then SRDS($lim = n/2$) RDS and FSRDS. Random problems instances have values with very homogenous costs. In this case a conservative strategy which only specialize those that seem promising is very advantageous. Moreover it can be seen by the best performance of ORDS in terms of number of nodes that it often occurs that the specializations

are productive in increasing the lower bound.

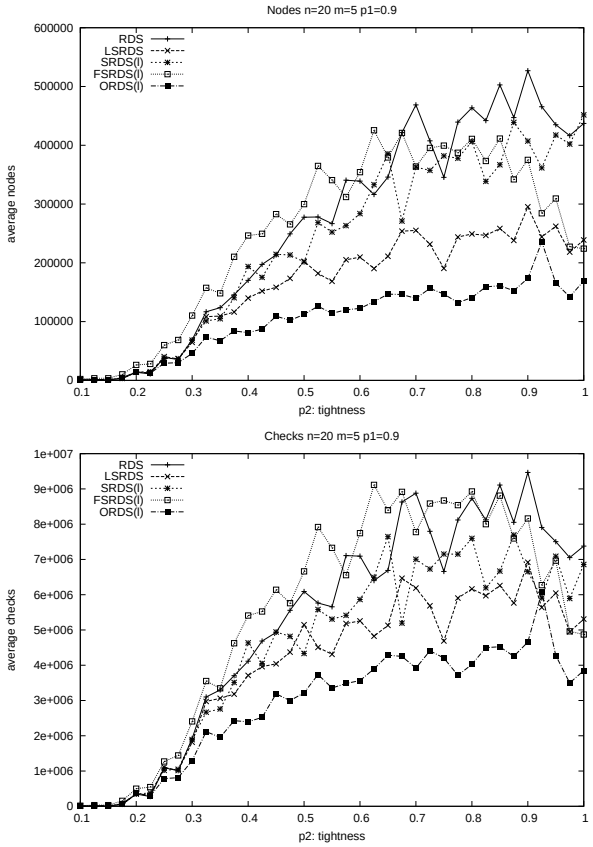


Figure 3.14: Average number of visited nodes (left) and average number of constraint checks of algorithms RDS, LSRDS, SRDS($lim = n/2$), FSRDS($lim = n/2$), ORDS($lim = n/2$) executed in random problem class$\langle n = 20, m = 5, p1 = 0.9\rangle$ with varying tightness $p2$ (horizontal axis).

All the implementations of the specialized algorithms can share the same data structures for storing the lower bounds of each value. Then each algorithm performs its specific specializations but the lower bound is computed using all the information tables. This fact makes it possible to combine all the algorithms easily. FSRDS performs more searches than SRDS and SRDS performs more searches than LSRDS, so FSRDS is computationally more expensive than SRDS and SRDS is computationally more expensive than LSRDS. One could think of an hybrid strategy that would apply FSRDS in the first subproblems, when they are small, then continue with SRDS and finally apply LSRDS. The advantage that we have with this strategy is that the last executions of LSRDS have a tighter lower bounds for values of the first subproblems. This fact will

presumably increase its pruning capabilities.

3.5.2 Frequency assignment problems (FAP)

In Appendix A.2.3 we present the FAP benchmark . We have solved the five subinstances of CELAR instance 6 with the following hybrid strategy: we start solving subproblems with FSRDS. When we reach the subproblem of size 13 variables we switch to SRDS. When we reach the subproblem of size 17 variables we switch to LSRDS. Results are given in Table 3.15. Comparing with the same instances solved by SRDS, we observe an speed up of one order of magnitude in CPU time.

instance	variables	connectivity	optimal cost	Hybrid	SRDS(lim)	ORDS(lim)
SUB_0	16	0.47	159	$60.9s$	$20.7s$	$9.8s$
SUB_1	14	0.82	2669	$1.2 \times 10^3 s$	$8.3 \times 10^2 s$	$3.1 \times 10^2 s$
SUB_2	16	0.74	2746	$5.6 \times 10^3 s$	$4.5 \times 10^3 s$	$5.5 \times 10^2 s$
SUB_3	18	0.69	3079	$7.6 \times 10^3 s$	$7.6 \times 10^3 s$	$1.2 \times 10^4 s$
SUB_4	22	0.56	3230	$1.6 \times 10^4 s$	$1.5 \times 10^5 s$	-

Figure 3.15: Results on CELAR-6 subinstances for the hybrid version, limited SRDS and limited ORDS. The CPU time corresponds to a Pentium at 2.8GHZ machine with 1G of RAM.

Our results show a substantial decrement in CPU time, with respect to results reported in [Givry et al., 1997] and [Larrosa et al., 1999]. With respect to [Jégou and Terrioux, 2004] the comparison is machine dependent but we solved the biggest subinstance in one order of magnitude less time. The most competitive approach is [Koster et al., 1999] but we must say that it is not a search approach.

In Fig. 3.16 we show in detail the resolution of FAP subinstance SUB_4 with three algorithms: the mentioned hybrid strategy, SRDS(lim=17) and ORDS(lim=17). First column contains the number of variables of the current subproblem. The second and third columns correspond to solving time and type of specialization performed for the hybrid strategy. The fourth and fifth columns correspond to solving time and specialization performed by SRDS(lim=17). The sixth and seventh columns correspond to solving time and specialization performed by ORDS(lim=17). Then the three final columns are optimal cost of the subproblem, the maximum mean of the costs of the values for all future variables (that is $\max_{x_j \in F} \texttt{mean}_{a \in D_j} rds^i_{ja}$), the mean of the costs of the values of the top future variable (that is $\texttt{mean}_{a \in D_j} rds^i_{ja}$ being x_j the first future variable). From these table we extract two conclusions that motivate (1) the use of SRDS and (2) the used hybrid strategy:

1. The mean specialized cost of the first future variable is much higher than the optimal cost of the subproblem, that is $\texttt{mean}_{a \in D_j} rds^i_{ja} >> rds_i$. This

observation motivates the use of SRDS with respect to RDS, because the combination with $ic's$ will probably give as result a higher lower bound. In addition the specialized cost of every value will permit to prune them sooner.

2. When using FSRDS it can happen that the maximum mean of the specialized cost in one future variable is much higher than the mean cost of a the first future variable. In the execution this happens after the inclusion of variable 9 for example (observe the difference of max mean and mean top columns). This fact motivates the use of FSRDS as in SRDS the combination of ic's and rds is most probably done in the first future variable. In addition FSRDS has a better prunning capacity.

Those facts are corroborated in Fig. 3.17. FSRDS is more time costly at the first subproblem resolutions, but then it is rewarded with exponential savings in the resolution of latter subproblems. What intuitively happens is that FSRDS, as it specializes all values for all subproblems, computes more information of each subproblem so in latter resolutions has more accurate lower bounds for prunning values. We have observed that in this instance ORDS has a similar behavior than RDS. It solves the first 17 variables quicker than both the hybrid strategy and SRDS(lim) but after that is unable to solve the whole instance. The reason is that the lower bound table computed by ORDS is of less quality. In the CELAR instances the lower bound cost of a value can vary from 0 to 6000. This variability is the reason why a full specialized approach is more competitive because the combination of ic's and rds is more powerful and values are pruned in an early stage of the search. Random problems, for example, have a very low variability of costs.

Variable Ordering Sensitivity

The fact that RDS imposes a static/fixed variable ordering disables the possibility of using heuristics for dynamic variable ordering (DVO). This fact causes RDS algorithms to be extremely sensitive to the variable ordering. We have evaluated this fact by extracting a subinstance of 8 variables from a frequency assignment problem. We have then executed RDS for all possible 8! orderings. The result of this experiment is shown in Fig. 3.18. Time resolutions range from 0.001 to 170 seconds. In [Verfaillie et al., 1996] it is shown that best orders are the ones with minimum bandwidth (that is the maximum distance in terms of number of variables between two connected variables). We corroborate this fact as we see that the orders that spend more time have all a greater bandwidth. Nevertheless this measure is not sufficient as many instances with high bandwidth have also a low solving time.

There is a special case when we can enable DVO and that is at some deep parts of the search tree where the rds_i contribution is 0. In this case the order of the variables doesn't affect the rds_i contribution because it will be zero from all remaining future variables. To evaluate the importance of this feature we have executed RDS with DVO enabled and without it in random problems with

	Hybrid		SRDS(lim)		ORDS(lim)		cost		
vars	cpu time	algorithm	cpu time	algorithm	cpu time	algorithm	optimal	max mean	mean top
2..5	0.05s	FSRDS	0.01s	SRDS	0.00s	ORDS	0	0.00	0.00
6	0.07s	FSRDS	0.01s	SRDS	0.00s	ORDS	0	47.32	47.32
7	0.12s	FSRDS	0.01s	SRDS	0.00s	ORDS	100	147.39	102.82
8	0.26s	FSRDS	0.02s	SRDS	0.01s	ORDS	202	302.22	237.55
9	2.04s	FSRDS	0.09s	SRDS	0.02s	ORDS	323	2294.68	2294.68
10	2.98s	FSRDS	0.13s	SRDS	0.03s	ORDS	347	2324.81	619.56
11	57.11s	FSRDS	3.78s	SRDS	0.12s	ORDS	350	2391.68	469.05
12	250.2s	FSRDS	8.46s	SRDS	0.56s	ORDS	435	2557.10	849.68
13	2812.4s	FSRDS	36.41s	SRDS	11.3s	ORDS	1115	2982.77	1362.00
14	2868.4s	SRDS	137.37s	SRDS	72.3s	ORDS	1632	2982.77	1871.09
15	3056.4s	SRDS	477.03s	SRDS	273.3s	ORDS	2271	2982.77	2698.14
16	5616.7s	SRDS	4533.8s	SRDS	559.3s	ORDS	2746	3277.23	3277.23
17	6671.1s	SRDS	6425.4s	SRDS	3639s	ORDS	2858	3339.32	3339.32
18	7696.5s	LSRDS	7698.5s	LSRDS	12240s	LSRDS	3079	3565.52	3565.52
19	9335.5s	LSRDS	11015.5s	LSRDS	79718.0s	LSRDS	3109	3565.52	3109.00
20	9335.5s	LSRDS	11029.9s	LSRDS	79732.5s	LSRDS	3109	3565.52	3109.00
21	16860.0s	LSRDS	105223.4s	LSRDS	-	LSRDS	3230	3565.52	3240.00
22	16860.0s	LSRDS	105223.4s	LSRDS	-	LSRDS	3230	3565.52	3230.00

Figure 3.16: Results on FAP subinstance SUB_4.

low connectivity to increase the number of included variables with no RDS contribution. We didn't find much relevance in incorporating DVO, although the solving time was slightly reduced.

3.5.3 Combinatorial Auctions

In Appendix A.2.4 we describe the Combinatorial Auctions benchmark and how instances can be formulated as WCSPs. We have generated instances that have around 100 variables that represent bids. Our methods are not competitive with the current solving methods that have been applied to these instances, but our objective here was to test the relative performance of RDS, LSRDS, SRDS(lim=$n/2$) and SRDS. In Fig. 3.19 we show a table of the solved instances. Results show that an specialization with no limitation can be advantageous. LSRDS is always better than RDS, SRDS($lim = n/2$) is always better than LSRDS and SRDS with no limitation is always better than SRDS($lim = n/2$). SRDS is 10 times faster than RDS. The instances have unary constraints that assign very heterogenous costs. The domains are binary and one value it is assigned cost 0 and the other a cost that ranges from 400 to 1400. This is possibly the reason why SRDS with no limitation is so advantageous.

3.5.4 Earth Observation Satellite Management, Spot

In Appendix A.2.2 we describe the Spot instances. We have tested RDS, LSRDS and SRDS(lim=$n/2$) algorithms. We discard the use of FSRDS because spot instances have more than 100 variables and during search deep levels of the

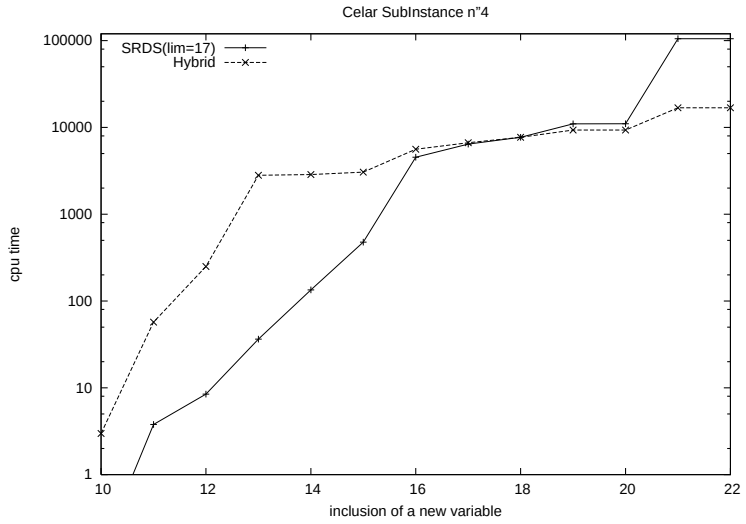


Figure 3.17: On the horizontal axis we show the sequence of subproblems $\mathcal{W}^i$. We compare SRDS(lim=17) and the hybrid strategy on SUB_4 subinstance

tree are never reached, so using FSRDS in the first subproblems would be of no advantage. ORDS has also a degenerate behavior when applied to big problem sizes, so for the same reason as FSRDS we discard its use.

The variable ordering used for the resolution is the same as the specified in the input file which corresponds to the photographic order (as suggested in [Verfaillie et al., 1996]). In Fig. 3.20 we show a table of the solved instances. LSRDS spends a factor of 0.4 less time than SRDS in the instances that were more costly to solve. Spot instances have unary soft constraints that assign very homogenous costs to all values (that ranges between 0 and 2) and in rare occasions a higher cost of 1000. These small variability on costs may be the reason why a SRDS(lim) does not show any advantage.

We extract the following conclusions from the experimental results. Versions that specialize all values for all subproblems (FSRDS and ORDS) can have a degenerate behavior when the problems increase in size, that's why we use its limited versions. LSRDS is always better than RDS. Random problems have values with homogenous costs and experiments show that a conservative strategy like ORDS is more efficient in this case. Only values that seem promising to increase the lower bound are specialized, and it happens often that the specializations are productive. The advantages of specializing values show up in problems where values have a high variability of costs, that is, when values are very heterogenous. So for example a full specialized approach like FSRDS com-

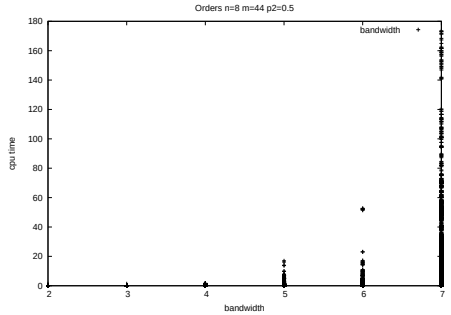


Figure 3.18: Solving time (vertical axis on the right) versus bandwidth (vertical axis on the left) of RDS executions for all possible orderings of an 8 variable problem extracted from CELAR subinstance SUB4.

	$\|X\|$	$\|C\|$	d	RDS	LSRDS	SRDS(lim)	SRDS	UB
auction01	102	1094	2	470.3	105.9	86.8	54.9	57431
auction02	101	742	2	>2000	1170.6	1155.3	867.3	60274
auction03	100	1184	2	1293.0	176.6	168.5	142.2	51968
auction04	100	937	2	427.2	92.7	86.4	69.3	69059
auction05	100	1033	2	523.2	84.2	77.65	47.5	66609
auction06	100	705	2	800.4	229.2	216.0	161.4	48848
auction07	103	1001	2	>2000	775.7	741.9	600.7	70929
auction08	100	746	2	1706.0	305.4	272.6	188.4	58167
auction09	100	999	2	343.3	59.2	55.2	37.9	64459

Figure 3.19: Results on combinatorial auctions instances. Columns: instance, number of variables, number of constraints, maximum domain size, time for RDS solving, LSRDS, SRDS($lim = n/2$), SRDS and the optimal cost.

	$\|X\|$	$\|C\|$	d	RDS	LSRDS	SRDS(lim)	UB
spot1502	209	203	2..4	0.015	0.18	0.063	28042
spot404	100	610	2..4	2.40	2.28	1.84	114
spot503	143	492	2..4	11.59	12.89	13.71	11113
spot408	200	2032	2..4	604.0	75.156	83.421	6228
spot505	240	2002	2..4	660.6	520.4	520.4	21253
spot412	300	4048	2..4	1187.9	239.5	265.0	32381
spot507	311	5421	2..4	>2000	1389	1802	27390
spot414	364	9744	2..4	>2000	>2000	>2000	
spot509	348	8276	2..4	>2000	>2000	>2000	

Figure 3.20: Results on spot instances. Columns: instance, number of variables, number of constraints, domain size, time for RDS solving, time for LSRDS solving, time for SRDS and the optimal cost.

bined with SRDS and LSRDS is very advantageous for the frequency assignment problems (FAP). The actions benchmark has also very heterogenous costs and

SRDS with no limitation has the best performance. The Spot benchmark confirms this hypothesis. Spot has more heterogenous costs and in this case LSRDS has a much better performance than RDS but SRDS does not pay-off.

3.6 Related Work

Russian Doll Search is an original and effective technique to take into account the cost that will certainly occur because of functions that link future variables during search. Other approaches exist for this purpose. PFC evolved to *Directed Arc consistency Counts* PFC-DAC [Wallace, 1995]. PFC-DAC adds *dac* cost counters that are similar to *ic*'s but they refer to constraints that link exclusively future variables. Analogously to *ic*'s, the minimum sum of *dac*'s for every variable can be taken into account in the lower bound. *Maintaining Reversible DAC* [Larrosa et al., 1999] updates and maintains *dac* counters during search (see Section B.1). Soft Arc Consistency techniques are a generalization of these techniques and are under recent development [Schiex, 2000, Larrosa and Schiex, 2004]. The interesting thing that we point out in the next section is that RDS and these techniques might not be exclusive. Russian Doll Search has a close relation to dynamic programming algorithms, like for example Bucket Elimination (BE) [Dechter, 1999]. BE solves the problem without search but as a counterpart it needs exponential memory space. The main difference between dynamic programming algorithms and RDS is that the latter computes a certain information of the current subproblem that is not sufficient to guarantee that the next subproblem will be solved without search. Even with the whole table of optimal costs that FSRDS computes we cannot guarantee that search is not needed. The advantage is that RDS remains a search algorithm and it has linear space complexity.

3.7 Perspectives of future work

3.7.1 An exact RDS

RDS-like algorithms have the flavor of dynamic programming, but are not dynamic programming algorithms. In fact, they can be seen as degraded versions of dynamic programming, due to the fact that they compute a limited amount of information of every solved subproblem. In the case of RDS a 0 arity constraint, SRDS a unary constraint and FSRDS a unary constraint for every variable. In all cases this information is not sufficient to solve the next subproblems in polynomial time and exponential search is needed again.

In a complete version of dynamic programming, following the behavior of RDS, it would be necessary to compute the optimal cost of including each combination of values for the variables of $\mathcal{W}^i$ that are connected to variables outside $\mathcal{W}^i$ (that is $x_n, ..., x_{i+1}$). So would imply in fact to compute constraints of arity greater than one as the described versions of RDS do. Depending on how variables may be connected this fact obliges to perform an exponential number of

searches. This is closer to a complete inference algorithm like Bucket Elimination than to a search algorithm. This is one of the reasons why the second part of this thesis is devoted to such algorithms.

3.7.2 RDS with two parallel searches

One of the main RDS drawbacks is the existence of unproductive searches that obliges us to optimally solve every subproblem, even if the addition of a variable didn't increase the optimal cost. It is clear that solving more subproblems can be disadvantageous if subproblems are easy and it would have been faster to directly solve the whole problem. This is proven by the experimental results where low connected problems are solved faster with simple PFC. To tackle this issue, some work has been done to explore variations on the number of variables being included in every subproblem at one time [Tounsi and David, 2002]. This problem is even more important when we increase the number of searches as we do in SRDS and FSRDS. In Section 3.4 we addressed the problem performing the resolutions that where found promising, that could be able to increment the lower bound. We describe in the following an alternative idea. We could launch two collaborative searches:

- One process would be devoted to solve all the problem assigning variables in the RDS static order. We call this process global.
- Another process would proceed as usual RDS solving sequences of increasing subproblems.

Both process can share the *rds* computations: a vector in the case of RDS and tables for SRDS and FSRDS. Then if the RDS process improves a lower bound that can be useful to the global process because it is assigning variables in a deeper level of the search they can send to each other a message notifying this fact. Then the global process could check if the pruning condition is satisfied with this new lower bound. If the global process finishes the search it can stop the other collaborative process. It seems that this is a significant characteristic that doesn't simply mean dividing by two the resolution process, but gets rid of one of the main drawbacks of RDS and can lead to exponential gains. This approach can also be very helpful for performing what we call opportunistic specialization in the sense that extra specialization can be done in parallel.

3.7.3 Combining Soft Arc Consistency and RDS

RDS is not the only way of adding the contribution of future variables into the lower bound computation; directed arc consistency counts (DAC, [Larrosa et al., 1999]) and more recently developments in soft arc consistency (SoftAC, [Schiex, 2000] which generalizes DAC) are other ways of doing so. The first feeling that one has is that they are exclusive with RDS as they may refer to the same constraints, the same cost would be counted twice and so leading to an incorrect lower bound. Both DAC and SoftAC contribute to the lower

bound recording the costs that will certainly occur because of the future binary constraints. To detect costs coming from higher implicit dependencies one would be obliged to enforce more powerful forms of soft consistency as soft path consistency or soft 3-consistency. These stronger forms of soft arc consistency are not used in practice mainly because of its high temporal complexity. When a problem is made Soft Arc Consistent, some costs are propagated down to unary constraints (usually called c_i) and others down to a zero arity constraint (usually called $c_\emptyset$) which is in fact a lower bound of the problem. So RDS techniques may be useful to consider costs coming from non binary arity constraints to contribute to the lower bound. This idea is only sketched here but seems promising for further development in the future.

3.7.4 Generalizing RDS to arbitrary relaxations

Any type of simplification of a problem can be used to extract lower bound information of it. For this reason it is reasonable to think that any type of simplification can potentially be used to build up a RDS type of algorithm. By RDS type of algorithm we mean that previous resolutions are reused latter in other resolutions to contribute to the lower bound.

Methods for simplifying the problem (also called relaxing in many contexts) and obtaining lower bounds on the total cost have been explored in [Cabon et al., 1998, Givry et al., 1997] mostly from a search point of view. In the latter the most successful combination to obtain anytime lower bounds was found to be an iterative deepening RDS.

Two lower bounding methods that seem promising to be used in conjunction with RDS based algorithms: removing constraints of the problem, merging values into a single value. 1) One can think of instead of adding one variable, at each time adding a function at each time. 2) In addition to the RDS way of solving subproblems, we could add more resolutions starting from merged values. An advantage of doing so is that we would dispose of global upper bounds sooner than RDS. These ideas need to be explored carefully in both cases because the inclusion of a variable at each resolution remains the main step of RDS as it is oriented to search and in search main step is to assign a variable.

3.8 Conclusions

Two enhancements of the RDS lower bound have produced algorithms Specialized RDS (SRDS), Full Specialized RDS (FSRDS) and Opportunistic RDS (ORDS). A special case of SRDS the limited version LSRDS has the same computational effort than RDS but it has always an equal or superior lower bound. The other specialized versions SRDS, FSRDS and ORDS have proven to be more efficient than RDS if a good combination of them is used and specialization is applied to certain problem sizes. The main idea of specialization consists in computing not only the optimal cost of a subproblem but the optimal cost of certain values of the variables of the subproblem. In that way the RDS special-

ized contribution can be combined with the inconsistency costs and in general provides a superior lower bound than RDS. For RDS-based approaches, the cost of solving a subproblem increases exponentially with the problem size. RDS is able to solve quickly small subproblems, but it slows down dramatically when subproblems increase to a reasonable size. All specialized versions can perform worst than RDS in small subproblems but its superior lower bound allows for a better performance with increasing size.

Chapter 4

Pseudo-Tree Search

In CSP and WCSP, constraints specify interaction among variables. Such interaction is usually represented by means of the so-called constraint graph (see Def. 2.3). It is well known that the structure of the constraint graph can be exploited. For instance, if the constraint graph is formed by independent connected components, each component can be solved independently. If the constraint graph is acyclic, there are specific efficient algorithms to solve it.

All the search algorithms presented so far do not take advantage of the structure. Thus, their worst case complexity is exponential on the number of variables no matter how the graph is structured. In this Chapter, we show how search algorithms can be adapted to exploit the structure. In particular, we extend the idea of pseudo-tree to WCSP. Such idea, already used in the CSP context [Freuder and Quinn, 1985, Bayardo and Miranker, 1995], allows to dynamically detect independent subproblems as search is conducted. As we show, the complexity of algorithms based on pseudo-trees is exponential on the tree height h, so it is bounded by this structural parameter. The space complexity is polynomial.

Our contribution is the extension of pseudo-tree search to WCSP. We show that the same general principles apply in this context. However, the use of pseudo-trees for WCSP has a source of inefficiency: in each independent subproblem the distance between the upper and the lower bound increases which has an adverse effect on pruning. We show that one way to overcome such problem is to introduce local subproblem lower and upper bounds. We combine pseudo-tree search with RDS techniques explained in Chapter 3 which adapt nicely. We obtain then high quality local lower and upper bounds [Larrosa et al., 2002].

Complete Inference algorithms have time and space complexities exponential in a problem structural parameter. With respect to search, they have the disadvantage that space is also exponential and algorithms quickly exhaust memory resources if a tractable decomposition is not found. Pseudo-tree search is a first step on getting closer to the nice properties of Complete Inference algorithms with respect to the structure, while preserving the polynomial space complexity of search algorithms.

4.1 Pseudo-tree

Definition 4.1 [**pseudo-tree**] *A pseudo-tree arrangement of a constraint graph is a rooted tree having as nodes the set of variables. It satisfies the property that adjacent vertices from the graph belong to the same branch of the tree.*

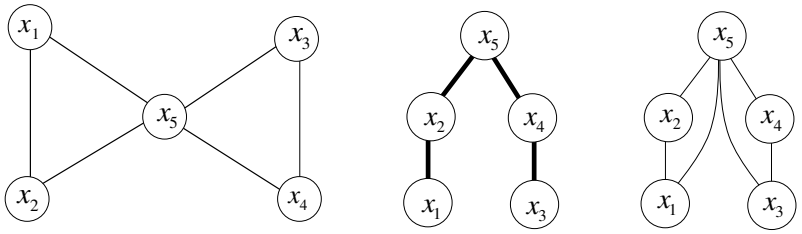


Figure 4.1: Left: a constraint graph. Middle: a possible pseudo-tree for it. Right: a pseudo-tree arrangement of the original constraint graph.

In Fig. 4.1 we show a constraint graph, a pseudo-tree and a pseudo-tree arrangement of the original constraint graph. We call h the height of the tree. In Chapter 3 we defined subproblems with respect to an order of variables (see Def. 3.9). In this Chapter we define a subproblem rooted at one variable of the pseudo-tree arrangement.

Definition 4.2 [**subproblem rooted at**] *Let $\wp = \langle X, D, C\rangle$ be a CSP arranged as a pseudo-tree. The subproblem rooted at x_i denoted $\wp^i = \langle X^i, D^i, C^i\rangle$ is the one induced by all variables in the subtree rooted at x_i. The set of variables X^i is ordered depth-first starting in the root $X^i = \{x_i, ...\}$.*

Consider the CSP and its pseudo-tree arrangement of Fig. 4.1. The subproblem rooted at x_2, that is $\wp^2$, has $X^2 = \{x_2, x_1\}$, $D^2 = \{D_2, D_1\}$ and $C^2 = \{c_{21}\}$.

The induced width of a constraint graph (see 2.2.2) is in relation width the pseudo-tree height. If a graph has induced width w^{opt} then, we can find a pseudo-tree of height less than $w^{opt}\log(n)$, being n the number of variables [Darwiche, 2001, Bayardo and Miranker, 1995].

4.2 CSP pseudo-tree search

Consider a CSP $\wp = \langle X, D, C\rangle$ whose variables can be partitioned in three sets $X = \{x_i\} \cup X^j \cup X^k$ and no constraint connects variables of X^j and X^k. The CSP can be arranged as a pseudo-tree where x_j and x_k are children of x_i. This arrangement appears in Fig. 4.2.

We assign variables from top to down in depth-first order. We first assign x_i and then the subproblems $\wp^j$ and $\wp^k$ become independent as no constraint connects them. We can then solve the subproblems independently. The same idea can be applied recursively to each subproblem.

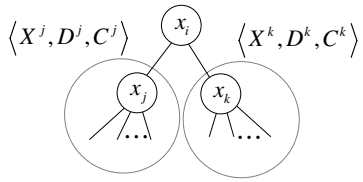
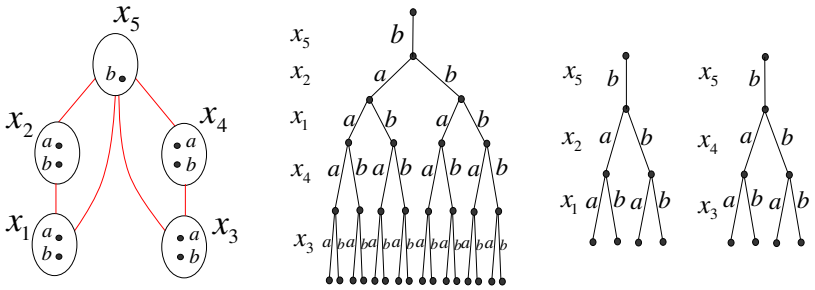


Figure 4.2: A problem instance arranged as a pseudo-tree. When x_i is assigned it can be divided into two subproblems $\langle X^j, D^j, C^j \rangle$ and $\langle X^k, D^k, C^k \rangle$.

Example 4.1 *Consider the CSP in Fig. 4.1 whose variables can be partitioned in three sets $X = \{x_5\} \cup X^2 \cup X^4$. After assigning variable x_5 the set of future variables is $F = X^2 \cup X^4$. $\wp$ is now separable into two subproblems: $\wp^2$ and $\wp^4$. Now it is sufficient to find a consistent extension of the current assignment in $\wp^2$ and $\wp^4$. If there is no consistent extension into $\wp^2$, we can backtrack to the following assignment of x_5 independently of $\wp^4$, because it is a necessary condition to have a consistent assignment in every independent subproblem.*

Solving the subproblems independently we get rid of the multiplicative effect of solving them as if they were a single problem. This fact can be formalized comparing the size of the associated search trees in both cases. If we don't exploit subproblem independence, the size of the search tree is exponential in the number of variables. If we exploit subproblem independence, the search tree is exponential with respect to the height of the pseudo-tree. We have now rather an additive effect that sums the sizes of the search trees associated to different subproblems instead of multiplying them. See the following example.

Example 4.2 *Find below on the left side a CSP instance arranged as a pseudo-tree. The domain of x_5 is $\{b\}$, all other domains are $\{a, b\}$.*



In the middle a search tree developed without exploiting subproblem independence. When we traverse this search tree we will visit in the worst case $1 + 2 + 2^2 + 2^3 + 2^4 = 31$ nodes. On the right, the two expanded search trees by the two independent resolutions of $\wp^2$ and $\wp^4$. The total number of nodes that we will visit in the worst case are the ones from the $\wp^2$ resolution $1 + 2 + 2^2 = 7$. The same for subproblem $\wp^4$, so the total number of nodes is now $7 + 7 = 14$.

The algorithm that we present in the next Section applies the previous ideas recursively in a pseudo-tree arrangement and is also a generalization to an arbitrary number of subproblems.

4.2.1 The CSP pseudo-tree algorithm

In Fig. 4.3 we present *Pseudo-tree Forward Checking* (PT-FC). It is the adaptation of FC (see Section 2.2.1) to the pseudo-tree idea. It differs from the algorithm presented in [Bayardo and Miranker, 1995] in that it incorporates look-ahead. It has three input parameters: the current assignment t, the set of future variables F and the collection of domains D. The algorithm is called initially PT-FC($\{\}, X, D$). PT-FC returns true if there exists a consistent assignment of variables in X.

PT-FC assigns variables in depth-first order according to a pseudo-tree arrangement. It starts from the pseudo-tree root (line 2). PT-FC iterates over all possible values of the current variable. If the current variable x_i has children in the pseudo-tree ($\texttt{succ}(x_i)$ returns the children of x_i), the current problem can be divided into $|\texttt{succ}(x_i)|$ independent subproblems, which are solved independently. The problem has a solution iff every independent subproblem has a solution. PT-FC first performs look-ahead in the subproblem rooted at x_i (line 4). If no empty domain is found, *sol* gets the value true, and false otherwise. Then we iterate over all the subproblems (line 6). A solution must exists in all subproblems. PT-FC is called to solve each subproblem if a solution exists (line 7). If a solution was found for every subproblem then it returns *true* and abandons search (line 8). In other case it tries another value for x_i. Every time PT-FC reaches a search state where the set of future variables is empty then the current assignment t assigns all variables in a branch of the arrangement (line 1), *true* is returned.

```
function PT-FC(t, F, D)
1 if F = ∅ then return true
2 x_i ← get-var(F)
3 for each a ∈ D_i do
4   D' ← look-ahead(x_i, a, t, X^i, D^i)
5   sol ← (∅ ∉ D')
6   for each x_j ∈ succ(x_i) do
7     if sol then sol ← PT-FC(t · ⟨x_i, a⟩, X^j, D')
8   if sol then return true
9 return false
```

Figure 4.3: Pseudo-Tree Forward Checking algorithm for CSP.

Pseudo-tree search has time complexity $O(l\ exp(h))$, where l and h are the number of leaves and the height of the pseudo-tree, respectively. Pseudo-tree search is polynomial in space.

4.3 WCSP pseudo-tree search

The same subproblem definition 4.2 applies to WCSP as it is defined with respect to the constraint graph. The only difference between CSP and WCSP is that constraints are cost functions. Accordingly, we denote $\mathcal{W}^i = \langle X^i, D^i, C^i \rangle$ the subproblem rooted at x_i.

Again consider the WCSP in Fig. 4.2 whose set of variables can be partitioned into three sets $X = \{x_i\} \cup X^j \cup X^k$ and $\texttt{succ}(x_i) = \{x_j, x_k\}$ such that no constraint connects variables from X^j and X^k. Consider a search state where t is the current assignment and $var(t) = \{x_i\}$.

The following strategy can be applied to a pseudo-tree arrangement. In WCSP, an assignment t has an associated $\texttt{cost}(t)$ which is the sum of costs assigned by constraints that all of its variables are assigned. The consistency of an assignment is defined with respect to an upper bound. After the assignment of t, $\mathcal{W}$ is separable into two subproblems: $\mathcal{W}^j$ and $\mathcal{W}^k$ that can be solved independently. After solving them, the optimal cost of extending t to both subproblems can be trivially computed. Suppose we first solve $\mathcal{W}^j$ and we obtain cost $\texttt{ub}^j$ and after we solve $\mathcal{W}^k$ and obtain $\texttt{ub}^k$. Then the optimal cost of extending t to variables X^j and X^k is: $\texttt{ub}^j + \texttt{ub}^k - \texttt{cost}(t)$. Since $\mathcal{W}^j$ and $\mathcal{W}^k$ have no constraints between them, the optimal cost of $\mathcal{W}^i$ after the assignment of t is the optimal cost of extending t to $\mathcal{W}^j$ plus the optimal cost of extending t to $\mathcal{W}^k$, and $\texttt{cost}(t)$ needs to be subtracted because it has been counted twice.

4.3.1 The basic pseudo-tree search algorithm

In Fig. 4.4 we present the simplest version of *Pseudo-Tree Partial Forward Checking* (PT-PFC-basic). It is the adaptation of PFC algorithm (see Section 2.4.1) to pseudo-tree search. It has four input parameters: the current assignment t, the set of future variables F, the collection of domains D and the upper bound $\texttt{ub}$. It is initially called PT-PFC-basic($\{\}, X, D, \texttt{ub}$). If the set F is empty, the result is trivially computed (line 1). PT-PFC-basic assumes that variables are selected depth-first according to a pseudo-tree arrangement (line 2). PT-PFC-basic selects a variable x_i and iterates over its values (lines 3). When value a is assigned we do look-ahead in the subtree below (line 4). Then the global current lower bound can be computed (line 5). If it is less than the upper bound we can prune values and check if the empty domain is found (lines 6 and 7). If not, for all $x_j \in \texttt{succ}(x_i)$ each $\mathcal{W}^j$ is solved (line 10). $\texttt{succ}(x_i)$ returns the set of variables that are direct children of x_i. For each resolution an optimal cost is obtained that is summed to the total optimal cost $\texttt{lb}$ (line 10). $\texttt{cost}(t)$ must be subtracted because it is initially included in line 8. If this total cost is better than the current upper bound we update it (line 11).

With respect to CSP pseudo-tree search, PT-PFC-basic incorporates lower and upper bounds. The search tree has to be explored completely in the worst case so the time complexity of PT-PFC-basic remains $O(l\ exp(h))$, where l and h are the number of leaves and the height of the pseudo-tree, respectively. PT-PFC-basic is polynomial in space.

```
function PT-PFC-basic(t, F, D, ub)
1  if F = ∅ then return min{ub, cost(t)}
2  x_i ← get-var(F)
3  for each a ∈ D_i do
4    look-ahead(i, a, t, X^i, D^i, ub)
5    if LB^PFC(t · ⟨x_i, a⟩, F, D) < ub then
6      D' ←prune(i, a, t, X^i, D^i, ub)
7      if ∅ ∉ D' then
8        lb ← cost(t)
9        for each x_j ∈ succ(x_i) do
10         lb ← lb+ PT-PFC-basic(t · ⟨x_i, a⟩, X^j, D', ub) - cost(t)
11       if lb < ub then ub ← lb
12 return ub
```

Figure 4.4: Basic Pseudo-Tree Partial Forward Checking algorithm.

Although PT-PFC-basic has an exponential time complexity bounded by the height of the pseudo-tree and PFC is exponential in the number of variables, PT-PFC-basic has a source of inefficiency. The reason is that PT-PFC-basic disposes of bad quality upper bounds when starting solving subproblems. Each independent resolution (each recursive call to PT-PFC-basic) is started with the upper bound of the whole problem. The local task of solving one subproblem is a simpler task but we use the upper bound of the whole problem to solve it. This fact increases the distance between its lower bound and the used upper bound. As a consequence, pruning becomes unlikely. To overcome this problem we introduce local upper and lower bounds.

4.3.2 Refinement of pseudo-tree search algorithm

Definition 4.3 [**local** lb, ub] *Let $\mathcal{W}$ be a WCSP. Consider a pseudo-tree search state where $t = \{..., \langle x_i, a\rangle\}$ is the current assignment and x_i the last assigned variable. We denote* $\texttt{lb}^i$ *a lower bound of the cost of extending t to variables in $\mathcal{W}^i$. We denote* $\texttt{ub}^i$ *an upper bound of the optimal cost of extending t to variables in $\mathcal{W}^i$.*

After the resolution of the subproblem $\mathcal{W}^i$, $\texttt{ub}^i$ and $\texttt{lb}^i$ coincide and are the optimal cost of extending t into the subproblem below. In this case, we use he suffix *opt* to indicate its optimality. We detail three uses of local lower and upper bounds that enhance PT-PFC-basic performance.

Again consider the WCSP arranged as a pseudo-tree in Fig. 4.5 whose variables can be partitioned in three sets $X = \{x_i\} \cup X^j \cup X^k$ and no constraint connects variables of X^j and X^k. The current assignment is $t = \{\langle x_i, a\rangle\}$ and $\texttt{succ}(x_i) = \{x_j, x_k\}$.

1) We start solving $\mathcal{W}^j$. Aiming at efficiency, we want $\texttt{ub}^j$ as low as possible, in order to decrease the difference with the lower bound which is essential for

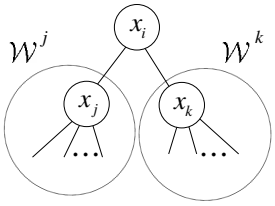


Figure 4.5: A problem instance arranged as a pseudo-tree. When x_i is assigned it can be divided into two subproblems $\langle X^j, D^j, C^j \rangle$ and $\langle X^k, D^k, C^k \rangle$.

pruning. The simplest idea is to use $\texttt{ub}^j = \texttt{ub}$, the maximum acceptable cost for the whole problem. A better approach is to compute $\texttt{lb}^k$, a lower bound of the cost of solving $\mathcal{W}^k$. Then we set $\texttt{ub}^j = \texttt{ub} - (\texttt{lb} - \texttt{lb}^k)$. $\texttt{lb}^k$ can be subtracted because it refers to subproblem $\mathcal{W}^k$.

2) This approach may still be too weak when $\texttt{lb}^k$ is a bad lower bound. One way to overcome this problem is to compute a lower local upper bound of the cost solution of $\mathcal{W}^j$ using for example local search. Let's consider that $\texttt{UB}^{\texttt{local}}(t, X^j)$ computes an upper bound of extending t to X^j variables then we can use $\texttt{ub}^j = \min\{\texttt{ub}^j, \texttt{UB}^{\texttt{local}}(t, X^j)\}$ when solving $\mathcal{W}^j$. We will describe in Section 4.4 an alternative way to compute a local upper bound.

3) After solving subproblem $\mathcal{W}^j$, $\texttt{lb}^j_{opt}$ is the optimal cost of extending t to this subproblem. Then, $\texttt{lb}^j$ is replaced by $\texttt{lb}^j_{opt}$ in all computations. So, when we start solving $\mathcal{W}^k$ we can set $\texttt{ub}^k = \texttt{ub} - (\texttt{lb} - \texttt{lb}^j_{opt})$. Again, we can compute a local upper bound and take the minimum between the two upper bounds, but now that we have the optimal solution of $\mathcal{W}^j$, it is more unlikely that we can improve over $\texttt{ub}^k$.

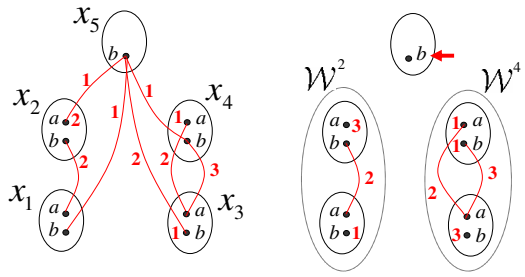
These local upper and lower bound computations can be easily generalized to an arbitrary number of subproblems and we incorporate them into algorithm PT-PFC-basic (see 4.4). We produce algorithm *Pseudo-tree Partial Forward Checking* (PT-PFC) which appears in Fig. 4.6. PT-PFC computes lb a lower bound of all subproblems (line 6). Then it starts solving each subproblem independently. A local lower bound $\texttt{lb}^j$ is computed (line 10) for each subproblem. Then a local upper bound is computed for the current subproblem $\texttt{ub}^j = \texttt{ub} - (\texttt{lb} - \texttt{lb}^j)$ (line 11). We can subtract from ub all the lower bounds of other subproblems except the lower bound of the current subproblem $\mathcal{W}^j$, that is $\texttt{lb} - \texttt{lb}^j$. The local upper bound may be enhanced in line 12 with local search or with the Russian Doll Search algorithms explained in next section. A backtrack condition is tested for local lower bounds and upper bound (line 13). If the lower bound $\texttt{lb}^j$ of the subproblem is greater than or equal to $\texttt{ub}^j$, the current subproblem does not need to be solved because either $\texttt{ub}^j$ is the solution, or there is no solution improving over $\texttt{ub}^j$ thus ub. If any of the tests has failed we can exit the loop and backtrack. If not we can proceed solving recursively assigning depth-first the next variable of the current subproblem. The cost that the recursive call returns may be better than the estimated lower bound that we computed for

```
function PT-PFC(t, F, D, ub)
1   if F = ∅ then return min{ub, cost(t)}
2   x_i ← get-first(F)
3   for each a ∈ D_i do
4     look-ahead(i, a, t, X^i, D^i, ub)
5     D' ← prune(i, a, t, X^i, D^i, ub)
6     lb ← LB^PFC(t · ⟨x_i, a⟩, F, D')
7     if lb < ub and ∅ ∉ D' then
8       for each x_j ∈ succ(x_i) do
9         if lb < ub then
10          lb^j ← LB^PFC(t · ⟨x_i, a⟩, X^j, D^j)
11          ub^j ← ub − (lb − lb^j)
12          ub^j ← min{ub^j, UB^local(t, X^j)}
13          if lb^j < ub^j then
14            lb ← lb − lb^j + PT-PFC(t · ⟨x_i, a⟩, X^j, D', ub^j)
15  if lb < ub then ub ← lb
16  return ub
```

Figure 4.6: Pseudo-Tree Partial Forward Checking algorithm.

that subproblem so it can directly be interchanged in the global lower bound (line 14). Once all independent subproblems have been solved, if lb is smaller than the global upper bound ub, a better solution has been found so ub is updated (line 13). After trying all feasible values of variable x_i, the cost of the best solution remains in ub, which is returned (line 14).

Example 4.3 *Find on the left side of the drawing that follows next a WCSP instance in which an arc appears when a combination of values has a cost greater than 0. Next to each value we show its inconsistency cost ic_{ia}. On the right, variable x_5 is assigned value b and its inconsistency costs propagated.*



After the assignment, both PT-PFC-basic and PT-PFC compute a global lower bound using the PFC lower bound function to underestimate the cost of extending the current assignment $t = \{\langle x_5, b\rangle\}$ to both subproblems $\mathcal{W}^2$ and $\mathcal{W}^4$. The computed lower bound is $\texttt{lb} = 0 + 0 + 1 + 0 = 1$*, the sum of minimum inconsistency costs in every variable. Consider the global upper bound is* $\texttt{ub} = 2$*. PT-PFC-basic solves both subproblems using this global upper bound* $\texttt{ub} = 2$*.*

PT-PFC computes the lower bound $\texttt{lb} = 0 + 1$ *(line 5). Then it solves each independent subproblem separately. First of all it checks the global condition* $\texttt{lb} < \texttt{ub}$ *(*$1 < 2$*). As it is satisfied computes the local lower and upper bounds* $\texttt{lb}^2 = 0$ *and* $\texttt{ub}^2 = 2 - (1 - 0) = 1$*. The local upper bound can then be improved by local search (line 10). It then checks the local condition* $\texttt{lb}^2 < \texttt{ub}^2$ *(*$0 < 1$*) which is also satisfied. When* $\mathcal{W}^2$ *resolution finishes (*$\texttt{lb}^2_{opt} = 1$ *is optimal) PT-PFC updates the current lower bound* $\texttt{lb} = \texttt{lb} - \texttt{lb}^2 + \texttt{lb}^2_{opt} = 1 - 0 + 1 = 2$*. At this point PT-PFC skips the resolution of* $\mathcal{W}^4$ *because the current lower bound has reached the upper bound.*

4.4 Combining Pseudo-Tree and Russian Doll Search

As outlined before, PT-PFC has still a source of inefficiency due to the fact that when we start a subproblem resolution we drag the uncertainty of the other subproblems that we leave behind. We showed that performance can be improved with good local lower and upper bounds but no particular method for improving them was developed. In this section we show how Russian Doll Search (Chapter 3) can be adapted to work with pseudo-trees and may provide effective local bounds.

The main source of inefficiency of PT-PFC may come from what we call the *uncertainty gap*. At a search state where some subproblems become independent, some may be already solved by a PT-PFC call, other are still unsolved, we associate an uncertainty gap to every unsolved subproblem. Let $\mathcal{W}^j$ and $\mathcal{W}^k$ be two unsolved subproblems. The only information that we dispose of $\mathcal{W}^k$ is its local lower bound $\texttt{lb}^k$. $\texttt{lb}^k$ is lower than the optimal $\texttt{lb}^k_{opt}$. When solving subproblem $\mathcal{W}^j$ we drag this difference $\texttt{lb}^k_{opt} - \texttt{lb}^k$ in the upper bound. The uncertainty gap is the necessary cost that our local upper bound will include when start solving a subproblem. The uncertainty gap cannot be known in advance because we would have to solve all subproblems first. As we solve subproblems depth-first uncertainty gaps are summed at each level.

Example 4.4 *Consider the WCSP in example 4.3 after the assignment* $t = \{\langle x_5, b\rangle\}$ *(left of the drawing).* $\texttt{lb}^4 = 1$ *because the sum of minimum inconsistency costs. But the optimal cost of solving* $\mathcal{W}^4$ *is in fact* $\texttt{lb}^4_{opt} = 3$*. So when we solve* $\mathcal{W}^2$ *we drag an uncertainty gap of* $\texttt{lb}^4_{opt} - \texttt{lb}^4 = 3 - 1$*.*

We now combine the pseudo-tree approach with Russian Doll Search (Chapter 3) with the aim of reducing the uncertainty gap caused by a bad quality upper bound when we start solving a particular independent subproblem. RDS provides two advantageous features to pseudo-tree search:

1. Good quality lower bounds of all independent subproblems. That is, RDS computes a better $\texttt{lb}^k$ to get closer to the optimal $\texttt{lb}^k_{opt}$.

2. Good quality local upper bounds of all independent subproblems. RDS allows to compute a better $\mathtt{ub}^j$ to get closer to the optimal $\mathtt{ub}^j_{opt}$.

RDS nicely adapts to pseudo-tree search. Let's recall that the idea of RDS is to replace one resolution by n successive resolutions on nested subproblems. So when solving a particular subproblem we can suppose that previous subproblems below have already been solved and reuse its optimal solving cost.

In the pseudo-tree context, nested subproblems are associated to the different subtrees of the pseudo-tree arrangement. For instance in Fig. 4.7 we have on top a constraint graph and on the right the nested resolutions that RDS does. On bottom the constraint graph is arranged as a pseudo-tree and on the right we have the resolutions that a pseudo-tree RDS does.

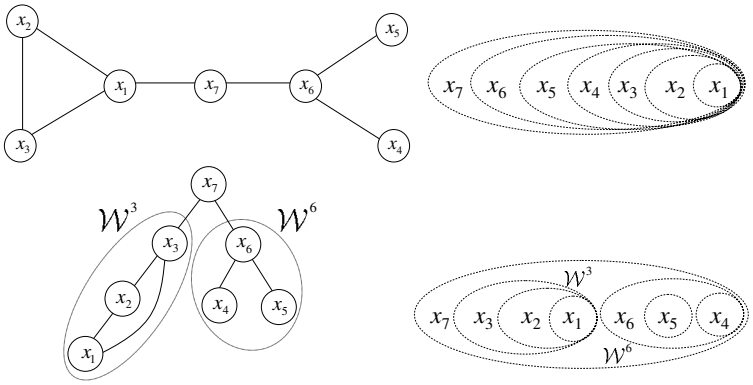


Figure 4.7: Top left: a constraint graph. Top right: the n resolutions that RDS performs. Bottom left: a possible pseudo-tree arrangement. Bottom right: the n resolutions that PT-RDS performs.

Now the optimal cost of solving subproblem $\mathcal{W}^i$, is rds_i. In Fig. 4.7 rds_6 would be the optimal cost of solving the problem including variables $X^6 = \{x_6, x_5, x_4\}$. We solve subproblems from bottom to top. In Fig. 4.7 this means that we first compute rds_1, rds_4 and rds_5. Then rds_2 and rds_6. Then rds_3. Finally rds_7, the whole problem.

When solving a subproblem rooted at x_i all subproblems below in the tree have been previously solved. This means that for all nodes below x_i in the pseudo-tree arrangement we know its optimal cost rds_i and we know also its optimal solution $Sol(\mathcal{W}^i)$. The optimal costs rds_i provides us good lower bounds of subproblems. $Sol(\mathcal{W}^i)$ is likely to be a good extension of any current assignment containing variables up to the parent variable of X_i, so we can use it to compute a local upper bound. Therefore, pseudo-tree RDS provides local lower and upper bounds.

Again consider the WCSP in Fig. 4.2 where the current assignment is $t = \{\langle x_i, a\rangle\}$ and $\mathtt{succ}(x_i) = \{x_j, x_k\}$. When we start assigning variables in X^j we

function PT-RDS($\mathcal{W}^i$)
1 **if** $|X^i| = 1$ **then** $rds_i \leftarrow 0$
2 **else**
3 **for each** $x_j \in \texttt{succ}(x_i)$ **do** PT-RDS($\mathcal{W}^j$)
4 $rds_i \leftarrow$ PT-PFC-RDS($\emptyset, X^i, D^i, \texttt{ub}^i$)
5 **return** rds_i

function $\texttt{LB}^{\texttt{PT-RDS}}(t, F)$
1 **return** $\texttt{cost}(t) + \sum_{x_j \in F, j \neq i} \min_{b \in D_j} \{ic_{jb}\} + \sum_{x_j \in \texttt{succ}(x_i)} rds_j$

function $\texttt{UB}^{\texttt{PT-RDS}}(t, X^i)$
1 **return** $\texttt{cost}(t \cup Sol(\mathcal{W}^i))$

Figure 4.8: Pseudo-Tree Russian Doll Search algorithm.

can subtract the lower bound of $\mathcal{W}^k$ that now includes the RDS contribution to the upper bound. Additionally we can check if $Sol(\mathcal{W}^j)$ is a good extension of the current assignment that improves the local upper bound. We call this algorithm *Pseudo-Tree* RDS (PT-RDS) and appears in Fig. 4.8. PT-PFC-RDS is obtained from PT-PFC (Fig. 4.6) replacing lower and upper bound functions by the corresponding ones appearing in Fig. 4.8. PT-RDS calls recursively PT-PFC-RDS for every node of the pseudo-tree from bottom to top.

4.4.1 Specializing Pseudo-Tree RDS Search

The concepts of SRDS (see Section 3.2) can be extended to the pseudo-tree context. Consider $\mathcal{W} = \langle X, D, C \rangle$ a WCSP and a pseudo-tree arrangement. $\mathcal{W}^{ia}$ is the subproblem rooted at x_i with respect to this pseudo-tree where the domain of variable x_i has been reduced to the singleton $\{a\}$. The optimal cost of solving subproblem $\mathcal{W}^{ia}$ is rds_{ia}.

Pseudo-Tree Specialized RDS (PT-SRDS) appears in Fig. 4.9. Lower bound functions have been modified because now if the current variable is x_i we can sum the ic's plus rds contribution at every variable belonging to $\texttt{succ}(x_i)$.

4.5 Experimentation

We have tested two algorithms SRDS and PT-SRDS to asses the effect of pseudo-tree search in three benchmarks: random problems, combinatorial auctions and spot instances.

function PT-SRDS($\mathcal{W}^i$)
1 $\forall_{i \in X} \forall_{a \in D_i} rds_{ia} \leftarrow 0$
2 **for each** $x_j \in$ children(x_i) **do**
3 PT-SRDS(X^j, D_i)
4 **for each** $a \in D_i$ **do**
5 $rds_{ia} \leftarrow$ PT-PFC-RDS($\{\}, x_i \cup X^j, \{a\} \cup D^j, \mathtt{ub}_{ia}$)

function PT-PFC-SRDS($t, F, D, \mathtt{ub}$)
1 **if** $F = \emptyset$ **then return** $\min\{\mathtt{ub}, \mathtt{cost}(t)\}$
2 $x_i \leftarrow$ get-first(F)
3 **for each** $a \in D_i$ **do**
4 **if** $\mathtt{LB}^{\mathtt{PFC}}(t \cdot \langle x_i, a\rangle, F, D) + rds_{ia} < \mathtt{ub}$ **then**
5 $D' \leftarrow$ look-ahead($x_i, a, t, X^i, D^i, \mathtt{ub}$)
6 $\mathtt{lb} \leftarrow \mathtt{LB}^{\mathtt{PT-SRDS}}(t \cdot \langle x_i, a\rangle, F, D')$
7 **for each** $x_j \in \mathtt{succ}(x_i)$ **do**
8 $\mathtt{lb}^j \leftarrow \mathtt{LB}^{\mathtt{PT-SRDS}}(t \cdot \langle x_i, a\rangle, X^j, D')$
9 $\mathtt{ub}^j \leftarrow \mathtt{ub} - (\mathtt{lb} - \mathtt{lb}^j)$
10 $\mathtt{ub}^j \leftarrow \min\{\mathtt{ub}^j, \mathtt{UB}^{\mathtt{PT-RDS}}(t \cdot \langle x_i, a\rangle, X^j)\}$
11 **if** $\mathtt{lb} < \mathtt{ub}$ **or** $\mathtt{lb}^j < \mathtt{ub}^j$ **then**
12 $\mathtt{lb} \leftarrow \mathtt{lb} - \mathtt{lb}^j + \mathtt{PT} - \mathtt{PFC} - \mathtt{SRDS}(t \cdot \langle x_i, a\rangle, X^j, D', \mathtt{ub}^j)$
13 **if** $\mathtt{lb} < \mathtt{ub}$ **then** $\mathtt{ub} \leftarrow \mathtt{lb}$
14 **return** ub

function $\mathtt{LB}^{\mathtt{PT-SRDS}}(t = \{..., \langle x_i, a\rangle\}, F, D)$
1 **return** $\mathtt{cost}(t) + \sum_{x_j \in \mathtt{succ}(x_i) \cap F} \min_{b \in D_j} \{rds_{jb} + ic_{jb}\} + \sum_{x_j \in F - \mathtt{succ}(x_i) - \{x_i\}} \min_{b \in D_j} \{ic_{jb}\}$

Figure 4.9: Pseudo-tree Specialized Russian Doll Search algorithm.

4.5.1 Random Problems

Random problems are described in detail in Appendix A.2.1. As we can generate problems with increasing connectivities they seem an adequate benchmark for this case study.

Pseudo-trees are computed using the maximum cardinality search (MCS) heuristic [R.E.Tarjan and M.Yannakakis, 1984]. We have used the implementation of Toolbar library [Toolbar, 2003] for computing a tree-decomposition and directly transforming it into a pseudo-tree. The transformation is direct. We first choose a root node for the tree decomposition which minimizes the height of the tree. Then we go from the root to the leaves eliminating any variable that has already appeared. Since the temporal complexity is exponential in the pseudo-tree height, we have recorded the average height of pseudo-trees used in the experiments.

Integrating RDS versions into pseudo-tree search

We discard the use of FSRDS (Section 3.3) and ORDS (Section 3.4) inside pseudo-tree search the main reason being its degenerate behavior for increasing problem sizes (see Section 3.5). SRDS seems a good approach for using in conjunction with pseudo-tree search. There is an implementation issue that can be incorporated to PT-SRDS (Section 4.4.1) that corroborates this fact. Consider that we are including a variable x_i that has several children. If we solve this variable for every value and for every subproblem we can skip the whole resolution of the subproblem $\mathcal{W}^i$ because its optimal cost can be computed by taking the minimum of the sum of the costs of including each value in every subproblem. This cannot be done with RDS nor LSRDS because they only store the minimum of each cost. Therefore, the sum of the minimums would only be a lower bound and not the optimal cost. So for the experiments we use always PT-SRDS. Comparisons are done by solving each problem with SRDS and PT-SRDS. SRDS uses a static variable ordering that heuristically combines degree and locality, to produce low bandwidth orderings. PT-SRDS follows the variable ordering existing in the pseudo-tree. All the variables that are placed in the same branch in between two nodes that have children, or in between a leaf and a node that has children are interchangeable in the pseudo-tree. This means that we can apply a min bandwidth heuristic to order the variables of the pseudo-tree by these groups of variables.

We have tested four classes of random problems: $\langle n = 20, m = 5, p2 = 0.7\rangle$, $\langle n = 30, m = 3, p2 = 0.7\rangle$, $\langle n = 45, m = 2, p2 = 0.7\rangle$ and $\langle n = 52, m = 2, p2 = 0.7\rangle$. Each class has an increasing number of variables. All classes were tested with increasing connectivity which is the fundamental parameter that directly influences the pseudo-tree height. In Fig. 4.10 and 4.11 we show the cpu time for all these classes of random problems. We also show the ratio between pseudo-tree height and total number of variables. When this ratio is equal to one means that the pseudo-tree has a single branch, so the techniques developed in this Chapter have no effect. With an increasing number of variables better pseudo-trees (less height) can be constructed, this fact is observed in the different plots. In the first class of random problems we have also tested algorithm PT-PFC which does not exploit the RDS lower bound. PT-PFC has a clear worst performance suggesting that is a good idea to combine PT-PFC and RDS. In the following random classes we skip the execution of PT-PFC. We observe also that PT-SRDS is better than SRDS below a connectivity of $p1 = 0.5$. Above this connectivity PT-SRDS shows in some cases a worst performance that may be due to overhead reasons or also because a different variable ordering is used. It can happen that the pseudo-tree has few branches and this fact forces the algorithm to use a different static order. With random problems of less than 20 variables, we observed a moderated advantage of PT-SRDS.

In Section 3.2.3 of previous Chapter we explained some upper bounding techniques that we used in RDS algorithms and turned out to be very effective reducing the cpu time in about half the time. PT-SRDS can also exploit the

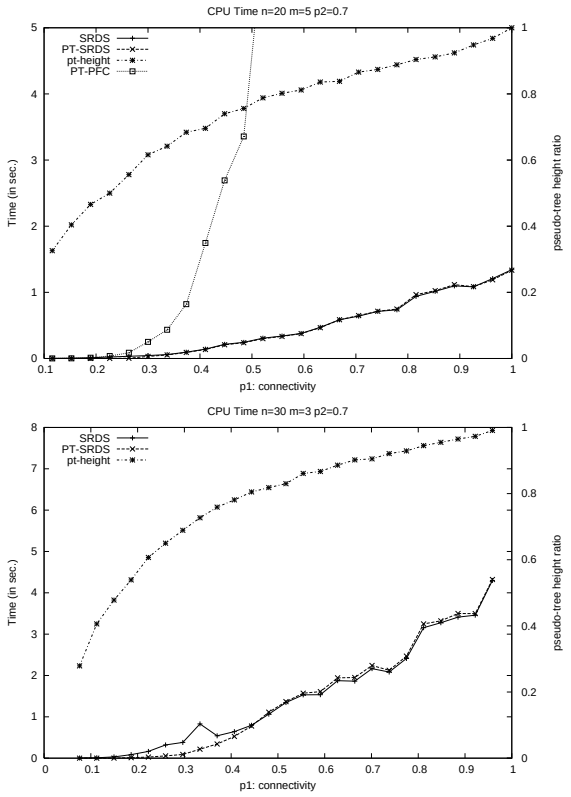


Figure 4.10: Average CPU time for two classes of random problems. The tested algorithms are SRDS and PT-SRDS and in the first plot also PT-PFC

explained upper bound adjustment but only in restricted occasions. We recall that the described upper bounding techniques try to improve the upper bounds of the values of the next variable that we will include every time that a better solution is found for the current subproblem. PT-SRDS can only adjust the upper bounds when it is assigning the last branch of the pseudo-tree so it is less effective.

We also have explored the relevance of local upper bounds in our implementation, by substituting line 10 of Fig. 4.9 by, $\mathtt{ub}_k \leftarrow \infty$. In this case, experimental results show that PT-SRDS looses performance. This result confirms that, although solving independent subproblems independently is an attractive strategy, working with the global upper bound is not cost-effective and the presence of local upper bounds is essential. Local upper bounds are of much importance in the first levels of the search.

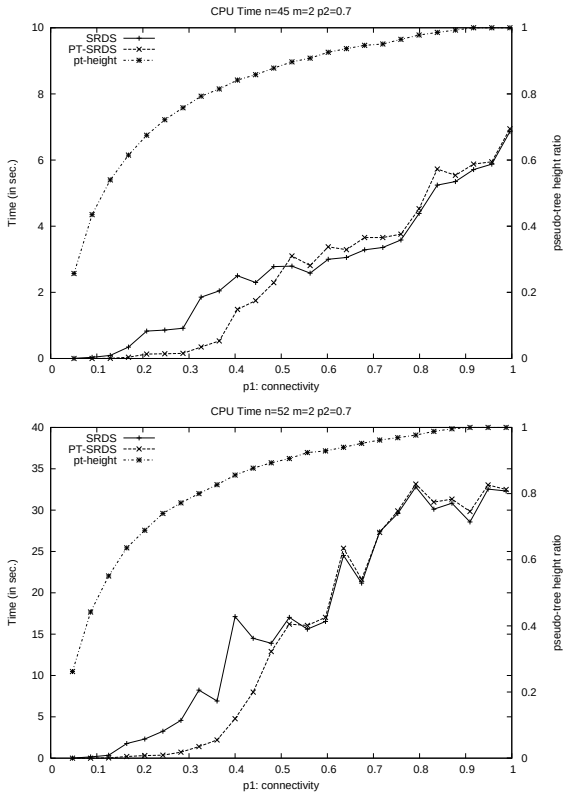


Figure 4.11: Average CPU time for two classes of random problems. The tested algorithms are SRDS and PT-SRDS and in the first plot also PT-PFC

4.5.2 Combinatorial Auctions

In Appendix A.2.4 we describe the Combinatorial Auctions benchmark and how instances can be formulated as WCSPs. In Section 3.19 we present results concerning Russian Doll Search algorithms. We compare here these results with algorithm PT-SRDS. In Fig. 4.12 we show the table of results. We have added connectivity and pseudo-tree height of every instance (columns fifth and sixth). All connectivities are very homogenous and low. Pseudo-tree height is nearly half the number of variables of the problem. PT-SRDS is always advantageous and, in some instances, it spends two orders of magnitude less time.

4.5.3 Earth Observation Satellite Management, Spot

In Appendix A.2.2 we describe the Spot instances. In Section 3.20 we present results concerning Russian Doll Search algorithms. We compare here these results with algorithm PT-SRDS. As in [Verfaillie et al., 1996] we choose the photo-

	$\|X\|$	$\|C\|$	d	$p1$	h	SRDS	PT-SRDS	UB
auction01	102	1094	2	0.21	61	54.0	15.58	57431
auction02	101	742	2	0.14	52	867.9	9.97	60274
auction03	100	1184	2	0.23	61	142.2	2.89	51968
auction04	100	937	2	0.18	62	69.3	15.6	69059
auction05	100	1033	2	0.20	62	47.5	30.0	66609
auction06	100	705	2	0.14	56	161.4	8.5	48848
auction07	103	1001	2	0.19	63	600.7	13.37	70929
auction08	100	746	2	0.15	53	188.4	20.23	58167
auction09	100	999	2	0.18	61	37.9	28.0	64459

Figure 4.12: Results on combinatorial auctions instances. Columns: instance, number of variables, number of constraints, maximum domain size, connectivity, pseudo-tree height, SRDS cpu time, PT-SRDS cpu time and optimal cost.

	$\|X\|$	$\|C\|$	d	$p1$	h	SRDS	PT-SRDS	UB
spot1502	209	203	3	0.007	18	0.18	0.09	28042
spot404	100	610	3	0.11	50	2.28	0.33	114
spot503	143	492	3	0.03	29	12.89	1.69	11113
spot408	200	2032	3	0.08	62	75.15	21.5	6228
spot505	240	2002	3	0.05	79	520.4	135.4	21253
spot412	300	4048	3	0.08	108	239.5	71.2	32381
spot507	311	5421	3	0.06	105	1389	301.3	27390
spot414	364	9744	3	0.07	145	>2000	1550.1	38478
spot509	348	8276	3	0.07	137	>2000	1297.7	36446

Figure 4.13: Results on spot instances. Columns: instance, number of variables, number of constraints, maximum domain size.

graphic order of variables, which corresponds to the same order in which variables appear in the input file. The photographic order is applied in each cluster of variables of a branch that has no bifurcations. Local upper bounds are computed from previous solutions of smaller subproblems that PT-SRDS has already computed. In the spot instances for every variable there is a value that indicates that the corresponding photo is not selected. When this value belongs to the optimal solution of a subproblem we also try the value with lowest weight for the local upper bound computation (also suggested in [Verfaillie et al., 1996]). In Fig. 4.13 we show the table of results. We have added connectivity and pseudo-tree height of every instance (columns fifth and sixth). All connectivities are very homogenous and extremely low. Pseudo-tree height is around 4 times less the number of variables. PT-SRDS is advantageous in all the tested instances. The gains in time are very heterogenous. Sometime we divide the CPU time by two, in other cases we reach gains of two orders of magnitude.

4.6 Related Work

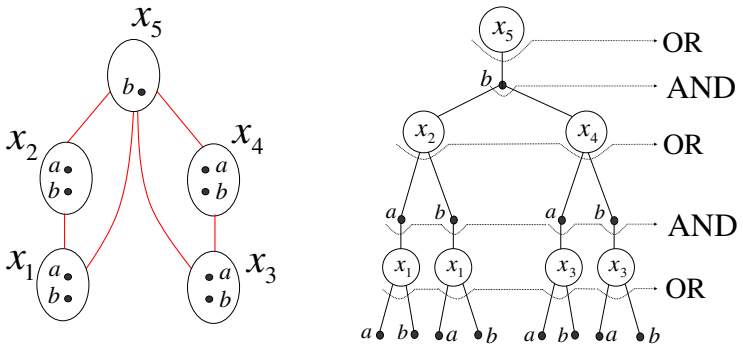
4.6.1 AND/OR trees

Recently pseudo-tree search methods have been reformulated in terms of the so called AND/OR Search spaces for the CSP case [Dechter and Mateescu, 2004] and WCSP [Marinescu and Dechter, 2005]. To our knowledge this approach is essentially equivalent to pseudo-tree search and has been developed independently to our work.

The usual search tree is also called OR search tree and does not capture any structure of the problem. Exploiting a pseudo-tree arrangement and identifying subproblems that can be solved independently during search compacts the usual search tree (see example 4.2). We associate an AND/OR tree to a pseudo tree arrangement. Independent subproblems are represented by AND nodes. So OR nodes represent alternative ways of solving the problem.

AND nodes states which usually represent problem decomposition into independent subproblems, all of which need to be solved. To build an AND/OR tree we label variables x_i as OR nodes and individual assignments $\langle x_i, a \rangle$ as AND nodes following a pseudo-tree.

Example 4.5 *Find on the left of drawing below a constraint graph arranged as a pseudo-tree. On the right the AND/OR search tree is expanded following the pseudo-tree. Variables are labeled OR nodes because a solution must exist in one of the branches below. Value assignment are labeled AND nodes because a solution must exist in every branch.*



AND/OR search traverses the AND/OR search tree depth-first. In CSP context when we reach an AND node a solution must exist in every branch below it. This is equivalent as saying that a solution must exists in every subproblem. OR nodes indicate that a solution must exist in at least one branch below. In WCSP context the same principle can be applied performing a sum of costs obtained in every branch in AND nodes and a minimum of all costs in OR nodes.

Recently in [Marinescu and Dechter, 2005] the AND/OR search tree is transformed into a graph. Nodes that are identified to be unifiable are fusioned. Thus

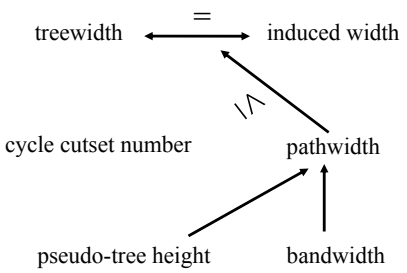


Figure 4.14: Decomposition parameters dominance diagram.

some nodes may have various parents, converting the tree into a graph. Then searching into an AND/OR graph can exploit what is called *caching*, a similar idea to good and nogood recording. Unifiable nodes can store the result of the first computation and then reuse it when the same node is reached again. This idea is also exploited in the belief networks [Darwiche, 2001, Pearl, 1988]. It is also similar to the idea of pseudo-tree adapted with Russian Doll Search techniques presented in Section 4.4, where we reuse the optimal cost of the subtree below inside the lower bound computation. The idea of caching has also been used in BTD algorithm [Jégou and Terrioux, 2004]. BTD exploits another type of decomposition called tree decomposition and it is outlined in the next section.

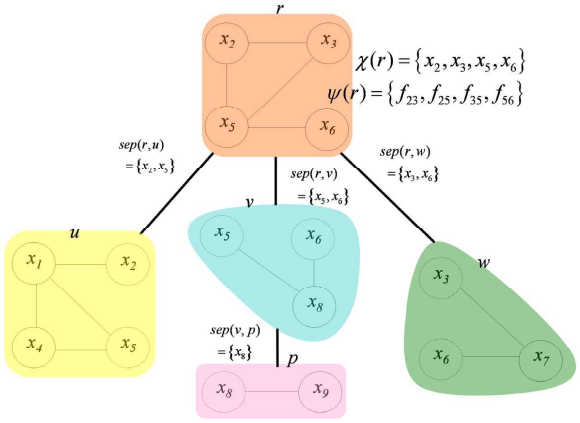
4.6.2 Other decomposition methods

Other decomposition methods exist based on different decomposition parameters. The cycle cutset number for example is the minimum number of vertices that must be eliminated from a constraint graph to convert it into a tree. When the constraint graph is a tree it can be solved in polynomial time and space. Cycle cutset is exploited inside MAC in [Sabin and Freuder, 1997]. Induced width was explained in Chapter 2 Section 2.2.2 and it is exploited to bound the complexity of Complete inference methods. The tree width is the decomposition parameter associated to a tree decomposition and it will be explained formally in Chapter 7 Section 7.2. A tree decomposition is a clustering of functions of a constraint graph in such a way that clusters from a tree. We call the separator of two clusters the common variables that they share. The tree width is the maximal number of variables in a cluster. The complexity of solving a problem instance using a tree decomposition is exponential with respect to the tree width. The tree width is equivalent to the induced width [Schiex, 1999]. It has been proven that tree-decomposition theoretically dominates all the other methods [Gottlob et al., 2000]. In Fig. 4.14 we show a dominance diagram of different parameters.

BTD algorithm

Recently tree decomposition has originated its corresponding search method [Jégou and Terrioux, 2004]. *Branch and Bound Tree Decomposition* search (BTD) assigns variables depth-first following a rooted tree decomposition. From a tree decomposition a pseudo-tree can be constructed directly. The main difference between such a pseudo-tree and a usual one is that variables that belong to clusters that are not adjacent in the tree decomposition aren't connected by any constraint if there is one variable that does not belong to the separator. This may not happen in a usual constructed pseudo-tree as variables from the top can be connected to bottom variables as long as variables in different branches aren't connected. BTD takes advantage of this fact. Consider a rooted tree decomposition where we assign variables in depth-first order. At some point of our assignments the problem may be divided into several subproblems (in example 4.6 after assigning variables of the root cluster), just like in pseudo-tree search. The difference is that now the only variables that can influence the subproblems are the ones in the separator of the two clusters. So the optimal cost of solving a subproblem given a particular instantiation of the separator is always the same. The idea instead of computing it several times is to store it after the first resolution. If a particular instantiation of the separator was already computed we don't need to enter that subproblem with that particular instantiation, it's optimal cost is already stored in a data structure.

Example 4.6 *Find on the drawing below a rooted tree decomposition, where functions belong to one single cluster and variables can be in several clusters.*



In this particular decomposition variables from the root cluster are not connected with variables of the cluster p. BTD assigns variables in the root cluster then continues depth-first assigning variables in cluster u that haven't been assigned yet. The assignment of variables x_1, x_4 in cluster r (variables that do not belong to the separator $sep(r, u) = \{x_2, x_5\}$) does not affect the cost of extending the current assignment to cluster u. Thus, when BTD returns from assigning cluster u, the obtained cost of extending the assignment can be stored for the assignment of variables in the separator. When BTD will solve again cluster u

with the same instantiation of variables in the separator, the resolution can be skipped and the cost retrieved from a data structure.

4.6.3 Backjumping

The introduced decomposition methods analyze the structure of the problem "a priori", as a preprocess. The different algorithms are able to exploit the structure during search after the preprocess. There is another way of exploiting the structure of the problem that can be considered as "a posteriori", during search. We are thinking of *Backjumping* (BJ) [Gaschnig, 1978]. When a dead-end is encountered during search, or when all the values of a variable have been exhausted, backjumping looks for the higher variable in the levels of the search tree where it can jump to. Usual backtracking algorithm always backtracks to the previous level of the search. *Graph Based* BJ [Dechter, 1990], for example, computes the variable where to backtrack based on the constraint graph. *Conflict directed* BJ improves BJ by following a more sophisticated jumping strategy that is based on the conflicts between variables [Prosser, 1993]. The relation between backjumping and decomposition methods has not been studied to our knowledge and seems a very interesting line of research.

4.7 Perspectives of future work

We propose two ways of extending our current work on WCSP pseudo-tree search.

We reach a state of the search where the problem can be divided in several subproblems. Up to now, the strategy that we have applied is to solve subproblems sequentially. Solving the subproblems sequentially has always one disadvantage: we have to chose a problem to solve in the first place. Another possibility would be to solve the different subproblems at the same time. The only advantageous way of solving subproblems simultaneously in WCSP is to compute increasing lower bounds of every subproblem. Doing so process can share a cost as soon as a better lower bound is discovered. A way of computing increasing lower bounds is to use iterative deepening.

The presented ideas, more precisely the introduction of local upper bounds and the adaptation with the RDS schema, can be extended to any other Branch and Bound based schema that works with other decomposition methods. One possible extension of the present work would be to enhance BTD algorithm (see Section 4.6.2) with the presented techniques. In this direction we found very promising the work in [de Givry et al., 2006] which mixes soft arc consistency techniques with tree decomposition methods.

4.8 Conclusions

It is a fundamental issue that search algorithms exploit the structure of the constraint graph to improve its solving efficiency. Pseudo-tree search is a well

known algorithm that exploits pseudo-tree decomposition for CSP with two nice properties: *i*) its time complexity is bounded by a structural parameter and *ii*) its space complexity is polynomial. In this Chapter we have extended Pseudo-tree search to WCSP, producing an algorithm that we call PT-PFC. We have identified that this algorithm has a source of inefficiency which is due to bad quality local lower and upper bounds of subproblems, an issue that we name uncertainty gap. Even if the theoretical time complexity is bounded by the pseudo height, in some cases usual PFC may perform better. To tackle this problem we extend PT-PFC to the Russian Doll Techniques presented in Chapter 3 and provide an elegant solution to the uncertainty gap problem. As result of this extension we produce two algorithms PT-RDS and PT-SRDS which we prove to have substantial advantages over PT-PFC.

Part II

Complete Inference

Chapter 5

ADC with factorized constraints

Complete Inference solves a CSP by performing a sequence of problem transformations, until the problem can be trivially solved and the set of solutions is obtained. An inference operation transforms the problem into an equivalent one (in the remaining set of variables), deducing implicit constraints (constraints that where implicit in the original problem formulation). The obtained problem is supposed to be smaller or easier to solve.

Adaptive Consistency (ADC) is the basic Complete Inference algorithm for solving CSP (see Section 2.2.2 of Chapter 2). ADC main operation is variable elimination. The idea is constructing a constraint that is the join of all constraints in which the eliminated variable participates. Then the new constraint can be added to the problem and the variable removed. ADC performs a sequence of variable eliminations until there is no variable left and the solution can be trivially obtained. ADC has time and space complexity exponential on the problem induced width (see Def. 2.10). When the induced width is not small, ADC cannot be used because it exhausts the available memory. In this Chapter we introduce a novel implementation of ADC that, in some cases is more efficient in space. Our algorithm is based on two ideas:

(*i*) Negative constraints: ADC stores constraints as sets of permitted tuples. However sometimes it may be more advantageous to store constraints as sets of forbidden tuples. Forbidden tuples are called nogoods. When a constraint is represented by a set of nogoods we call it *negative constraint*.

(*ii*) Factorized constraints: Sometimes constraints can be factorized (i.e. decomposed) into a set of smaller arity constraints.

We incorporate both ideas inside ADC. We first experiment with ADC working exclusively with negative constraints. We show that it is rarely more efficient than usual ADC. We then incorporate factorization. To factorize we propose an alternative way of eliminating a variable. Instead of generating a constraint that captures the join effect of the eliminated variable, we generate a set of con-

straints that have the same effect (forbids the same combinations) and does not mention the eliminated variable. This is done by generating a set of constraints expressed as sets of forbidden tuples. The new variable elimination process is particulary advantageous if the eliminated variable is linked to many but very loose constraints. The intuition is that, with the usual procedure, we generate a large constraint expressing all the permitted assignments. It may be a better option to record in that part of the problem the forbidden tuples instead of the permitted ones. So in that case it may be a better option to eliminate a variable building a set of negative constraints. [Sanchez et al., 2004b] We also show that this procedure of variable elimination can be specialized for binary domain variables to eliminate them in polynomial space.

5.1 ADC with negative constraints

ADC was explained in Chapter 2 Section 2.2.2. ADC performs a sequence of variable eliminations. A variable is eliminated joining the constraints where it appears and then projecting the variable out of the constraint. We consider how constraints are implemented and how it affects its join and projection out operations.

Constraint storage

A constraint c can be stored as a set containing all permitted tuples. The *size* of a constraint c, denoted $|c|$, is its number of permitted tuples. If the set is implemented as a hash table, $c(t)$ can be retrieved in constant time. In the following, we assume that constraints are stored as sets. Then, computing $c \Downarrow x_i$ has time complexity $O(|c|)$. Regarding the join of two constraints $c \bowtie v$, there are two basic ways to compute it: (i) iterate over all $t \in c$ and $t' \in v$ and see if they match, which has complexity $O(|c||v|)$, and (ii) compute every tuple t over $var(c) \cup var(v)$ and retrieve the $c(t)$ and $v(t)$ values, which has complexity $O(exp(|var(c) \cup var(v)|))$. Since one can choose the best option beforehand, the cost is $O(\min\{|c||v|, exp(|var(c) \cup var(v)|)\})$. Observe that the cost of previous operations depends on the size of constraints.

Negative constraints

Usually, it is assumed that ADC stores constraints in positive form, that is, it stores the set of permitted tuples. However, when constraints are very loose this is not the best option. An alternative idea is to use constraints in negative form, that is, storing the set of forbidden tuples. We note c^- a constraint c stored in negative form: if $t \in c^-$, t is forbidden by the constraint. A positive constraint can be negated $\neg c = c^- = \{t \mid t \notin c\}$. Double negation produces the original set of tuples. Join and projection definitions can be easily extended to deal with negative constraints:

$$c^- \bowtie r^- = \{t \cdot t' \mid t \in c^- \vee t' \in r^-\}$$

$$c^- \Downarrow_{x_i} = \{t \mid \forall a \in D_i, t \cdot \langle x_i, a \rangle \in c^-\}$$

Consequently, ADC can be redefined to work exclusively with negative constraints. Fig. 5.1 presents this new version, called ADC$^-$. C^- denotes a set of constraints in negative form. ADC$^-$ is like standard ADC but original as well as intermediate constraints are assumed to be negative. The only difference with respect to ADC is that negative form join and projection is used.

function `Var-Elim`$^-(x_i, C^-)$
1 $B^- \leftarrow \{s^- \in C^- | x_i \in var(s^-)\}$
2 $c^- \leftarrow (\bowtie_{s^- \in B^-} s^-) \Downarrow x_i$
3 $C^- \leftarrow C^- \cup \{c^-\} - B^-$
4 **return** C^-

function `ADC`$^-(\wp)$
1 $X \leftarrow$ elimination-order(X)
2 **for each** $x_i \in X$ **do**
3 $\quad C^- \leftarrow$ `Var-Elim`$^-(x_i, C^-)$
4 **return** C^-

Figure 5.1: Negative Adaptive Consistency pseudo-code.

ADC$^-$ has also space and time complexity exponential with respect to the induced width of the problem. Although both ADC and ADC$^-$ have the same complexities, the question is which one is more efficient in practice. We have observed that it depends on the problem to be solved. Problems with loose constraints are better solved with ADC$^-$ because ADC generates large positive constraints. The opposite occurs for problems with tight constraints. This can be observed in Fig. **??** (left), which shows the relative performance of ADC and ADC$^-$ with respect to the number of tuples generated, for the random binary class $\langle n = 7, m = 5, p_1 = 1 \rangle$ and varying tightness.

5.2 Factoring negative constraints

Definition 5.1 [**factorize**] *A constraint c is factorized (also called decomposed) into a set of constraint C if* $\cup_{s \in C} var(s) = var(c)$ *and,*

$$c = \bowtie_{s \in C} s$$

In words we say that a constraint c is decomposed into a set of constraints C if the join of all constraints in C has the same scope as c and permits the same tuples as c. Using the stronger relation introduced in Chapter 2 we could say that a constraint c is decomposed into a set of constraints C, if $\{c\} \preccurlyeq C$ and $C \preccurlyeq \{c\}$, thus in fact if $\{c\} \approx C$. In addition, as the stronger relation is defined for the common set of variables of both sides we must impose that the union of scopes of constraints in C is the same as c.

Both, positive and negative constraints can be factorized in theory. However in this Chapter we consider only factorization of negative constraints.

Next we introduce a new implementation of negative variable elimination, `Var-Elim`$^-$, which works in practice more efficiently than both ADC and ADC$^-$.

It exploits the fact that nogoods can be factorized and small arity nogoods are very powerful because they forbid many assignments. As a preliminary step, we propose two new operations on constraints. The first operation is *projecting out a variable with memory.*

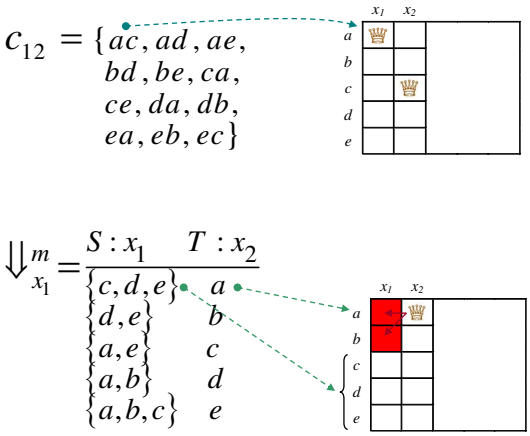
Definition 5.2 [**projecting out with memory**]. *Given a positive constraint c and $x_i \in var(c)$, the result of projecting out x_i from c with memory, denoted $c^i = c \Downarrow^m_{x_i}$, is a set of pairs $\langle S, T\rangle$ where S is a non-empty subset of D_i and T is a non-empty subset of tuples of $c \Downarrow x_i$. If $t \in T$, S is said to be its support set. By definition, if $t \in T$ then the support set S is,*

$$S = \{a \in D_i \mid t \cdot \langle x_i, a\rangle \in c\}$$

In words, the union of all sets T are the tuples of the usual projection. Tuples are grouped by their supporting set of values in the eliminated variable. Observe that sets T are mutually disjoint, although this is not necessarily the case for sets S.

Projecting out a variable with memory differs from standard projection in that it *remembers* for each tuple those supporting values in D_i. For convenience in subsequent computations tuples with the same support set are grouped together. Note that c^i is a constraint with memory about x_i. Constraints with memory are always in positive form. $var(c^i)$ returns the scope of c^i which does not include x_i.

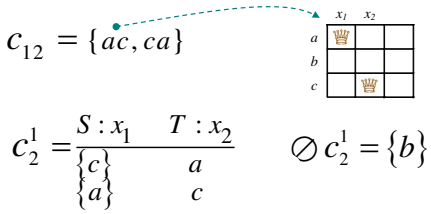
Example 5.1 *Find at the top of the drawing below constraint c_{12} from the 5-Queens problem. The valid tuple $\{\langle x_1, a\rangle, \langle x_2, c\rangle\}$ is shown in the 5-Queens board on the right.*



The projection with memory $c_{12} \Downarrow^m_{x_1} = c^1_{12}$ is shown above. Each row of the table is a pair $\langle S, T\rangle$. The tuple $\{\langle x_2, a\rangle\}$ and its support set S in x_1 are indicated on the 5-Queens board.

The second operation is *inferring nogoods* from constraints with memory. Inferring nogoods can be applied to a single constraint with memory c^i, noted $\oslash c^i$, or to a pair of constraints with memory, noted $c^i \oslash v^i$. In both cases it returns a negative constraint. When applied to a single constraint it returns the tuples that do not appear in c^i. It returns the set of tuples t of c that have no support in x_i. See the following example,

Example 5.2 *Find at the top of the drawing below constraint c_{12} from the 3-Queens problem. The valid tuple $\{\langle x_1, a\rangle, \langle x_2, c\rangle\}$ is shown in the 3-Queens board on the right.*



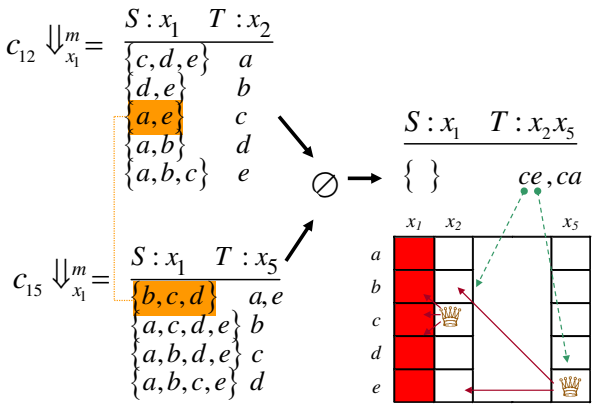
The projection with memory $c_{12} \Downarrow^m_{x_1} = c_2^1$ and its inferred nogoods $\oslash c_2^1$ are shown above. The tuple $\{\langle x_2, b\rangle\}$ is a nogood discovered by inferred nogoods operation as it has no supporting value in domain of x_1.

Definition 5.3 **[inferred nogoods]** *Given two constraints with memory of x_i, c^i and v^i, their inferred nogoods, denoted $c^i \oslash v^i$, is a set of tuples with scope $var(c^i) \cup var(v^i)$ defined as,*

$$\{t \cdot t' \mid \exists_{\langle S,T\rangle \in c^i} \exists_{\langle S',T'\rangle \in v^i} \quad t \in T,\ t' \in T',\ S \cap S' = \emptyset\}$$

In words, the inferred nogoods are those tuples that were permitted by either c^i and v^i but are not permitted by their join. Observe that the operation $\oslash$ produces a negative constraint.

Example 5.3 *Consider c_{12} and c_{15} two constraints of the 5-Queens CSP problem. See in the first column of the drawing below their projections with memory of x_1. On the right we show the result of applying inferred nogoods operation to the two projec*



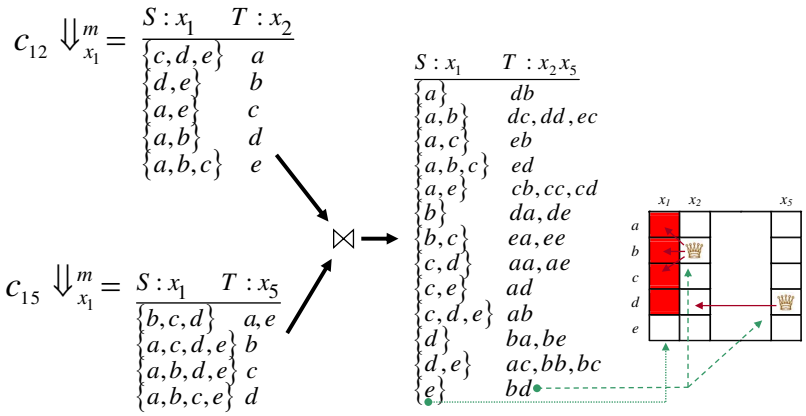
The discovered nogoods are $\{\langle x_2, c\rangle, \langle x_5, e\rangle\}$ *and* $\{\langle x_2, c\rangle, \langle x_5, a\rangle\}$ *(shown in the 5-Queens board). This is because there are only two support sets that have an empty intersection:* $\{a, e\}$ *of* c^1_{12} *and* $\{b, c, d\}$ *of* c^1_{15} *(highlighted in the drawing).*

The join operation can be extended to constraints with memory. Joining two constraints consists in combining the tuples which have non empty support sets intersections:

Definition 5.4 [join with memory] *Given two constraints with memory of* x_i, c^i *and* v^i, *their join with memory, denoted* $c^i \bowtie v^i$, *is another projection with memory defined as,*

$$\{\langle S, T\rangle \mid \exists_{\langle S', T'\rangle \in c^i}\ \exists_{\langle S'', T''\rangle \in v^i} \quad S = S' \cap S'' \neq \emptyset, \quad T = T' \bowtie T''\}$$

Example 5.4 *Consider* c_{12} *and* c_{15} *two constraints of the 5-Queens CSP problem. See in the in the first column of the drawing below their projections with memory of* x_1. *On the right the result of joining with memory both constraints, that is* $c^1_{25} = c^1_{12} \bowtie c^1_{15}$.



In the 5-Queens board on the right we show one tuple $\{\langle x_2, b\rangle, \langle x_5, d\rangle\}$ *and its support set of* x_1 $\{e\}$.

Property 5.1 $(c \bowtie c') \Downarrow^m_x = c \Downarrow^m_x \bowtie c' \Downarrow^m_x$.

Proof. Notation: $c \Downarrow^m_x = \{..., \langle S_i, T_i\rangle, ...\}, c' \Downarrow^m_x = \{..., \langle S'_j, T'_j\rangle, ...\}$
$c \Downarrow^m_x \bowtie c' \Downarrow^m_x = \{..., \langle S''_k, T''_k\rangle, ...\}, \quad (c \bowtie c') \Downarrow^m_x = \{..., \langle \mathcal{S}_l, \mathcal{T}_l\rangle, ...\}$

$\Rightarrow$) If $\tau \in \mathcal{T}_l$ and $a \in \mathcal{S}_l$, then $\tau \cdot \langle x, a\rangle \in c \bowtie c'$. We can write $\tau = t \cdot t'$, where $t = \tau[var(c) - \{x\}]$ and $t' = \tau[var(c') - \{x\}]$. Then, $t \in T_i$, $t' \in T'_j$, and $a \in S_i \cap S'_j = S''_k$. So, $\tau = t \cdot t' \in T''_k$ and $a \in S''_k$.

$\Leftarrow$) If $t'' \in T''_k$ and $a \in S''_k$, then $t'' = t \cdot t'$, with $t \in T_i$, $t' \in T'_j$, $a \in S_i \cap S'_j$. Then, $t'' \cdot \langle x, a\rangle \in c \bowtie c'$, so $t'' \in \mathcal{T}_l$ and $a \in \mathcal{S}_l$. □

5.2.1 ADC with negative factorized constraints

We present now a new implementation of ADC$^-$ which factorizes negative constraints. The first step is to redefine the variable elimination using inferred nogoods and projections with memory operations. We describe a new variable elimination that instead of building a single negative constraint returns an equivalent set of negative constraints that factorize it. Let us remember that the motivation of such a new variable elimination is to reduce the memory consumption, measured as total number of generated tuples.

In Fig. 5.2 we show the new algorithm that works with negative constraints and performs factoring. Function Var-Elim$^-_{factor}$ receives as input a variable and a set of negative constraints C^-. First the bucket of constraints mentioning x_i is constructed (line 1) and its projections with memory computed in (line 2). As the operation $\oslash$ discovers only new nogoods when applied to two constraints projected with memory, initial nogoods of all functions have to be extracted (line 3). Then, it joins constraints two by two (line 5,6) and extracts nogoods (line 7). In practice joining and extracting nogoods (lines 6,7) can be done at the same time. The negative constraint generated with the inferred nogoods is added to set N^- (line 7).

Property 5.2 *The set of constraints returned by* $\texttt{Var-Elim}^-_{factor}(i, C^-)$ *is equivalent to (forbids the same tuples as) standard variable elimination.*

Proof. Consider that we eliminate variable x_i. The set of constraints in which x_i appears is B. The constraint that usual variable elimination `Var-Elim` returns is $g = (\bowtie_{c \in B} c) \Downarrow x_i$. The set of negative constraints that $\texttt{Var-Elim}^-_{factor}$ returns is N^-. Applying property 5.1 we see that the unique constraint that remains in P when the while finishes (line 4) is g_i, with memory about x_i. At each point that two constraints c^i and v^i are joined, we extract in $c^i \oslash v^i$ those tuples which cannot belong to the join because there is no common support in the eliminated variable. Therefore, the set N^- already contains the tuples that are forbidden by the initial constraints, and contains the tuples that are discovered forbidden in the join process. So its union forbids the same tuples as g. □

function $\texttt{Var-Elim}^-_{factor}(x_i, C^-)$
1 $B^- \leftarrow \{c^- | c^- \in C^-, x_i \in var(c^-)\}$
2 $P \leftarrow \{(\neg c^-) \Downarrow^m_{x_i} | c^- \in C^-, x_i \in var(c^-)\}$
3 $N^- \leftarrow \{\oslash c | c \in P\}$
4 **while** $|P| > 1$ **do**
5 $\quad \{c^i, v^i\} \leftarrow \texttt{extract-two}(P)$
6 $\quad P \leftarrow P \cup (c^i \bowtie v^i)$
7 $\quad N^- \leftarrow N^- \cup (c^i \oslash v^i)$
8 **return** $C^- \cup N^- - B^-$

function $\texttt{ADC}^-_{factor}(\wp)$
1 $X \leftarrow$ elimination-order(X)
2 **for each** $x_i \in X$ **do**
3 $\quad C^- \leftarrow \texttt{Var-Elim}^-_{factor}(x_i, C^-)$
4 **return** C^-

Figure 5.2: Negative Adaptive Consistency with Factorization.

Example 5.5 *Let us consider the 3-Queens problem with the usual formulation. It has three constraints with the following permitted tuples,*

$$c_{12}(x_1, x_2) = \{ac, ca\}, \quad c_{13}(x_1, x_3) = \{ab, ba, bc, cb\}, \quad c_{23}(x_2, x_3) = \{ac, ca\}.$$

Standard variable elimination of x_1 first joins the positive constraint c_{12} and c_{13}, $c_{12} \bowtie c_{13} = \{acb, cab\}$, and then projecting out x_1 obtaining, $(c_{12} \bowtie c_{13}) \Downarrow x_1 = \{cb, ab\}$.

Next we proceed eliminating x_1 with $\texttt{Var-Elim}^-_{factor}(x_1, \{c^-_{12}, c^-_{13}, c^-_{23}\})$ Projecting out x_1 with memory we get,

$$c^1_2 = c_{12} \Downarrow^m_{x_1} (x_2) = \begin{array}{|cc|}\hline S : x_1 & T : x_2 \\ \hline \{a\} & c \\ \{c\} & a \\ \hline \end{array} \qquad c^1_3 = c_{13} \Downarrow^m_{x_1} (x_3) = \begin{array}{|cc|}\hline S : x_1 & T : x_3 \\ \hline \{a, c\} & b \\ \{b\} & a, c \\ \hline \end{array}$$

Then the projections with memory that have to be joined are $P = \{c^1_2, c^1_3\}$. As there are only two, only one iteration is needed to perform the variable elimination (line 4 of the algorithm). The join of both projections with memory is actually not needed. Inferred nogoods operation suffices. Original and inferred nogoods are,

$$n^-_2 = \oslash c^1_2 = \{b\} \qquad \oslash c^1_3 = \{\} \qquad n^-_{23} = c^1_2 \oslash c^1_3 = \begin{array}{l}\{ca, cc, \\ aa, ac\}\end{array}$$

These three negative constraints are equivalent to the generated positive constraint by usual variable elimination.

Fig. 5.2 presents ADC^-_{factor}. ADC^-_{factor} receives as input a constraint network $\langle X, D, C^- \rangle$, where C^- is a set of negative constraints. We substitute the variable elimination operation that generates a positive constraint by $\texttt{Var-Elim}^-_{factor}$. Variables are eliminated one by one. In ADC^-_{factor} the bucket of a variable is formed by negative constraints only, and the result of variable elimination is a new set of negative constraints. When processing a bucket, some of these constraints are turned positive (when projecting out the variable to eliminate).

Example 5.6 *Let us see in detail how the process works in the 4-Queens problem, with the usual formulation: columns are variables and rows are values a, b, c, d. The initial constraints of the problem are:*

$c_{12}(x_1, x_2) =$
$\{ac, ad, bd,$
$ca, da, db\}$

$c_{13}(x_1, x_3) =$
$\{ab, ad, ba, bc,$
$cb, cd, da, dc\}$

$c_{14}(x_1, x_4) =$
$\{ab, ac, ba, bc, bd,$
$ca, cb, cd, db, dc\}$

$c_{23}(x_2, x_3) =$
$\{ac, ad, bd,$
$ca, da, db\}$

$c_{24}(x_2, x_4) =$
$\{ab, ad, ba, bc,$
$cb, cd, da, dc\}$

$c_{34}(x_3, x_4) =$
$\{ac, ad, bd,$
$ca, da, db\}$

None of them generates original nogoods ($\oslash c^1_{ij} = \emptyset$). Elimination of x_1:

$$c_2^1 = \begin{array}{|cc|}\hline S:x_1 & T:x_2 \\ \hline \{a\} & c \\ \{a,b\} & d \\ \{c,d\} & a \\ \{d\} & b \\ \hline \end{array} \qquad c_3^1 = \begin{array}{|cc|}\hline S:x_1 & T:x_3 \\ \hline \{a,c\} & b,d \\ \{b,d\} & a,c \\ \hline \end{array} \qquad c_4^1 = \begin{array}{|cc|}\hline S:x_1 & T:x_4 \\ \hline \{a,b,d\} & c \\ \{a,c,d\} & b \\ \{b,c\} & a,d \\ \hline \end{array}$$

$$p_{23}^1 = c_2^1 \bowtie c_3^1 = \begin{array}{|cc|}\hline S:x_1 & T:x_2x_3 \\ \hline \{a\} & cb,cd,db,dd \\ \{b\} & da,dc \\ \{c\} & ab,ad \\ \{d\} & aa,ac,ba,bc \\ \hline \end{array} \qquad n_{23}^- = c_2^1 \oslash c_3^1 = \{bb,bd,ca,cc\}$$

$$n_{2,3,4}^- = p_{23}^1 \oslash c_4^1 = \{aaa,aad,abc,aca, acd,adc,baa,bad, bca,bcd,cba,cbd, cda,cdd,dab,dba, dbd,dcb,dda,ddd\}$$

Bucket $B_2 = \{c_{23}^-, c_{24}^-, n_{23}^-, n_{234}^-\}$. *Two constraints have the same scope,* c_{23}^- *and* n_{23}^-, *so we perform its join and leave the result in* c_{23}^-. *Now,* $B_2 = \{c_{23}^-, c_{24}^-, n_{234}^-\}$. *Elimination of* x_2*:*

$$p_3^2 = \begin{array}{|cc|}\hline S:x_2 & T:x_3 \\ \hline \{a\} & c,d \\ \{d\} & a,b \\ \hline \end{array} \qquad c_4^2 = \begin{array}{|cc|}\hline S:x_2 & T:x_4 \\ \hline \{a,c\} & b,d \\ \{b,d\} & a,c \\ \hline \end{array}$$

$$q_{34}^2 = \begin{array}{|cc|}\hline S:x_2 & T:x_3x_4 \\ \hline \{a,b\} & ba,da,bd,dd \\ \{a,b,c\} & ab,cb \\ \{b,c,d\} & bc,dc \\ \{c,d\} & aa,ca,ad,cd \\ \hline \end{array}$$

$$p_{34}^2 = p_3^2 \bowtie c_4^2 = \begin{array}{|cc|}\hline S:x_2 & T:x_3x_4 \\ \hline \{a\} & cb,cd,db,dd \\ \{d\} & aa,ac,ba,bc \\ \hline \end{array}$$

$$n_{34}^- = p_3^2 \oslash c_{24}^2 = \{ab,ad,bb,bd, ca,cc,da,dc\}$$

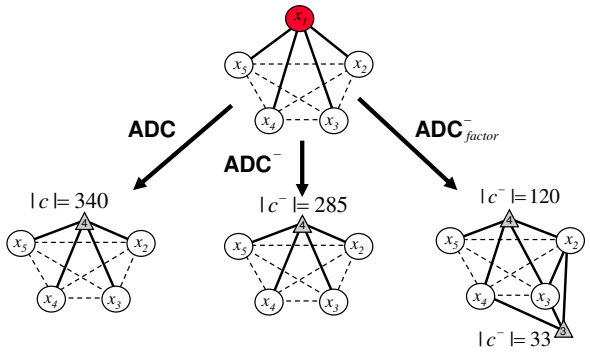
$$m_{34}^- = q_{34}^2 \oslash p_{34}^2 = \{ba,cd\}$$

Bucket $B_3 = \{c_{34}^-, n_{34}^-, m_{34}^-\}$. *The three constraints have the same scope, so we perform its join as their union, producing* n_{34}^-. *Now,* $B_3 = \{n_{34}^-\}$. *Elimination of* x_3*:*

$$p_{34}^3 = \begin{array}{|cc|}\hline S:x_3 & T:x_4 \\ \hline \{a\} & c \\ \{d\} & b \\ \hline \end{array}$$

The x_4 *variable is trivially eliminated. The problem has solution which can be obtained assigning variables in reverse order.*

Example 5.7 *Find on top of the drawing below the constraint graph of the 5-Queens problem. Underneath we show the effect of eliminating variable x_1 with algorithms ADC, ADC$^-$ and ADC$^-_{factor}$.*



On the left ADC constructs a constraint c with $|c| = 340$ tuples. In the middle we show the constraint c^- obtained by ADC$^-$ which has $|c^-| = 285$ tuples. If we sum positive tuples generated ADC plus negative tuples of ADC$^-$ we obtain the total number of tuples $5^4 = 625$. The constraint obtained by the elimination of the first variable is quite loose in the n-Queens problem. It can be seen as n grows that the number of generated negative constraints decreases with respect to positive ones. Finally on the right ADC$^-_{factor}$ eliminates x_1 and obtains two negative constraints $|c^-| = 120$ and $|c^-| = 33$.

5.2.2 Variable Elimination of binary domain variables

When a variable has binary domain and it is exclusively linked by binary constraints, it can be eliminated in polynomial time and space. It can be seen that this fact is analogous to what has been observed in the SAT field to applying a resolution rule when a variable only appears in binary clauses. In [Bacchus, 2002] the usual search algorithm for SAT, Davis and Putnam is enhanced with this rule. Observe that disjunctive clauses are like negative constraints as they forbid a particular combination of values: so for example $x_i \vee x_j$ is saying that the assignment $x_i \leftarrow false$, $x_j \leftarrow false$ is forbidden. Our result may be stronger than SAT as the eliminated variable can be linked by variables of arbitrary domain. In fact the procedure that we describe can also be generalized to a binary domain variable linked by constraints of any arity, but then the polynomial time and space is not guaranteed.

With the techniques developed in the previous section we can extend the resolution rule used for SAT into CSP. Suppose a variable x_i with a binary domain $D_i = \{a, b\}$ is only involved in binary constraints c_{ij}. If we express the effect of eliminating x_i with one positive constraint c (as ADC would do) it will contain large arity tuples. Similarly, if we express the effect of eliminating x_i with one negative constraint c^- (as ADC$^-$ would do) it will contain large arity

nogoods. However, these nogoods can always be factorized into nogoods of size less than or equal to two.

Property 5.3 *Consider variable x_i with domain $D_i = \{a, b\}$ linked only by binary constraints c_{ij}. x_i can be eliminated with a collection of nogoods of size at most 2 because larger arity nogoods can always be factorized by smaller ones.*

Proof. Let us suppose that there can exist a nogood of size three that cannot be factorized with smaller nogoods. This means that three variables linked to x_i by a binary constraint are the cause of a nogood that cannot be factorized by a nogood between any two of them or by any nogood of size one. As the variable x_i has a binary domain $D_i = \{a, b\}$ all projections with memory of these three binary constraints $c_{ij} \Downarrow_{x_i}^{m}$ are of the form $\langle\{a\}, T\rangle$, $\langle\{b\}, T\rangle$ $\langle\{a, b\}, T\rangle$; either one value supports a tuple either the other one, or both. Let us remember that a nogood is generated when an empty intersection of support sets is present. Tuples supported by both values can be discarded as they cannot generate an empty intersection, so are always factorized by smaller ones. We observe that if a ternary nogood exists then a ternary intersection of support sets must exist. So three support sets must generate an empty intersection. Support sets are of the form either $\{a\}$ or $\{b\}$. Thus at this point we have encountered a contradiction as it is not possible to generate an empty intersection of three support sets of this form. For example: $\{a\} \cap \{b\} \cap \{a\} = \emptyset$ but $\{a\} \cap \{b\}$ is also empty. Thus we conclude that our first supposition is false. It is not possible to generate a ternary nogood that is not factorized by a binary or unary one because all ternary empty intersections of supports sets have a smaller binary empty intersection or where already forbidden by a unary nogood. □

When domains are binary all nogoods can be factored into binary and unary no-goods. It is sufficient to compute all possible binary generated nogoods between every pair of projections with memory and we are done; the resulting set of negative constraints forbids exactly the same tuples as eliminating the variable with a usual procedure. When domains are ternary $D_i = \{a, b, c\}$ this does not happen as we can generate nogoods of arity 3. When domains are ternary supports sets can be of the form $\{ab\}$, $\{ac\}$, $\{bc\}$, $\{a\}$, $\{b\}$ and $\{c\}$. For example, $\{ab\} \cap \{ac\} \cap \{bc\} = \emptyset$ and has no smaller empty intersection.

Now, we aim at eliminating variable x_i producing an equivalent problem. The idea is that when a variable has a binary domain and it is linked only to binary constraints we can replace its effect generating all possible nogoods of size 1 and 2 of the variables it is linked to (all x_j such that $c_{ij} \in C$). All the generated unary and binary negative constraints, when joined together forbid the same tuples as the positive constraint that would generate a usual variable elimination.

An algorithm to implement such an elimination is shown in Fig. 5.3. It is based on the operations projection with memory and inferred nogoods defined in previous sections. It first generates all projections with memory mentioning variable x_i (set P^i of line 2). After generating all possible nogoods of size 1 (line 3) proceeds generating all new nogoods of size 2, that is all possible combinations two by two between the projections with memory in P^i and adding them to the

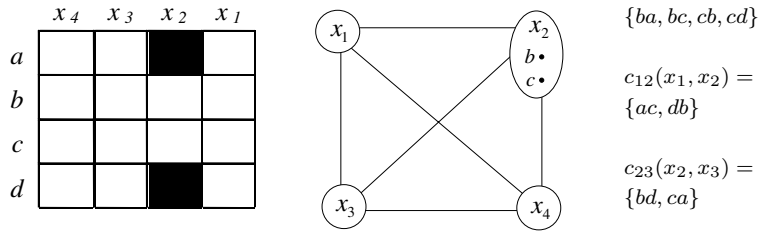
set N^- (line 6). At this point N^- forbids the same tuples as the constraint that a usual variable elimination would generate.

function `Bin-Var-Elim`$^-_{factor}(x_i, C^-)$
1 $B^- \leftarrow \{c^- | c^- \in C^-, x_i \in var(c^-)\}$
2 $P \leftarrow \{c^- \Downarrow^m_{x_i} | c^- \in C^-, x_i \in var(c^-)\}$
3 $N^- \leftarrow \{\oslash c | c \in P^i\}$
4 **for each** $c^i \in P$ **do**
5 **for each** $v^i \in P$, $v^i \neq c^i$ **do**
6 $N^- \leftarrow N^- \cup (c^i \oslash v^i)$
7 **return** $C^- \cup N^- - B^-$

Figure 5.3: Variable Elimination with negative factorized constraints for binary domain variables.

Algorithm `Bin-Var-Elim`$^-_{factor}$ has polynomial time and space complexities $O(n^2d^2)$. A variable is linked to a maximum of $n-1$ binary constraints. Loops in lines 4,5 take the projections with memory of these constraints two by two, so perform a maximum of $\binom{n-1}{2}$ iterations. Every negative constraint generated by the nogoods extraction operation has a maximum size d^2, being d the maximum domain size.

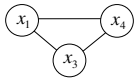
Example 5.8 *Consider the 4-Queens problem below where the the domain of x_2 is $D_2 = \{b, c\}$. Find in the the drawing below the board, its associated constraint graph and the set of constraints,*



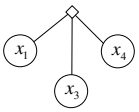
We eliminate x_2 using `Bin-Var-Elim`$^-_{factor}$. *We compute the projections with memory of constraints involving x_2:*

$$c_1^2 = \begin{array}{|c|c|}\hline S:x_2 & T:x_1 \\ \hline \{c\} & a \\ \{b\} & d \\ \hline \end{array} \quad c_3^2 = \begin{array}{|c|c|}\hline S:x_2 & T:x_3 \\ \hline \{c\} & a \\ \{b\} & d \\ \hline \end{array} \quad c_4^2 = \begin{array}{|c|c|}\hline S:x_2 & T:x_4 \\ \hline \{b\} & a,c \\ \{c\} & b,d \\ \hline \end{array}$$

If we generate all possible inferred nogoods we obtain the problem on the left, if we proceed usually summing all constraints and projecting, we obtain problem on the right:



$c_1^2 \oslash c_3^2 = \{ad, da\}$
$c_1^2 \oslash c_4^2 = \{aa, ac, db, dd\}$
$c_3^2 \oslash c_4^2 = \{aa, ac, db, dd\}$
$\oslash c_1^2 = \{b, c\}$
$\oslash c_3^2 = \{b, c\}$



$c_{24} \bowtie c_{23} \bowtie c_{12} = c_{4321} = \{adbd, cdbd, baca, daca\}$
$c_{4321} \Downarrow x_2 = \{add, cdd, baa, daa\}$

If we join all generated negative constraints after the inferred nogood it can be seen that is the exact negation of the constraint c_{4321} generated in a usual variable elimination.

Property 5.4 *Eliminating a binary domain variable with* `Bin-Var-Elim`$^-_{factor}$ *operation can have exponential saving in space with respect to* `Var-Elim`.

Proof. Imagine a binary variable x_0 with domain $D_0 = \{a, b\}$ linked to n variables of domain size d by n binary inequality constraints c_{0j} where $j = 1..n$. A usual variable elimination would join all the n constrains and then project x_0. Joining all constraints has as result a n arity positive constraint of size $2(d-1)^n$ that has exponential size. Let's look now at `Bin-Var-Elim`$^-_{factor}$. The unary inferred nogoods $\oslash c_{0j}$ produce no nogoods as we can always find a support for all values x_j. The projections with memory is of the form:

$$c_0^j = \begin{array}{cc} S : x_0 & T : x_j \\ \hline \{a\} & b, c, d, ... \\ \{b\} & a, c, d, ... \end{array}$$

The intersection between support sets $\{a\} \cap \{b\}$ produce an empty intersection, so the tuples $\{ba, bc, bd, ..., ca, cc, cd, ...\}$ are added as nogoods between any pair of variables x_j. The total size of these constraints is $\binom{n}{2}d^2$ which is polynomial. □

5.3 Experimental Evaluation

In the experiments we compare ADC, ADC$^-$ and ADC$^-_{factor}$. We focus in evaluating the memory that the algorithms spend by the successive variable eliminations. Time is also reported to justify that in any case the improvements in memory are substituted by an unacceptable amount of CPU time.

5.3.1 Random, SAT and n-queens problems

When eliminating a variable ADC$^-_{factor}$ adds a linear number of new constraints with respect to the size of the bucket. So the sequence of joins is an important decision. We have worked on several heuristics and two showed to bring substantial benefits: *minimum resulting arity* and *maximum expected nogoods generated*. The first one selects the two functions of smaller scope. The second one selects the two functions that have smaller support sets in its tables of memory projection, smaller supports are intended to produce more empty intersections.

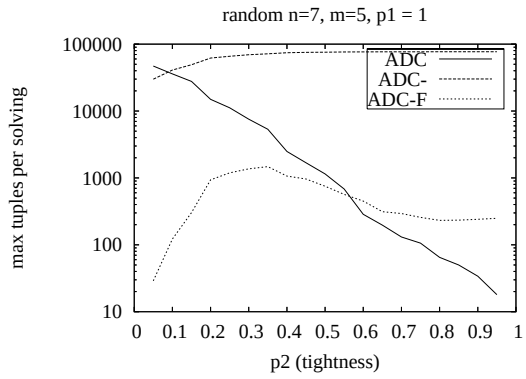


Figure 5.4: Results for the random binary class $\langle n = 7, m = 5, p_1 = 1 \rangle$.

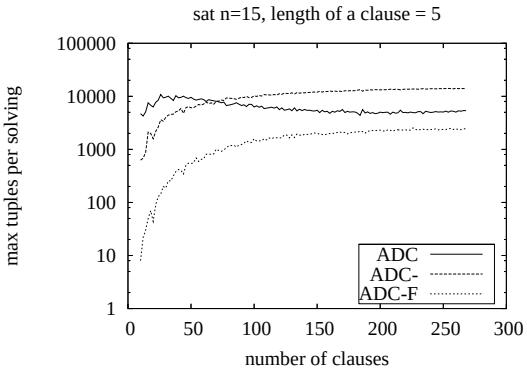


Figure 5.5: Results for 5-SAT instances.

Random problems are defined in Appendix A.1.1. With random problems the advantages of ADC^{-}_{factor} with respect to ADC can be controlled by the tightness and the connectivity of the generated problems. ADC^{-}_{factor} has its greater gain when the tightness is inferior to 0.5 and connectivity is close to 1. In that case, a gain of 3 orders of magnitude in memory consumption is reached. Connectivity is also an important parameter because combined with loose constraints can make a variable elimination very expensive for ADC. We observed in our experiments that no matter the connectivity of the graph, the positive representation is more advantageous when tightness is greater than 0.5 . This can be seen in Figure 5.4 on the left where ADC and ADC^{-}_{factor} lines cross.

Satisfiability benchmark is defined in Appendix A.1.2. An extreme case of loose constraints is the SAT problem modeled as a CSP using the model of one

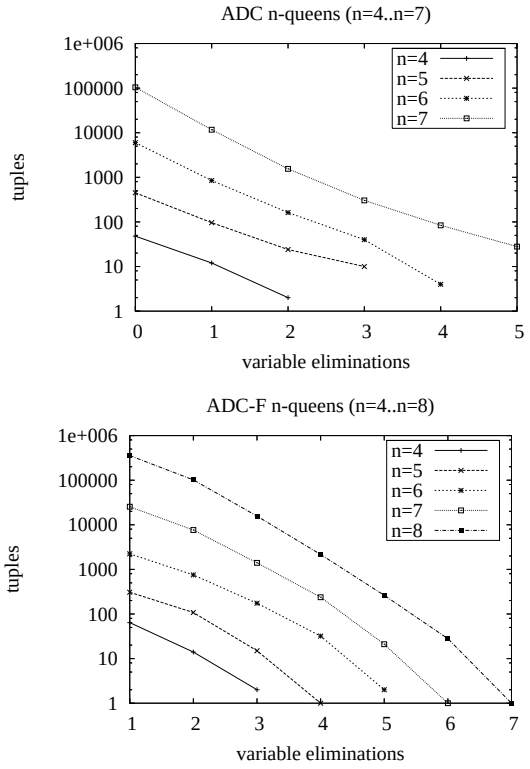


Figure 5.6: Number of tuples spent by ADC (on top) and ADC^{-}_{factor} (bottom) when solving different instances of the n-queens problem.

variable per logical variable, and each clause a constraint with a single negative tuple (the combination that forbids the clause) [Walsh, 2000]. In Figure 5.5 we can appreciate a gain of 3 to 1 orders of magnitude as the number of clauses grows (in this case the connectivity of the graph also grows). It is interesting to notice that ADC^{-}_{factor} maintains a constant gain of one order of magnitude even when the number of clauses grows, that is because there is a single forbidden tuple in every constraint.

In n-Queens problem (see example 2.1), at each elimination we have to build a constraint involving all the variables because each variable is related to all other variables. In this case, the heuristic *minimum arity* did not produce much benefits. The heuristic *minimum number of expected tuples* reduced considerably the number of tuples, a bit less than an order of magnitude. The factoring does not have any effect for small arity constraints because when n grows, nogoods also grow in size. For example, the smallest nogoods different from those contained in the original constraints for 8-queens are of arity 4. Figure 5 reports

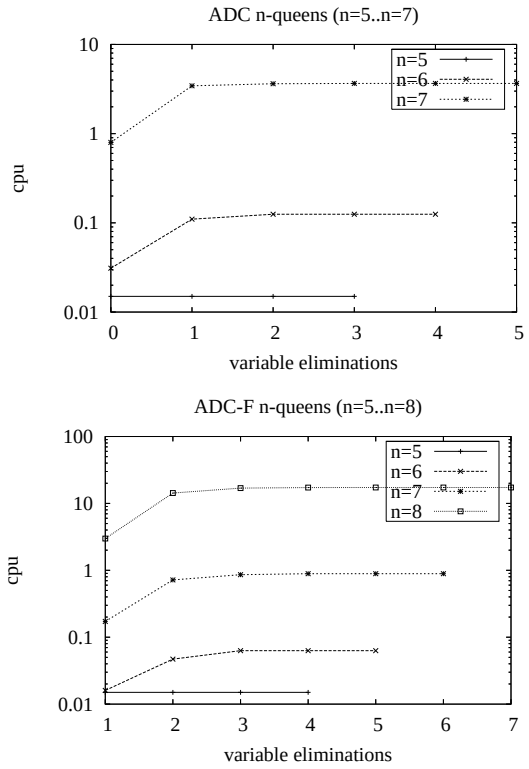


Figure 5.7: CPU time spent by ADC (on top) and ADC^-_{factor} (bottom) when solving different instances of the n-queens problem.

the results of this experiment. ADC^-_{factor} generates 6.93 times less tuples and is about 30 times faster. In the 8-queens the first elimination is the most expensive. We have tested all the instances until the program ran out of memory. ADC^-_{factor} could solve one more instance than ADC, that is shown in Fig. **??** as the line execution corresponding to $n = 8$ does not appear in the ADC plots.

5.3.2 Discussion

Although ADC is inefficient with loose constraints, ADC^- offers no improvements in practice. Only when constraints to join are really loose (for example in random problems when tightness is below 0.1), ADC^- improves over ADC. The intuitive reason is that, when ADC joins two constraints the resulting number of tuples is bounded by t^2. This is not the case for ADC^-, where the negative version of join generates an exponential number of tuples with respect to the non-common variables.

ADC^-_{factor} deals also with negative information. The difference with ADC^- is that at every two by two join it generates a negative constraint filled with nogoods that will in fact factorize other bigger nogoods that will not appear lately. ADC^-_{factor} can perform a linear number of those factorizations at every bucket. This set of factorized negative constraints when joined together are equal to the negation of the positive constraint that ADC would generate. Imagine a variable linked by many loose constraints. ADC generates a single constraint containing all the allowed combinations. Instead, ADC^-_{factor} generates a set of negative constraints that contain the factorized forbidden combinations. For example, if a single value is forbidden at any moment it only appears once inside a unary constraint. Moreover it can be proved that in very special cases, the elimination of a variable may not generate a constraint of arity equal to all its neighbors when performing ADC^-_{factor}, but a smaller arity constraint. This advantage of factoring nogoods raises the power of eliminating a variable linked by loose constraints and can reach gains of several orders of magnitude in random problems and SAT, and of one order of magnitude in n-Queens and Shur's lemma.

ADC^-_{factor} joins constraints in the bucket and generates constraints of nogoods at the same time. At the last join, when only two constraints remain, ADC^-_{factor} only needs to generate nogoods as the positive join is subsumed by all the generated nogoods. Because of this fact we can always choose a constraint of the bucket that when computing its projection with memory its completely permitted tuples (the ones that are supported by every value of the eliminated variable) will be skipped. This constraint will be the one of largest arity. When projecting with memory a negative constraint, we can compute at the same time the $\oslash$ operation (that is the tuples that are forbidden by all values of the eliminated variable) and also the tuples supported by all values of the eliminated variable.

5.4 Related Work

The idea of factoring a constraint is not new. In other contexts it is called decomposing a constraint. Certain classes of non-binary constraints are "decomposable" as they can be represented by binary constraints on the same set of variables. In [Gent et al., 1999] some implicit global constraints, like all-diff are identified as decomposable into smaller binary constraints. Then search techniques are applied to problems with global and decomposed constraints to experiment their different behavior. Our approach applies factorization to explicit constraints and is oriented to complete inference methods.

5.5 Perspectives of future work

In this Chapter we experimented with the advantages of solving a problem dealing with negative information instead of the usual way which uses positive infor-

mation. We described a process to factorize constraints into smaller arity constraints. It would be interesting to investigate the effect of factorization in other more complicated implementations of explicit constraints. *Binary Decision Diagrams* BDD [Bryant, 1986] and *Algebraic Decision Diagrams* ADD [Bahar, 1993] are examples of other implementations of explicit constraints that could exploit the advantages of factorization. [Sachenbacher and Williams, 2005] suggests to use these representations and performs some experiments on the memory savings that can be obtained.

The idea of factoring constraints lead us in Section 5.2.2 to a new inference rule, that can eliminate binary domain variables linked by binary constraints in polynomial time and space. This rule can be generalized to variables linked to n-ary constraints and to larger variable domains loosing the polynomial complexity. It would be interesting to experiment in which cases such a generalization would be applicable. Moreover, it is possible to exploit this fact inside search. In [Larrosa, 2000] a variable is eliminated during search if it has a degree inferior to a desired constant. As we assign variables during search variables decrease its degree, so elimination can be performed dynamically. We can also exploit the rule of Section 5.2.2 when a variable has binary domain during search.

5.6 Conclusions

The theoretical complexity of ADC is exponential with respect to the width of the induced graph, which is very sensitive to the arity of constraints and does not take into account its tightness. In this work, we show how sensitive is ADC not only to the arity but also to the tightness of constraints. Especially, very loose constraints and variables linked to many loose constraints can make the algorithm impractical in many cases. We have described ADC^{-}_{factor} that eliminates a variable by returning a set of constraints that does not mention that variable and that represent a set of factorized nogoods in such a way that variables are eliminated in a compact way, sometimes with exponential savings. As general conclusion, when constraints have more permitted than forbidden tuples, ADC^{-}_{factor} is the preferred choice. Otherwise, when constraints have more forbidden that permitted tuples, classical ADC may perform better.

Chapter 6

Constraint Filtering

ADC is the basic Complete Inference algorithm for solving CSP. ADC has an exponential spatial and temporal complexity with respect to the induced width (see Def. 2.10). The previous Chapter describes an alternative ADC that works with negative factorized information. It permitted to make savings in the memory usage of ADC. In this Chapter we continue in that direction, trying to reduce even further the amount of memory spent by Complete Inference methods. We introduce the idea of *Filtering* which consists in anticipating tuples that will become inconsistent when joined with other constraints of the problem, thus they can be deleted from their initial constraints. Filtering allows us to make use of parts of the problem to delete tuples of other parts. One could say that we are doing the equivalent to look-ahead during search but in an inference context where the goal is to reduce memory storage instead of pruning branches and reducing time. To our knowledge, this idea has never been exploited in Complete Inference algorithms. As we will see this idea has great impact in the variable elimination operation. Now when eliminating a variable, apart from the constraints that it is linked to, other parts of the problem can play an important role: they can be used as filters.

We also describe an idea that lead us to filtering. We call it *Delaying Variable Elimination* and it consists in relaxing the order in which classical ADC performs computations. Variable elimination is no longer viewed as an atomic process and is broken into its smaller steps that are not necessarily executed sequentially. We perform cheap joins right away, while expensive joins are delayed until the constraints have decreased their size due to other computations.

We incorporate filtering and delayed variable elimination into ADC and we prove the effectiveness of the obtained algorithm in various problems [Sanchez et al., 2004a].

```
function ADC-DVE(X, D, C)
1   while C ≠ {λ} do
2     if ∃x_i ∈ X s.t. |{c ∈ C | x_i ∈ var(c)}| = 1 then
3       X ← X − {x_i}; D ← D − {D_i}; C ← C ∪ {c ⇓ x_i} − {c}
4     else
5       x_i ← choose-variable(X)
6       B ← {c ∈ C| x_i ∈ var(c)}
7       {p, q} ← extract-two(B)
8       c ← p ⋈ q
9       if c = ∅ return false
10      else C ← C ∪ {c} − {p, q}
11  return true
```

Figure 6.1: Adaptive consistency delaying variable elimination algorithm.

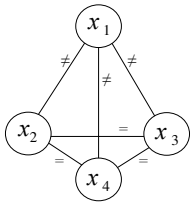
6.1 ADC and Delayed Variable Elimination

Let us recall that ADC consists in a sequence of problem transformations. At each step one variable from the problem is eliminated, while maintaining the equivalence of the problem. The elimination of a variable (see Def. 2.9) is described here in four operations *i*) we compute the constraints that mention the variable (i.e, its bucket). *ii*) we join all constraints in the bucket. *iii*) Then the variable is projected out of the resulting constraint. *iv*) The original constraints of the bucket and the new constraint are replaced in the set of constraints of the problem. We split here the join and projection in two operations as it will be convenient for the new presented algorithm. That's because variable elimination operation is no longer seen as a indivisible operation. ADC is redefined as a sequence of two by two joins, and variable eliminations when possible.

We consider (following Section 5.1 of previous Chapter) the assumption that constraints store the set of permitted tuples. Thus different joins of constraints may have different complexities (measured in terms of number of generated tuples) depending on the nature of constraints, if their share variables, etc. For this reason we aim at performing the most restrictive joins first, to keep memory usage as low as possible. We present the algorithm ADC with *delaying variable elimination* (ADC-DVE). This algorithm performs, like ADC, joins of constraints and eliminates variables. But differently from ADC, joins and variable eliminations are decoupled. One can start joining two constraints in one bucket, continue joining in another bucket, etc. The only condition it imposes is that as soon as one variable is mentioned by one constraint only, then that variable is eliminated. Given that we are allowed to perform joins of constraints that are not in the current bucket we call this idea *delayed variable elimination*. Taking advantage of DVE may produce exponential savings in memory. We show it with the following example.

Example 6.1 *Find on the drawing below a constraint graph with four variables. Every variable has a domain of d values.* x_1 *is linked to* x_2, x_3 *and* x_4 *with*

inequality constraints. Thus c_{12}, c_{13} and c_{14} have $d(d-1)$ tuples. x_2, x_3 and x_4 are linked with equality constraints. Thus c_{23}, c_{24} and c_{34} have d tuples. First we eliminate x_1 as standard ADC would do. We join all constraints in which x_1 participates $c_{12} \bowtie c_{13} \bowtie c_{14}$ and obtain a constraint $c_{1234} = \{abbb, abbc, ...\}$ with $d(d-1)^3$ tuples.



$c_{12} = c_{13} = c_{14} = \{ab, ac, ..., ba, bc, ..., ca, cb, ...\}$

$c_{23} = c_{24} = c_{34} = \{aa, bb, cc, ...\}$

Delaying the elimination of x_1 we have the chance of joining the equality constraints as soon as possible to reduce memory storage. So for example we join $c_{12} \bowtie c_{23} = c'_{123}$ and obtain a constraint with $d(d-1)$ tuples. We then join $c'_{123} \bowtie c_{13}$ and obtain the same constraint c'_{123}. If we finally join c_{34} and c_{14} with c'_{123} we obtain a constraint with the same number of tuples $d(d-1)$. Performing cheap joins right away and delaying the elimination of x_1 has in this case exponential memory savings as the needed space to eliminate x_1 is polynomial.

ADC-DVE appears in Fig. 6.1. It receives as parameters the sets of variables, domains and constraints, and returns *true* if a solution exists and $false$ otherwise. ADC-DVE iterates until the set of constraints has been reduced to the empty tuple λ (line 1). If there exists a variable x_i mentioned by a single constraint c (line 2), this variable is projected out from the constraint and finally eliminated from the problem. This is done replacing such a constraint c by $c \Downarrow x_i$ (line 3). If all variables participate in two or more constraints then ADC-DVE selects a variable (line 5). Its bucket is computed (line 6) and two constraints p, q of its bucket are selected and joined (lines 7, 8). If the resulting constraint c is empty, the problem has no solution and returns the empty set (line 9). Otherwise, p, q are replaced by c (line 10).

Different heuristics can be used for variable and constraint selection. We first select the variable that it is linked to less constraints. Then we select for joining the two constraints that are more tight.

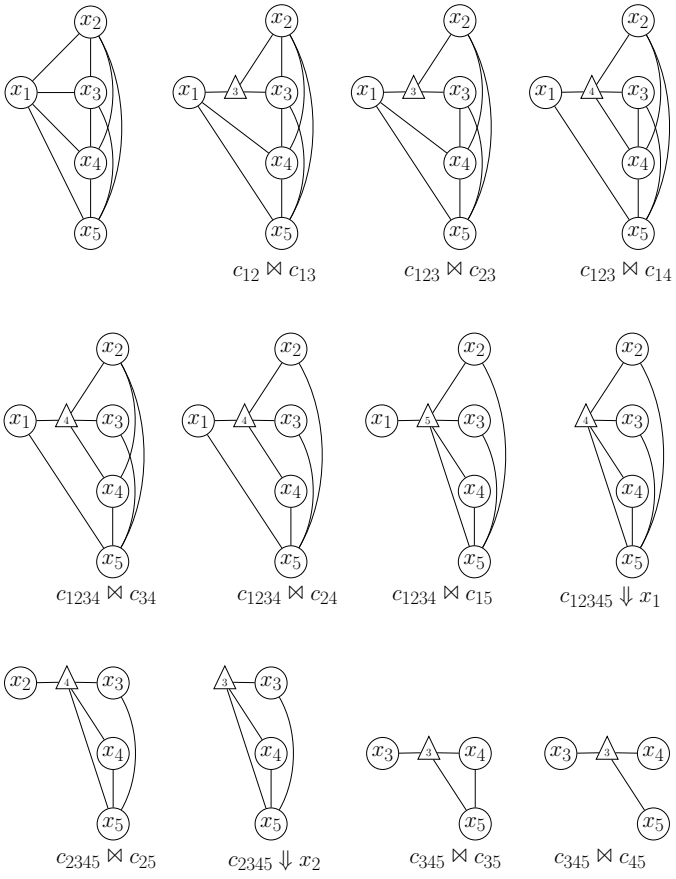
The main difference with respect to standard ADC is that we join constraints and we project a variable when it only appears in one constraint. Consider a CSP $\wp = \langle X, D, C \rangle$ such that here is a variable $x_i \in X$ which only appears in a single constraint. Consider the CSP $\wp'$ obtained from $\wp$ by projecting out variable x_i from this constraint. $\wp$ and $\wp'$ are equivalent in the remaining set of variables. Projecting out a variable from its unique constraint is the simplest case of variable elimination, the bucket only contains this constraint and no join is needed it can be directly projected out from the constraint.

ADC-DVE can be seen as an algorithm that performs a sequence of joins and variable elimination. Given a problem, a join of a subset of its constraints always

produces an equivalent problem. For variable elimination, the only requirement for correctness is that this operation cannot be done if more than one constraint mentions the variable to be eliminated.

The number of constraints decreases monotonically, and as soon as a variable is mentioned by a single constraint it is eliminated. Given that the number of constraints and variables is finite, the algorithm terminates.

Example 6.2 *Find on the drawing below a sequence of constraint graphs that illustrate a possible execution of ADC-DVE on the 5-Queens problem. ADC-DVE starts choosing variable* x_1*. The initial graph is shown at first place. The set of functions linked to* x_1 *is* $B = \{c_{12}, c_{13}, c_{14}, c_{15}\}$*. The first two constraints from* B *are selected and joined obtaining* c_{123}*. At this point standard ADC would continue joining constraints from* B *but ADC-DVE is able to select variable* x_2 *and join constraints* c_{123} *and* c_{23}*. The rest of the sequence of joins and projections is shown below,*



When only one constraint remains, the problem has a solution if this contraint is non empty.

6.2 ADC and Constraint Filtering for CSP

Delayed variable elimination allows to join tight constraints as soon as possible. In a way we are using other constraints of the problem that are unreachable for usual ADC when eliminating a variable. This idea lead us to what we call *constraint filtering* which is more powerful than DVE and that permits us to use groups of constraints of other parts of the problem.

6.2.1 Constraint Filtering

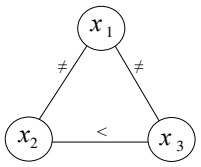
We now introduce the *constraint filtering* operation, which allows us to reduce the size of a constraint c before operating with it. The idea is to anticipate the detection of tuples that will become inconsistent when joined with other constraints of the problem in order to remove them from the constraint.

Definition 6.1 [**constraint filtering**] *Let c be an arbitrary constraint and H a set of constraints such that $\forall_{h\in H}\ var(h) \subseteq var(c)$. The filtering of c with respect to H is a new constraint $\overline{c}^H$ with the same scope as c that contains all the tuples of c permitted by every constraint of H. Formally,*

$$\overline{c}^H(t) = \begin{cases} true & c(t) \wedge \forall_{h\in H}\ h(t) \\ false & otherwise \end{cases}$$

We are under the assumption that constraints only store permitted tuples (following Section 5.1 of previous Chapter), thus filtering a constraint with a set of filtering constraints may delete tuples of it.

Example 6.3 *Find on the drawing below a constraint graph with three variables. Every variable has a domain $D_i = \{a, b, c\}$. x_1 is linked to x_2 and x_3, $x_1 \neq x_2$ and $x_1 \neq x_3$. x_2 and x_3 are linked by $x_2 < x_3$ (in lexicographic order).*



$c_{12} = c_{13} = \{ab, ac, ba, bc, ca, cb\}$

$c_{23} = \{ab, ac, bc\}$

We can filter constraint c_{12} by using c_{23}. Take for example $c_{23} \Downarrow x_3 = \{a, b\}$. We then apply filtering $\overline{c_{12}}^{\{c_{23}\Downarrow x_3\}}$ and obtain $c'_{12} = \{ab, ba, ca, cb\}$. Tuples $\{ac, bc\}$ have been eliminated as value c is not permitted by the filtering function $c_{23} \Downarrow x_3$.

The filtering can also be applied to a join operation. Then the tuples that are forbidden by the filters are never stored.

Definition 6.2 *Consider two constraints p, q and a set of filtering constraints H. We call joining p and q with filters H, noted* $\overline{p \bowtie q}^H$ *a new constraint which is the join of both constraints where tuples forbidden by the filters H have been deleted.*

We present a simple algorithm for joining two constraint with a set of filtering functions in Fig. 6.2. It receives as input two constraints p and q and the filtering set H. It returns as result the joined functions. The algorithm iterates over all tuples in p and all tuples in q. If the concatenation $t \cdot t'$ is defined then it checks if the resulting tuple is permitted by all functions in H.

```
function join-F(p, q, H)
1 c ← ∅
2 for each t ∈ p do
3   for each t' ∈ q do
4     if t · t' is defined then
5       if ∀_{h∈H} h(t · t') then c ← c ∪ {t · t'}
6 return c
```

Figure 6.2: Join with filters algorithm.

Example 6.4 *Consider the constraint graph of previous example 6.3. We perform the join* $c_{12} \bowtie c_{13} = \{abb, abc, acb, acc,\ baa, bac, bca, bcc,\ caa, cab, cba, cbb\}$ *and obtain a constraint with 12 tuples. Now if we perform the same join filtering with constraint* c_{23}, $\overline{c_{12} \bowtie c_{13}}^{\{c_{23}\}} = \{abc, bac, cab\}$ *we obtain a constraint where all the tuples where* x_2 *is lexicographically greater or equal than* x_3 *have been eliminated. The eliminated tuples have never been stored in memory.*

Filtering permits us to reduce memory storage of certain constraints by using other constraints of the problem. Suppose that we know that a constraint c will be eventually joined with constraint v. If there is a tuple $t \in c$ such that $t \cdot t'$ will not be permitted by the join of both constraints for any $t' \in v$, we can safely remove t from c right away. The following Property formalizes the previous observation. Remember from Def. 2.7 that a constraint stronger than a set of constraints means that the tuples that the constraint forbids are also forbidden by the join of all constraints in the set.

Property 6.1 *Let c (resp. s) be a constraint and C (resp. S) sets of constraints such that c is stronger than C,* $C \preccurlyeq \{c\}$*, and s is stronger than S,* $S \preccurlyeq \{s\}$*. When joining c and s, if we previously filter each constraint the result is preserved. Namely,*

$$\overline{c}^S \bowtie \overline{s}^C = c \bowtie s$$

Besides, the join is done with constraints of smaller size. Thus, it is presumably done more efficiently.

Proof. Suppose that constraints in S when joined together produce the universal constraint that permits all tuples. Then $S \preccurlyeq \{s\}$ and of course $\overline{c}^S = c$. Now suppose that constraints in S forbid a number of tuples. These tuples, or any extension of them, are forbidden by s because $S \preccurlyeq \{s\}$. If we eliminate these tuples from c, $\overline{c}^S$, we obtain a constraint with less permitted tuples than c, so $|\overline{c}^S| < |c|$. Then when joined with s, $\overline{c}^S \bowtie s$, we obtain the same as $c \bowtie s$ because s forbids all the tuples of constraints S and possibly more. The same argument applies for the dual case $\overline{s}^C$. □

The following property shows that filtering constraints can be safely brought inside joins, anticipating the detection of nogoods and reducing the size of constraints.

Property 6.2 *Let c and s be two constraints, and H a set of filtering constraints. We have that,*

$$\overline{c \bowtie s}^H = \overline{\overline{c}^H \bowtie \overline{s}^H}^H$$

Proof. It is correct to filter both constraints with the filtering set of functions because the eliminated tuples would also be eliminated after the join. Moreover by doing so we may reduce the size of the join. It is necessary to perform a final filtering join because they may be some functions $h \in H$ that have scope bigger than $var(c)$ and bigger than $var(s)$. □

The following property shows that constraints that served as filters can be eliminated from the problem.

Property 6.3 *If $var(h) \subseteq (var(c) \cup var(s))$, then $\overline{c \bowtie s}^{\{h\}} = (c \bowtie s) \bowtie h$.*

Proof. Considering $\overline{c \bowtie s}^{\{h\}}$, when generating every tuple of the join of c and s it is stored only if it is permitted by constraint h. Considering $(c \bowtie s) \bowtie h$, we first store in a temporal constraint all the tuples permitted by $c \bowtie s$ and then delete all the tuples forbidden by h with a second join. □

The worst case when joining constraints with filters happens when filters do not remove any tuple of the standard join. In this case, joining with filters uses the same space as standard join. Thus, joining two constraints with filters never uses more space than standard join. Moreover it may produce exponential savings in space. Let us illustrate it with an example. Let p and q be constraints such that $var(p) \cap var(q) = \emptyset$ and let h be a filter, $var(h) \subset (var(p) \cup var(q))$. Let $t \in p$ a tuple. When performing $p \bowtie q$, tuple t generates $|q|$ tuples in the join. If t is not allowed by h, when performing $\overline{p \bowtie q}^{\{h\}}$ saves $|q|$ tuples, a number that is bounded above by $d^{|var(q)|}$. Therefore, joining with filters may save exponential space.

6.2.2 Adding Filtering into ADC-DVE

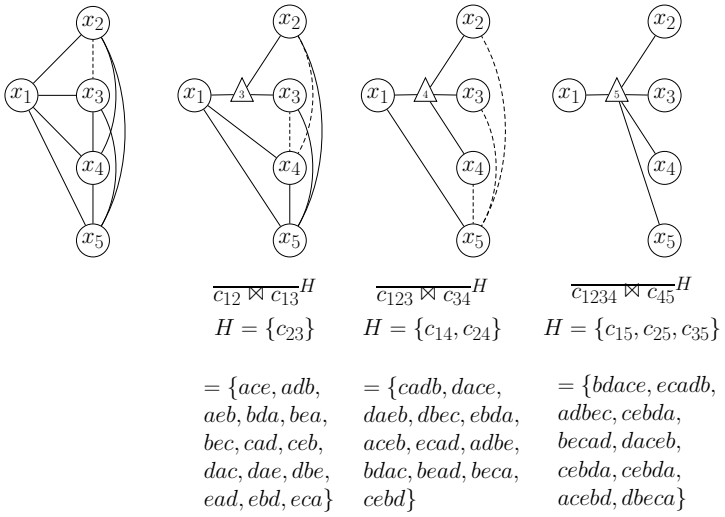
We include filtering into ADC-DVE. The corresponding algorithm is noted ADC-DVE-F and appears in Fig. 6.3. It is essentially the same algorithm as ADC-DVE but when joining two constraints a set H of constraints is used for filtering.

function ADC-DVE-F(X, D, C)
1 **while** $C \neq \{\lambda\}$ **do**
2 **if** $\exists x_i \in X$ s.t. $|\{c \in C \mid x \in var(c)\}| = 1$ **then**
3 $X \leftarrow X - \{x_i\}$; $D \leftarrow D - \{D_x\}$; $C \leftarrow C \cup \{c \Downarrow x_i\} - \{c\}$
4 **else**
5 $x_i \leftarrow$ choose-variable(X)
6 $B \leftarrow \{c \in C |\ x_i \in var(c)\}$
7 $\{p, q\} \leftarrow$ extract-two(B)
8 $H \leftarrow \{c \in C |\ var(c) \subseteq (var(p) \cup var(q))\}$
9 $c \leftarrow \overline{p \bowtie q}^H$
10 **if** $c = \emptyset$ **return** $false$
11 **else** $C \leftarrow C \cup \{c\} - \{p, q\} - H$
12 **return** $true$

Figure 6.3: Adaptive consistency with delayed variable elimination and filtering.

H includes all the constraints of the problem that are included in the scope of the resulting constraint of the join (line 8). Then join is performed with filtering $\overline{p \bowtie q}^H$ (line 9). Functions used as filters can be removed from C (line 11)

Example 6.5 *Find on the drawing below four constraint graphs that illustrate the execution of ADC-DVE-F on the 5-Queens problem. Under each constraint graph the operation to obtained is indicated. Dashed constraints are the ones used as filters at each step.*



The first constraint graph is the initial problem. ADC-DVE-F first selects x_1 and all the constraints it is linked to, $B = \{c_{12}, c_{13}, c_{14}, c_{15}\}$. From this set it selects c_{12} and c_{13} and computes their filtering set $H = \{c_{23}\}$. The result $\overline{c_{12} \bowtie c_{13}}^{\{c_{23}\}}$ is shown below the second constraint graph. The constraint that

has acted as filter disappears. If we had used no filter the resulting constraint would had $|c_{12} \bowtie c_{13}| = 34$ *tuples, 20 tuples have been discarded. Let's see now the advantages of DVE. Following ADC algorithm now we would be obliged to join one of these three constraints* $B = \{c_{123}, c_{14}, c_{15}\}$*. As we are in ADC-DVE we have the freedom to join two constraints of another variable we choose now* x_3 *which in fact has a constraint more tight than* x_1 *that is* c_{34} *because involves consecutive variables. A filter exists for this join* $H = \{c_{24}, c_{34}\}$*. In the final step we perform the join with the more tight constraint of* x_5*,* c_{45}*. A filter exists for this join* $H = \{c_{15}, c_{35}, c_{25}\}$*. The result is all the solutions without removing symmetries of the problem. Notice that no elimination (projection) of a variable was necessary. In fact the algorithm can stop when there is only one constraint left in the problem.*

6.2.3 Filtering and negative factorized constraints

In previous Chapter 5 we described an ADC that works with negative factorized constraints, we call it ADC^{-}_{factor}. It is possible to combine the idea of filtering with negative factorized constraints.

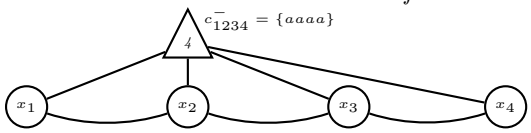
In Section 6.2 we introduced the join with filtering operation $\overline{c \bowtie r}^{H}$ that was able to use other constraints of the problem (constraints in set H) to filter tuples of the result of joining c and r reducing memory storage. This idea can also be used in conjunction with negative constraints and also factoring. If we generate a nogood that is already factorized by a smaller nogoods it can be discarded.

Property 6.4 *When used as filters constraints can be either positive or negative. So* $\overline{p \bowtie q}^{\{c\}}$ *has the same time and space complexity as* $\overline{p \bowtie q}^{\{c^-\}}$*.*

Proof. The space and time complexity of a join with filtering is dependant only on the two joined constraints as we can ask in constant time if a certain generated tuple is forbidden by the filter independently of being positif or negative. □

This fact suggests that negative and positive constraints can coexist without the necessity of being joined until negative constraints can serve as filters of a join of positive constraints. In this sense negative constraints may play the role of exceptions. There are usually few exceptions, so there is no point in working with them as positive, but rather use exceptions as filters of positive information when possible.

Example 6.6 *Consider the constraint hypergraph of* 4 *variables and* d *values per variable of the drawing below. All binary constraints are equality constraints. There is one four arity constraint that forbids only one combination of values. This four artiy constraint can be considered as an exception, a particular combination that we don't want to be in the set of solutions.*



ADC is forced to eliminate variable x_1 by joining all its positive constraints. If we want to join c^-_{1234} and r_{12} and obtain as result a positive constraint we have to negate the negative constraint $\neg c^-_{1234} = c_{1234}$. The resulting positive constraint has $|c_{1234}| = d^4 - 1$ tuples. Let's compute the join $c_{1234} \bowtie r_{12}$. The resulting constraint has $d^3 - 1$ tuples. The idea that we suggest in this Section would be to work separately with negative and positive information. We join all positive constraints and when possible we use negative constraints as filters: first we join $r_{12} \bowtie r_{23} = r_{123}$ and obtain a constraint with d tuples. We then perform the join with filters $\overline{r_{34} \bowtie r_{123}}^{c^-_{1234}}$ obtaining a constraint with $d - 1$ tuples. In this synthetic example we gain an exponential number of tuples because in the first case we store a maximum of $d^3 - 1$ and in the second case only d.

6.3 Experimental Evaluation

We tested three algorithms: ADC, ADC-DVE (ADC with delayed variable evaluation) and ADC-DVE-F (with filtering) in two classes of problems, n-Queens and Schur's lemma. Experimenters are focused in evaluating the memory consumption of all algorithms, although time is reported to justify that in any case the improvements in memory are substituted by an unacceptable amount of CPU time. The usual cause of non-solvability of an instance was always running out of memory (we assumed a memory limit of 2,000,000 tuples) for all the algorithms and the two type of problems. We put the emphasis on the different algorithmic behavior. No computation of the induced width of the graph is done.

6.3.1 N-Queens

The n-queens problem is described in example 2.1. The constraint graph is a clique (every variable is connected with all the others) and cliques have induced width $w^{opt} = n-1$. ADC time and space complexity is exponentially dependent on this parameter.

In Fig. 6.4 on the top ADC instances from $n = 4$ to $n = 7$ which was the highest dimension that could be solved. $n = 8$ runs out of memory in the first variable elimination. The peak in the plots is the number of tuples just before the first variable elimination. The first elimination is always the most expensive in number of tuples and for increasing n it grows exponentially. Time plot in the right is not very relevant because the lack of memory is reached very quickly.

In the middle we have results for ADC-DVE. The jagged shape in the plot is due to the storage of joins that afterwards can be filtered by smaller constraints because we can delay the evaluation of the current variable and perform the filtering joins of other constraints in the problem. Because of this property we can solve instances up to $n = 11$.

At the bottom, we report results for ADC-DVE-F. The plots of tuples get smoother because the filtering joins are included in each join as filters. Observe that last point in each line has a number of joins equal to the $n - 2$. The last

point in each line for n-queens also represents the total number of solutions without removing symmetries. We observe that it grows exponentially as n grows. Instances up to $n = 13$ could be solved. We reached a gain of 6 orders of magnitude with respect to standard ADC, increasing domain size and variable number.

6.3.2 Schur's Lemma

The shur lemma problem is described in Appendix A.1.3. In the modelisation a particular variable appears in few constraints (usually $n-1$ constraints) of arity 3, but of different scopes. In fact with this modeling each increasing n implies the addition of three new binary variables.

The Schur's lemma instances are not cliques, so ADC-DVE-F performs projections also, that explains the jagged shape in the tuples plot (see Figure 6.5). Peaks and descends are due to projections. A number of tuples greater than 0 in the last point of every line implies the existence of solutions. When we have performed projections this number is not necessarily the total number of solutions. The whole set of solutions can be obtained as in ADC by consistently extending the consistent tuples found in the last step. When applying filtering (ADC-DVE-F) as constraints in the filter can disappear, sometimes it is not necessary to project until one variable is left, we can stop when only one constraint is left in the problem. The tuples in that constraint are all the set of solutions of that part of the problem. This fact explains why in the tuple plots last points in the line do not go down as in ADC or ADC-DVE plots. ADC-DVE-F is able to solve problems 14 dimension bigger than ADC.

6.3.3 Discussion

In the instances on which ADC runs out of memory, it exhausts memory very quickly, so time plots are superfluous. For both problems, the first unsolved instance by ADC-DVE also runs out of memory, n-queens doing it more quickly than Schur-lemma instances. The number of performed joins (x axis) for ADC and ADC-DVE is equal to the initial number of constraints minus one. ADC-DVE-F performs much less joins. In a n-clique binary graph ADC-DVE-F performs $n-2$ joins with 1 to $n-1$ filters each one. The fact that the tuples of temporal joins are not stored and that all the constraints in the filter can be deleted from the problem gives a clear advantage in time and memory space to ADC-DVE-F. In n-queens, ADC-DVE-F could solve problems 6 dimensions bigger than ADC: In Schur's lemma ADC-DVE-F could solve problems 14 dimensions bigger than ADC.

6.4 Conclusions

The importance of decomposition methods in general, and ADC in particular, is often considered as purely theoretical, since most constraint satisfaction prob-

lems have a high induced width. While this is true in general, we believe that their true potential is still to be discovered. Some authors have studied how the limitation of these algorithms can be overcomed by using them in a restricted way (producing hybrid algorithms). In this paper we follow a different approach, since we think that not enough effort has been made to develop efficient implementations. We have proposed two modifications to the standard ADC definition with which its time and space efficiency can be greatly improved. First, we have shown that computations can be re-arranged and the expensive ones can be delayed with the hope that the involved relations may have their size decreased in the mean time. We call this idea delaying variable elimination. Second, we have shown that joins involving small arity constraints, which can be efficiently computed, can be anticipated in order to detect and filter out tuples from large arity constraints, where the exponential cost of the algorithm is more likely to become apparent. Any of these ideas can bring exponential saving over ADC. Our preliminary experimental results are very promising: time and space requirements of ADC may decrease several orders of magnitude. Therefore, the number of real problems where ADC can be applied may increase.

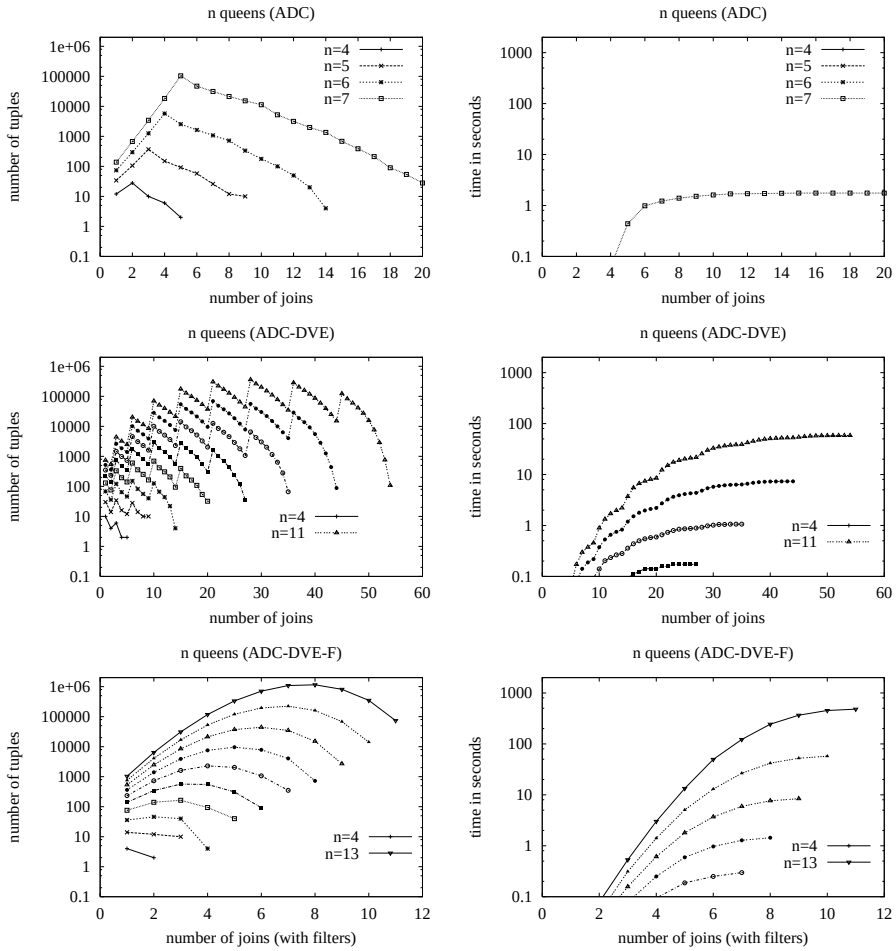


Figure 6.4: Plots on the left are number of stored tuples in the actual join (log scale and we assume a maximum value 2,000,000). Plots on the right are cpu time. X axis is the number of performed joins. Top: ADC. Middle: ADC-DVE. Bottom: ADC-DVE-F. Plotted lines are instances that could be solved

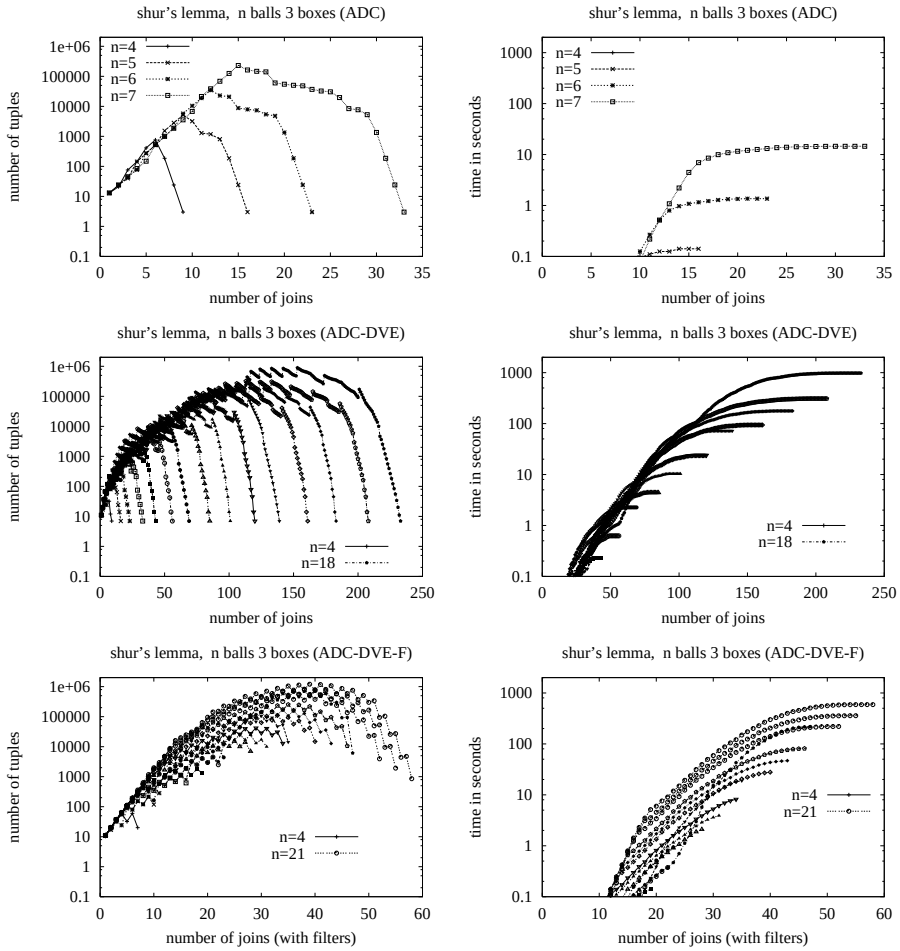


Figure 6.5: Plots on the left are number of stored tuples in the actual join (log scale and we assume a maximum value 2,000,000). Plots on the right are cpu time. X axes is the number of performed joins. Top: ADC executions. Middle: ADC-DVE. Bottom: ADC-DVE-F. Plotted lines are instances that could be solved.

Chapter 7

Function Filtering

In this Chapter we extend the idea of filtering (introduced in the previous Chapter 6) from the context of CSP to WCSP. In WCSP, and without loss of generality, we can assume the existence of an upper bound `ub`, a maximal acceptable cost that we are willing to pay. Then the inconsistency of a tuple can be defined with respect to this upper bound. If the cost of a tuple exceeds `ub` then it is inconsistent. As we show `ub` is used to delete inconsistent tuples (i.e., tuples that we know for sure that they don't belong to an optimal solution). Bucket Elimination (BE) [Dechter, 1999] is the extension of ADC to WCSP and it is defined in Section 2.4.2. We incorporate delayed variable elimination and a new definition of filtering using `ub` to BE.

We also extend the idea of filtering. Apart from functions of the original problem, manipulations of functions (projected functions or summed functions, or constant lower bounds) can also be used as filters. We take a new framework to develop this idea, Cluster Tree Elimination (CTE) [R.Dechter and J.Pearl, 1989] which is an algorithm that generalizes the Complete Inference algorithms explained up to the moment. BE for example can be seen as a particular execution of CTE. We introduce filtering into CTE. In this context, whole parts of the problem, namely, clusters of functions and manipulations of them, can be used as filters. We also develop an iterative approximation of CTE which computes a sequence of iterations of increasing complexity, reusing the previous iterations as filters [Sanchez et al., 2005a, Sanchez et al., 2005b].

7.1 From Constraint to Function Filtering

The central idea of filtering for CSP was to detect and remove tuples that will become inconsistent when joined with other constraints of the problem. Following [Larrosa and Schiex, 2004] in WCSP we assume without loss of generality a maximal acceptable cost, an upper bound (`ub`) of the problem. We suppose this `ub` is given to us or can also be obtained with local search techniques. Thus we can redefine the concept of consistency with respect to this `ub`. A tuple that has

cost greater than ub is inconsistent as we know it cannot belong to an optimal solution.

We consider now how functions are stored. In Chapters 5 and 6 we made the assumption that constraints are stored as sets of tuples for the CSP context (see for example Section 5.1). Analogously to the CSP case its is considered here that a function f is stored as a set of all its tuples with its associated cost. We make now the additional assumption that tuples that have cost ub or greater are not stored in the function. Then, if a tuple is not present in the constraint we know that it has cost $\geq$ ub. If a function is implemented as a hash table, $f(t)$ can be retrieved in constant time. We note $|f|$ the number of tuples of f that have a cost less than ub.

7.1.1 Function filtering

Function filtering operation is the extension of *constraint filtering* to WCSP. *Function filtering* allows to detect tuples of a function that will reach ub when summed to other functions of the problem. Thus they can be deleted from the function.

Definition 7.1 [function filtering] *Let f be an arbitrary function and H a set of functions such that $\forall_{h \in H}\ var(h) \subseteq var(f)$. The filtering of f with respect to H is a new constraint $\overline{f}^H$ with the same scope as f and where tuples that have reached cost* ub *with the addition of the cost assigned by functions in H have been assigned cost* ub*. Formally,*

$$\overline{f}^H(t) = \begin{cases} f(t) & \left(\bigoplus_{h \in H} h(t)\right) \oplus f(t) < \texttt{ub} \\ \texttt{ub} & \textit{otherwise} \end{cases}$$

Analogously to CSP we give a simple algorithm for summing two functions with filters H (see Fig. 7.1). It receives as input two functions f and g and the filtering set H. It returns as result $\overline{f+g}^H$, the summed functions. The algorithm iterates over all tuples in f and all tuples in g. If the concatenation $t \cdot t'$ is defined then it checks if the resulting tuple does not reach cost ub when its costs is summed with all the costs returned by functions in H.

function sum-F(f, g, H)
1 **for each** $t \in f$ **do**
2 **for each** $t' \in g$ **do**
3 **if** $t \cdot t'$ is defined **then**
4 lb $\leftarrow f(t) + g(t') + \sum_{h \in H} h(t \cdot t')$
5 **if** lb $<$ ub **then** $f'(t \cdot t') \leftarrow f(t) + g(t')$
6 **return** f'

Figure 7.1: Sum with filters algorithm.

Properties presented for CSP can be extended to WCSP. Suppose that we know that f will be eventually summed with g. If there is a tuple t belonging to f such that for every tuple t' we have that $(f+g)(t \cdot t') \geq \texttt{ub}$ then, we can safely remove t from f right away. The following property is the extension to WCSP of property 6.1 and formalizes the previous observation.

Property 7.1 *Let f (resp. g) be a function and F (resp. G) a set of functions that are a lower bound of f (resp. g) ($F \leq f$ and $G \leq g$). When summing f and g, if previously we filter each function with the lower bound of the other function, the result is preserved. Namely,*

$$\overline{f}^{G} + \overline{g}^{F} = f + g$$

Besides, the sum is done with functions of smaller size in terms of numbers of tuples. Thus, it is presumably done more efficiently.

Proof. Analogous to property 6.1. □

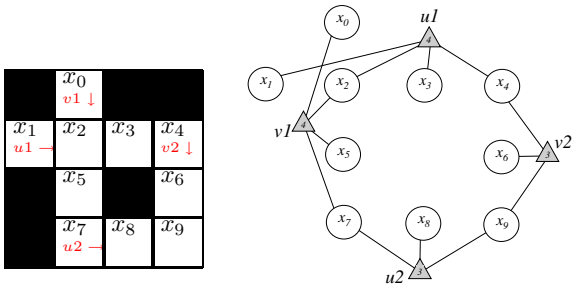
The following property is the extension to WCSP of property 6.2 and shows that filtering functions can be safely brought inside summations, anticipating the detection of nogoods and reducing the size of functions.

Property 7.2 *Let f and g be two constraints, and H a set of functions. We have that,*

$$\overline{f+g}^{H} = \overline{\overline{f}^{H} + \overline{g}^{H}}^{H}$$

Proof. Let us evaluate tuple t in the left expression: t has cost $f(t)+g(t)+\sum_{h\in H} h(t)$. If this cost reaches $\texttt{ub}$, then the tuple is deleted from the result, if not the resulting cost is $f(t)+g(t)$. Let us evaluate tuple t in the right expression. We first evaluate $f(t)+\sum_{h\in H} h(t)$ and leave the cost of $f(t)$ untouched if it does not reach $\texttt{ub}$. Same thing for $g(t)+\sum_{h\in H} h(t)$. We then finally evaluate $f(t)+g(t)+\sum_{h\in H} h(t)$. As before, if this cost reaches $\texttt{ub}$ we delete the tuple but if not the cost remains $f(t)+g(t)$. So finally only the tuples that finally reached $\texttt{ub}$ where deleted in both sides, but in the right they may be presumably deleted earlier. Notice that for the filtering to be applied in a correct way functions f and g must not belong to the set H, if this was the case, the same cost could be considered twice. □

Example 7.1 *Let us consider en example of WCSP inspired form the crossword puzzle. Find on the left of drawing below a crossword puzzle that can be modeled with a variable per cell and a constraint per each vertical and horizontal slot. Each variable takes values in all possible letters so each variable x_i has $|D_i| = 26$ values. Its constraint hyper-graph is shown in the middle. All constraints are shown in the right. Each tuple is assigned a cost. Tuples that are not present have cost $\texttt{ub} = \infty$.*



u1	x_1	x_2	x_3	x_4
0	z	e	r	o
1	o	r	e	z
4	f	o	u	r
5	r	u	o	f
5	f	i	v	e
6	e	v	i	f
9	n	i	n	e
10	e	n	i	n

u2	x_7	x_8	x_9
1	o	n	e
2	e	n	o
2	t	w	o
3	o	w	t
6	s	i	x
7	x	i	s
10	t	e	n
11	n	e	t

v1	x_0	x_2	x_5	x_7
0	z	e	r	o
1	o	r	e	z
4	f	o	u	r
5	r	u	o	f
5	f	i	v	e
6	e	v	i	f
9	n	i	n	e
10	e	n	i	n

v2	x_4	x_6	x_9
1	o	n	e
2	e	n	o
2	t	w	o
3	o	w	t
6	s	i	x
7	x	i	s
10	t	e	n
11	n	e	t

Function $u1$ *with scope* $var(u1) = \{x_1, x_2, x_3, x_4\}$ *has* 26^4 *potential tuples but as we only record consistent tuples we have* $|u1| = |u2| = 8$. *Functions* $u1$ *and* $u2$ *do not share any variable so* $|u1 + u2| = 64$. *If we set* $\mathtt{ub} = 5$, *this causes that some tuples of* $u1$ *and* $u2$ *become inconsistent and they can be eliminated. For instance, this is the case of tuple* $five$ *of* $u1$. *Now* $|u1| = 3$ *and* $|u2| = 4$. *To compute* $|u1 + u2| = 8$, *we need* $3 * 4 = 12$ *operations. We use Property 7.1, we take as* G *the function* $u2 \Downarrow \{x_7, x_8, x_9\}$, *that is,* $G = \{u2 \Downarrow \{x_7, x_8, x_9\}\} = \{1\}$ *(*G *is a lower bound of* $u2$*). Therefore,* $|\overline{u1}^G| = 2$. *Filtering with* G *allows us to detect that tuple "*$four$*" with* $u1(four) = 4$ *becomes a nogood with the sum of* G *(it reaches* $\mathtt{ub} = 5$*) and can be eliminated. Therefore, we only need* $2 * 4 = 8$ *operations to compute the sum.*

7.1.2 Bucket Elimination with Function Filtering

Bucket Elimination (BE) is the extension of ADC for WCSP. BE is presented in Chapter 2 in Section 2.4.2. The inclusion of filtering into BE is straightforward. It is the natural extension to WCSP of ADC-DVE-F (see Fig. 6.3). With respect to ADC-DVE-F, join is replaced by sum, projecting out is replaced by projecting out by minimization, the empty tuple is replaced by a constant and constraint filtering is replaced by function filtering. We obtain BE-DVE-F presented in Fig. 7.2. BE-DVE-F has four input parameters: the set of variables X, the collection of domains D, the set of constraints C and the upper bound $\mathtt{ub}$. BE-DVE-F returns $true$ if a solution exists and $false$ otherwise.

Another issue differentiates both algorithms. Property 6.3 of CSP constraint filtering states that a constraint that is used as filter can be eliminated from the problem because it is equivalent to joining it. This property is false for WCSP. In CSP after filtering a join of two constraints with the set of constraints H such

that $\forall_{h \in H} var(c) \subseteq (var(c) \cup var(v))$, that is, $\overline{c \bowtie v}^H$, we can discard constraints in H. This is not the case for WCSP as filtering operation for WCSP leaves the cost of the filtered function unmodified. Filtering only deletes tuples that have reached `ub`. So the filtering functions have to be summed at some point to take into account its cost.

BE-DVE-F iterates until the set of functions contains a single 0-arity function (a constant) (line 1). Then checks if there is variable linked to a single constraint, if there is it can be projected out (line 3). If not it chooses a variable and computes the set functions in which it appears B (lines 5,6). It selects two functions from B and computes their filtering set of functions, excluding functions from B themselves (line 8). This exclusion of functions in B was not necessary for CSP. BE-DVE-F continues joining the two constraints with filtering (line 9). If it has found an empty function then returns with no solution. In other case it eliminates both functions from the set of constraints, but the set H remains on C because it must be summed at some point (differently from CSP).

function `BE-DVE-F`$(X, D, C,$ `ub`$)$
1 **while** $C \neq \{\text{cte}\}$ **do**
2 **if** $\exists x_i \in X$ s.t. $|\{c \in C \mid x_i \in var(c)\}| = 1$ **then**
3 $X \leftarrow X - \{x_i\}$; $D \leftarrow D - \{D_i\}$; $C \leftarrow C \cup \{f \Downarrow x_i\} - \{f\}$
4 **else**
5 $x_i \leftarrow$ `choose-variable`(X)
6 $B \leftarrow \{f \in C |\ x_i \in var(f)\}$
7 $\{g, k\} \leftarrow$ `take-two`(B)
8 $H \leftarrow \{f \in C - B \mid var(f) \subseteq (var(g) \cup var(k))\}$
9 $h \leftarrow \overline{g + k}^H$
10 **if** $h = \emptyset$ **then return** $false$
11 **else** $C \leftarrow C - \{g, k\} + \{h\}$
12 **return** $true$

Figure 7.2: Bucket Elimination with filtering algorithm.

7.2 Tree Decomposition Methods: CTE and MCTE

We introduce now tree decomposition methods, CTE and MCTE, with the aim of adding function filtering to them. CTE extends BE algorithm in the sense that BE can be seen as a particular instance of CTE [Kask et al., 2006].

7.2.1 Tree decomposition

A *tree decomposition* of a CSP and a WCSP is a clustering of functions of C such that clusters are linked if they share variables and form an acyclic tree network. In a cluster we keep variables and functions. The maximum number of variables

in a cluster is a parameter of the degree of cyclicity of the constraint graph. The definition slightly differs for the CSP and WCSP case (see the following property 7.3). Formally we have,

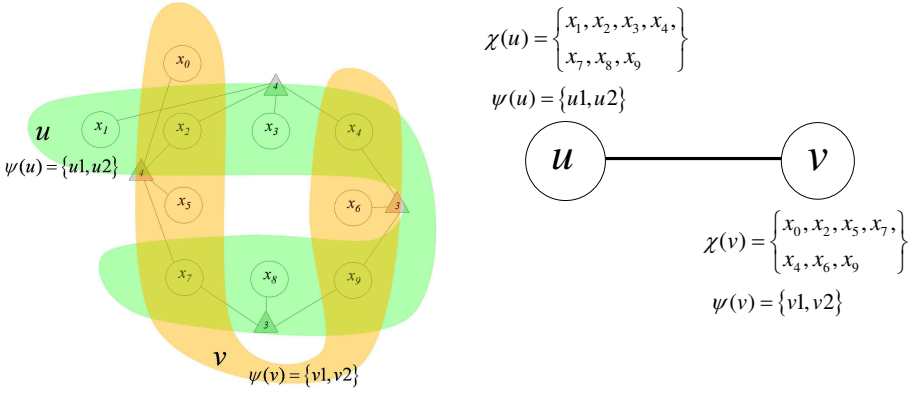
Definition 7.2 [tree decomposition] *A tree decomposition for a CSP* $\wp = \langle X, D, C\rangle$ *is a triplet* $\langle T, \chi, \psi\rangle$, *where* $T = \langle V, E\rangle$ *is a tree.* χ *and* ψ *are labeling constraints which associate with each vertex* $v \in V$ *two sets,* $\chi(v) \subseteq X$ *and* $\psi(v) \subseteq C$ *that satisfy the following conditions:*

1. *For each constraint* $c \in C$, *there is at least one vertex* $v \in V$ *such that* $f \in \psi(v)$. *In addition* $var(c) \subseteq \chi(v)$.
2. *For each variable* $x \in X$, *the set* $\{v \in V | x \in \chi(v)\}$ *induces a connected subtree of* T.

Property 7.3 Tree decompositions for WCSP *tighten condition (1) by requiring that any function* $f \in C$ *must appear in* exactly one *vertex* $v \in V$ *of the decomposition (see [Dechter, 2003]).*

Definition 7.3 [tree-width, separator, eliminator] *The tree-width of a tree decomposition is the maximum number of variables in a vertex minus one* $tw = max_{v \in V}|\chi(v)| - 1$. *Let* $\langle u, v\rangle$ *be an edge of a tree-decomposition, the separator of* u *and* v *is* $sep(u, v) = \chi(u) \cap \chi(v)$. *We will call* s *the maximum separator size* $s = max_{\langle u,v\rangle \in E}|sep(u, v)|$. *The eliminator of* u *and* v *is defined as* $elim(u, v) = \chi(u) - sep(u, v)$.

Example 7.2 *Find on the left of the drawing below a clustering of the functions of the constraint graph of example 7.1. A shaded region represents the clustering of functions* $u1$ *and* $u2$. *Another shaded region represents the clustering of functions* $v1$ *and* $v2$. *In the right we show the tree decomposition associated with this clustering. Vertex* u *of the tree decomposition contains horizontal functions* $u1$ *and* $u2$, *and vertex* v *contains vertical functions* $v1$ *and* $v2$.



7.2.2 Cluster Tree Elimination

Cluster-Tree Elimination (CTE) is a generic algorithm that can be used for CSP and WCSP solving and unifies most complete inference algorithms such as Bucket Elimination. In the following we focus on the WCSP case. CTE solves a WCSP by sending messages along the edges of the tree decomposition of the problem. Concepts presented in this Section are more extensively described in [Dechter, 2003].

Given a tree decomposition $\langle\langle V, E\rangle, \chi, \psi\rangle$, every edge $\langle u, v\rangle \in E$ has associated two *messages* that we denote $m_{\langle u,v\rangle}$, from u to v, and $m_{\langle v,u\rangle}$, from v to u. Message $m_{\langle u,v\rangle}$ is a function computed summing all functions in $\psi(v)$ with all incoming messages except from $m_{\langle v,u\rangle}$ and then projecting out the variables in u not mentioned by v, that is variables in $elim(u, v)$. Consequently, $m_{\langle u,v\rangle}$ has scope $sep(u, v)$.

In Fig. 7.3 we present the CTE algorithm. It receives as input a WCSP instance $\mathcal{W}$, a tree decomposition and the upper bound `ub`. CTE computes and sends two messages associated to each edge of the tree decomposition. CTE returns the structure of all computed messages. The solution can be then obtained from these messages.

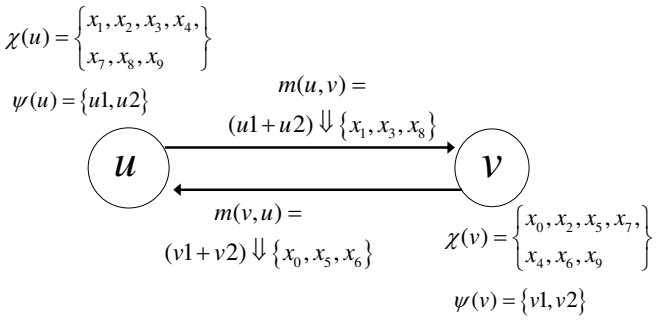
CTE iterates over all edges such that all their incoming messages but one have arrived (line 1). It gathers the set of functions to be summed (line 2) and performs their sum (line 3). Then before sending the message to node v all variables that are not mentioned by v (these are the variables in $elim(u, v)$) have to be projected out from the message (line 3). The algorithm finishes when all messages have been computed. The algorithm then returns the structure including all computed messages. Then, the optimal cost of solving the problem can be computed at every node. It suffices to sum all functions in the node including all incoming messages and if there are tuples with cost $<$ `ub` in the resulting function we know that these tuples can be extended to an optimal solution. To actually obtain these solutions we can proceed analogously as ADC or BE. In fact, the sum of all tuples of minimum cost that we can obtain in every cluster is the set of all optimal solutions.

procedure `CTE`$(\mathcal{W}, \langle\langle V, E\rangle, \chi, \psi\rangle,$ `ub`$)$
1 **repeat** select $\langle u, v\rangle \in E$ s.t. all incoming $m_{\langle i,u\rangle}$ but one ($m_{\langle v,u\rangle}$) have arrived
2 $\quad B \leftarrow \psi(u) \cup \{m_{\langle i,u\rangle} \mid \langle i, u\rangle \in E, i \neq v\}$
3 $\quad m_{\langle u,v\rangle} \leftarrow (\sum_{f\in B} f) \Downarrow elim(u, v)$
4 $\quad$ send $m_{\langle u,v\rangle}$
5 **until** all $m_{\langle i,u\rangle}$ have been computed

Figure 7.3: The CTE algorithm.

The complexity of CTE is time $O(ed^{tw+1})$ and space $O(ed^s)$ where e is the number of edges of the tree decomposition, tw is the tree-width, d is the largest domain size and s is the maximum separator size.

Example 7.3 *In the drawing below we show the tree decomposition of example 7.2 with the 2 CTE messages associated to edge* $\langle u, v \rangle$.



Once the messages have been sent we can compute the optimal cost of solving the problem in any of the two nodes. For example, in v *the minimum cost of the sum* $v1 + v2 + m_{\langle u,v \rangle}$ *is the optimal cost. To actually obtain the whole set of solutions we could sum* $v1 + v2 + m_{\langle u,v \rangle}$ *and* $u1 + u2 + m_{\langle v,u \rangle}$.

Let us now state a property that will help understanding the approximation algorithm based on CTE that we describe in the next section. Let $T(u, v)$ (resp. $T(v, u)$) denote the subtree of T containing the connected component containing vertex u (resp. v) after the removal of edge $\langle u, v \rangle$.

Property 7.4 $m_{\langle u,v \rangle}(t)$ *is equal to the minimum cost of extending tuple* t *to the subproblem formed by variables and constraints of clusters in* $T(u, v)$.

7.2.3 Mini Cluster-Tree Elimination

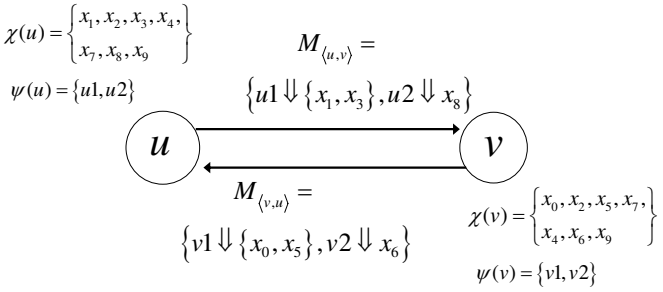
Mini-Cluster-Tree Elimination (MCTE(r)) is an approximation algorithm based on CTE. When the number of variables in a cluster is too high, it is not possible to compute a single message that captures the joint effect of all functions of the cluster plus all incoming messages due to memory limitations. In this case, MCTE(r) computes a lower bound of the problem by limiting by a constant r the arity of the functions sent in the messages. This is because we cannot afford to compute one single function that is of high arity and then project it.

A *MCTE(r) message*, noted $M_{\langle u,v \rangle}$, is a set of functions that approximate the corresponding CTE message $m_{\langle u,v \rangle}$ (namely $M_{\langle u,v \rangle} \leq m_{\langle u,v \rangle}$). It is computed as $m_{\langle u,v \rangle}$ but instead of summing all functions of set B (line 2 of CTE algorithm in Figure 7.3), it computes a partition $P = \{B_1, B_2, ..., B_p\}$ of B such that the sum of the functions in every B_i does not exceed arity r. Then it computes $M_{\langle u,v \rangle}$ from P by summing all functions in every partition and projecting out from each resulting function the variables not mentioned by node v. The MCTE(r) algorithm is obtained replacing line 3 of the CTE algorithm by the following lines,

3.1 $P \leftarrow partitioning(B, r)$
3.2 $M_{\langle u,v\rangle} \leftarrow \{(\sum_{f \in B_i} f) \Downarrow elim(u,v) \mid B_i \in P\}$

MCTE(r) time and space complexity is $O(d^r)$.

Example 7.4 *In the drawing below we show the tree decomposition of example 7.2 with the messages that MCTE(2) sends. The arity of the functions to compute is limited to 2. All functions in the constraint graph already exceed this arity thus we set the partition of functions such that each function is in a different partition. We set* $P = \{\{u1\}, \{u2\}\}$ *for sending message* $M_{\langle u,v\rangle}$*, and* $P = \{\{v1\}, \{v2\}\}$ *for sending message* $M_{\langle v,u\rangle}$*.*



Observe that now messages are sets of functions that does not exceed the specified arity.

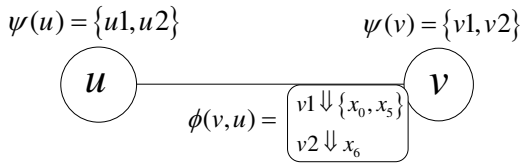
7.3 Tree Decomposition with Function Filtering

Now we integrate the idea of filtering into the CTE schema. First, we define a *filtering* tree-decomposition which adds a new labeling ϕ to a tree-decomposition that is used for filtering purposes.

Definition 7.4 [filtering tree-decomposition] *A filtering tree-decomposition of a WCSP is a tuple* $\langle T, \chi, \psi, \phi\rangle$ *where:*

- $\langle T, \chi, \psi\rangle$ *is a tree-decomposition as in definition 7.2.*
- ϕ *is a labeling.* $\phi(u,v)$ *is a set of functions associated to edge* $\langle u,v\rangle \in E$ *with scope included in* $sep(u,v)$*.* $\phi(u,v)$ *must be a lower bound of the corresponding* $m_{\langle u,v\rangle}$ *CTE message (namely,* $\phi(u,v) \leq m_{\langle u,v\rangle}$*).*

Example 7.5 *In the drawing below we show a tree decomposition of example 7.2 where we have added a possible filtering set of functions* $\phi(v,u)$*.*



The new algorithms CTEf and MCTEf(r) use a filtering tree decomposition. They are essentially equivalent to CTE and MCTE(r) except in that they use $\phi(u,v)$ as filtering functions before computing the message $m_{\langle u,v\rangle}$ or $M_{\langle u,v\rangle}$. The pseudo-code of CTEf is obtained by replacing line 3 of the algorithm by line,

3 $\quad m_{\langle u,v\rangle} \leftarrow \overline{\sum_{f\in B} f}^{\phi(u,v)} \Downarrow elim(u,v)$

Similarly for MCTEf(r) we replace line 3 by two lines,

3.1 $\quad P \leftarrow partitioning(B,r)$

3.2 $\quad M_{\langle u,v\rangle} \leftarrow \{\ \overline{(\sum_{f\in B_i} f)}^{\phi(u,v)} \Downarrow elim(u,v) \mid B_i \in P\ \}$

The effectiveness of the new algorithms depends on the ability of finding good lower bounds $\phi(u,v)$ for the messages $m_{\langle u,v\rangle}$ (resp. $M_{\langle u,v\rangle}$). If we use dummy lower bounds (i.e, $\phi(u,v)=\emptyset$, for all $\langle u,v\rangle \in E$), CTEf (resp. MCTEf(r)) is clearly equivalent to CTE (resp. MCTE(r)). It is important to note that the algorithms are correct as long as $\phi(u,v)$ is a true lower bound which can be computed with either a domain-specific or general technique (see [Givry et al., 1997] [Cabon et al., 1998] [Dechter et al., 2001] for a collection of general lower bound techniques). An option is to include in $\phi(u,v)$ all the original functions used to compute $m_{\langle v,u\rangle}$ properly projected,

$$\phi(u,v) = \{f \Downarrow S |\ f \in \psi(w), w \in T(u,v), S = var(f) - \chi(u)\}$$

Our CTEf and MCTEf implementations use this lower bound.

Example 7.6 *Consider the WCSP and the filtering tree decomposition of previous example 7.5. CTE solves the problem by sending message $m_{\langle u,v\rangle}$. Functions $u1$ and $u2$ of node u have to be added. Functions in $\phi(v,u)$ are a lower bound of $m_{\langle v,u\rangle}$ and are,*

$v1 \Downarrow \{x_0,x_5\}$	x_2	x_7
0	e	o
1	r	z
4	o	r
5	u	f
5	i	e
6	v	f
10	n	n

$v2 \Downarrow x_6$	x_4	x_9
1	o	e
2	e	o
2	t	o
3	o	t
6	s	x
7	x	s
10	t	n
11	n	t

First we compute $u1 + u2$. Since they have no variables in common $|u1 + u2| = 8.8 = 64$. Now we use the functions in $\phi(v,u)$ as filters. We compute $\overline{u1+u2}^{\phi(v,u)}$. By property 6.2 we can bring filtering to a deeper level $\overline{\overline{u1}^{\phi(v,u)} + \overline{u2}^{\phi(v,u)}}^{\phi(v,u)}$. First consider $\mathtt{ub} = \infty$. *Let's evaluate the result of $\overline{u1}^{\phi(v,u)}$ and $\overline{u2}^{\phi(v,u)}$. Values t,s,x of the domain of variable x_7 are not permitted by $v1 \Downarrow \{x_0,x_5\}$, so all remaining tuples including them are eliminated.*

Values z, r, f of the domain of x_4 are not permitted by $v2 \Downarrow x_6$, so all tuples including them are eliminated. The result of both filterings is:

$\overline{u1}^{\{v2 \Downarrow x_6\}} =$

u1	x_1	x_2	x_3	x_4
0	z	e	r	o
1	o	r	e	z
4	f	o	u	r
5	r	u	o	f
5	f	i	v	e
6	e	v	i	f
9	n	i	n	e
10	e	n	i	n

$\overline{u2}^{\{v1 \Downarrow \{x_0, x_5\}\}} =$

u2	x_7	x_8	x_9
1	o	n	e
2	e	n	o
2	t	w	o
3	o	w	t
6	s	i	x
7	x	i	s
10	t	e	n
11	n	e	t

Now $|\overline{u1 + u2}^{\phi(v,u)}| = 16$ and additional filtering causes no removals. So the application of Property 7.2 allows us to save $64 - 16 = 48$ tuples, a 75% of the initial memory.

Previous discussion assumes $\texttt{ub} = \infty$. *Now consider* $\texttt{ub} = 6$. *Lower values of* ub *causes further savings. For instance now two more tuples of $u1$ are eliminated because they have an associated cost greater or equal 6. Tuple five of $u1$ is also eliminated by filtering because the only matching tuple of $v2 \Downarrow x_6$ is eo which has cost 2 (this tuple is shown in the drawing), and $5 + 2$ reaches* ub. *The same thing happens with other tuples. See in the tables below the result of applying filtering when* $\texttt{ub} = 6$. *Tuples eliminated by filtering are crossed out.*

$\overline{u1}^{\{v2 \Downarrow x_6\}} =$

u1	x_1	x_2	x_3	x_4
0	z	e	r	o
1	o	r	e	z
4	f	o	u	r
5	r	u	o	f
5	f	i	v	e
6	e	v	i	f
9	n	i	n	e
10	e	n	i	n

+ 2 | e o

$\overline{u2}^{\{v1 \Downarrow \{x_0, x_5\}\}} =$

u2	x_7	x_8	x_9
1	o	n	e
2	e	n	o
2	t	w	o
3	o	w	t
6	s	i	x
7	x	i	s
10	t	e	n
11	n	e	t

+ 5 | i e

+ 3 | o t

Now, $|\overline{u1 + u2}^{\phi(v,u)}| = 1$ and additional filtering causes no removals. So the application of Property 7.2 allows us to save $8 - 1 = 7$ tuples, a 87% of the initial memory.

After the sum of functions in cluster u, we are ready now to send the message after projecting out the variables that do not belong to the separator: $m(u, v) = |\overline{u1 + u2}^{\phi(v,u)}| \Downarrow \{x_1, x_3, x_8\}$. We show in the drawing below the message between clusters u and v that we computed. The sent message is then added into $\psi(v)$.

$\psi(u) = \{u1, u2\}$

$m_{\langle u,v \rangle} = \overline{u1 + u2}^{\phi(v,u)} \Downarrow \{x_1, x_3, x_8\}$

$u \rightarrow v$

$\phi(v,u) = \{v1 \Downarrow \{x_0, x_5\},\ v2 \Downarrow x_6\}$

$\psi(v) = \{v1, v2, m_{\langle u,v \rangle}\}$

At this point we perform the sum of functions $v1 + v2 + m_{\langle u,v\rangle}$ *an as one tuple is obtained the problem has a solution.*

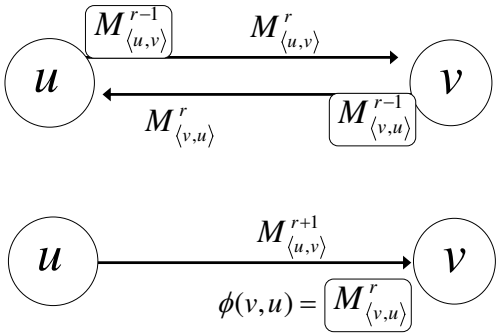
7.3.1 Iterative MCTE with filtering

The idea of iterative approximation for complete inference methods exists in the belief propagation framework (e.g. [Dechter and Mateescu, 2003]). Here we present an iterative version of MCTEf that exploits the advantages of function filtering for WCSP. MCTEf(r) bounds the arity of the functions it computes up to r. Higher r are harder to compute. The advantage of filtering is that it can help to compute a higher arity function with the help of a previous computed approximation of it. We now translate this idea to the defined filtering tree decomposition. An option is to include in $\phi(u, v)$ a message $M_{\langle v,u\rangle}$ from a previously computed execution of MCTE(r). We then obtain a recursive algorithm which naturally produces an elegant iterative approximating method that we call *iterative* MCTEf (IMCTEf). The idea is to execute MCTEf(r) using as lower bounds, $\phi(u, v)$, the messages $M^{r-1}_{\langle v,u\rangle}$ computed by MCTEf($r-1$) which, recursively, uses the messages $M^{r-2}_{\langle v,u\rangle}$ computed by MCTEf($r-2$), an so on. In Fig. 7.4 we show IMCTEf. It receives as input a WCSP instance, a filtering tree decomposition $\langle\langle V, E\rangle, \chi, \psi\rangle$ and the upper bound `ub`. Starting from dummy lower bounds (line 1), we execute MCTEf(r) for increasing values of r (line 4). The lower bounds computed by MCTEf(r) are used to detect and filter nogoods during the execution of MCTEf(r+1) (line 5). The algorithm follows this process until the exact solution is computed (namely, MCTEf does not break messages into smaller functions), or the available resources are exhausted.

procedure `IMCTEf`$(\mathcal{W}, \langle\langle V, E\rangle, \chi, \psi\rangle,$ `ub`$)$
1 **for each** $\langle u, v\rangle \in E$ **do** $\phi(u, v) \leftarrow \{\emptyset\}$
2 $r \leftarrow 1$
3 **repeat**
4 MCTEf(r, `ub`)
5 **for each** $\langle u, v\rangle \in E$ **do** $\phi(u, v) \leftarrow M_{\langle u,v\rangle}$
6 $r \leftarrow r + 1$
7 **until** exact solution or exhausted resources

Figure 7.4: IMCTE algorithm.

Example 7.7 *In the drawing below we show the two messages* $M^r_{\langle u,v\rangle}$ *and* $M^r_{\langle v,u\rangle}$ *of a* r *iteration of IMCTEf. After* $M^r_{\langle u,v\rangle}$ *is sent* $\phi(u, v)$ *is updated with it. Similarly after* $M^r_{\langle v,u\rangle}$ *is sent* $\phi(v, u)$ *is updated with it.*



Then in the next iteration $r+1$ the message $M^{r+1}_{\langle u,v\rangle}$ is sent and uses the updated $\phi(v,u)$.

7.4 Experimental Evaluation

We have tested CTE, CTEf, MCTEf(r) and IMCTEf on DIMACS dubois Max-Sat instances, Borchers Weigthed Max-Sat instances (described in Appendix A.2.5) and SPOT instances (described in Appendix A.2.2). Tree decompositions where computed using the ToolBar library (available at [Toolbar, 2003]) which uses a maximum cardinality search (MCS) heuristic [R.E.Tarjan and M.Yannakakis, 1984] for this purpose and visualized with LEDA library.

Experimenters are focused in evaluating the memory consumption of all algorithms, although time is reported to justify that in any case the improvements in memory are substituted by an unacceptable amount of CPU time.

We show that CTEf versus state of the art CTE uses less tuples to find the exact solution. We also show that, inside an approximation schema, MCTEf(r) exhausts resources at a smaller arity r and finds worst lower bounds than the iterative version IMCTEf where the previous messages of MCTEf(r) execution are used as filters.

The efficiency of inference algorithms strongly relies on achieving a good tree decomposition of the problem, ideally one with small maximum separator size, the bottleneck of CTE based algorithms. The number of edges of the decomposition is important for IMCTE algorithm because it has to store all the messages in both directions. Two instances and its corresponding tree decompositions are drawn in Fig. 7.5.

CTE always assumes that the memory spent by the algorithm is always equal to the worst case d^s for every sent message. So here we assume that we always use the upper bound of the problem to filter tuples. We want to prove that assuming that functions only store consistent tuples with the joint effect of applying filtering techniques to anticipate inconsistent tuples, the memory stored in the solving process is actually much less than the worst case spa+ce complexity assumed by CTE. When d^s is small usual CTE is feasible. For

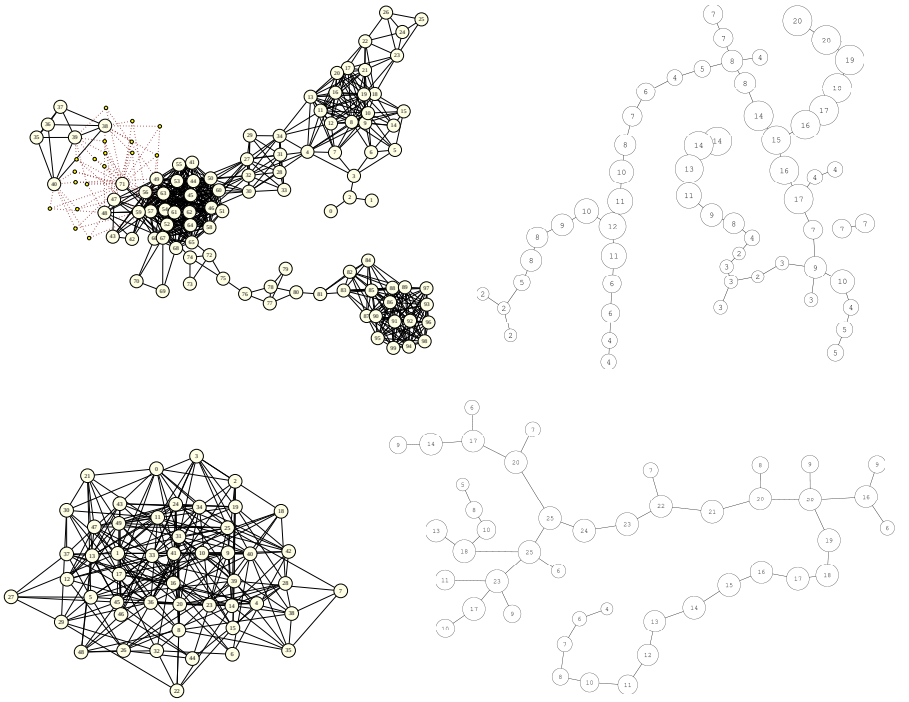


Figure 7.5: Left column: visualization of the SPOT404 and wp2250 instances where small dots represent ternary constraints. Right column: corresponding tree decomposition where each node is drawn proportionally to the number of variables $|\chi(v)|$ which is plotted inside the node.

example in instance dubois100 (which has a maximal separator size of $s = 3$) we can hardly see the improvement of CTEf. In instances where both CTE and CTEf are feasible (see the first Borcher's and first SPOT instances) the latter solves the problem with one order of magnitude less tuples. As the separator size increases CTE becomes at some point infeasible. This happens in wp2200 where $s = 19$ and however CTEf is still feasible in this instance spending 733k tuples. In instances wp2250 and wp2300 neither CTE nor CTEf can solve them, but the iterative version IMCTEf can solve it. In figure 7.6 the execution of the IMCTEf is plotted for instance wp2250. Each new execution with a bigger arity uses the previous computed messages as filtering functions. The total number of tuples for a particular execution is computed summing all the tuples of the sent messages. We can see that there is a critical arity where a maximum of tuples is generated. Iterations corresponding to last r's generate less tuples, this is due to the good increasing quality of previous messages. On the right of the same figure the computed lower bound for each arity is plotted.

							MCTEf(r)		IMCTEf		
	$\|X\|$	$\|C\|$	d	sep	CTE	CTEf	r	LB	r	LB	UB
dubois100	75	200	2	3	3k	2k					1^{opt}
wp2100	50	95	2	9	6k	1k					16^{opt}
wp2150	50	138	2	15	302k	40k					34^{opt}
wp2200	50	186	2	19	-	733k					69^{opt}
wp2250	50	233	2	24	-	-	23	71	25	96	96^{opt}
wp2300	50	261	2	26	-	-	22	84	26	132	132^{opt}
wp2350	50	302	2	30	-	-	21	129	21	159	212
wp2400	50	340	2	30	-	-	20	70	20	137	212
wp2450	50	378	2	31	-	-	20	130	20	187	257
wp2500	50	418	2	34	-	-	20	168	20	251	318
spot54	67	271	4	11	754k	16k					37^{opt}
spot29	82	462	4	14	-	63k					8059^{opt}
spot503	143	635	4	8	-	34k					11113^{opt}
spot404	100	710	4	20	-	306k					115^{opt}
spot505	240	2242	4	22	-	-	12	8044	15	19217	21254
spot42	190	1394	4	26	-	-	13	116001	15	127050	155051

Table 7.1: Columns: instance, number of variables, number of constraints, maximum domain size, maximum separator size, tuples consumed by CTE algorithm, tuples consumed by CTEf algorithm (- denotes exhausted memory), arity r reached by MCTE(r), LB computed by MCTE(r), arity r reached by IMCTEf, LB computed by IMCTE (before resources exhausted), optimal UB of the problem. When marked with (^{opt}) means that the instance is optimally solved by at least one of the algorithms.

When the separator size increases and instances cannot be optimally solved by any algorithm (CTE, CTEf, MCTEf, IMCTEf) the latter approximates the problem with a higher lower bound in all cases and reaches a higher arity in some of them.

When sending a particular message an important fact is how we sum all the available functions for that message. The direct way is to sum them two by two if the arity limit permits, applying filtering at each sum with all possible available filters. We must be careful with summing first functions with low cost, because they can quickly exhaust memory since almost no tuple reaches the level to be detected as inconsistent. So at this point some heuristics have been tested to select the pairs of functions to be summed. The two giving the best results are the following ones: (i) minimize mean cost of function tuples and (ii) minimum arity of the generated function. When minimum arity coincides then we minimize cost.

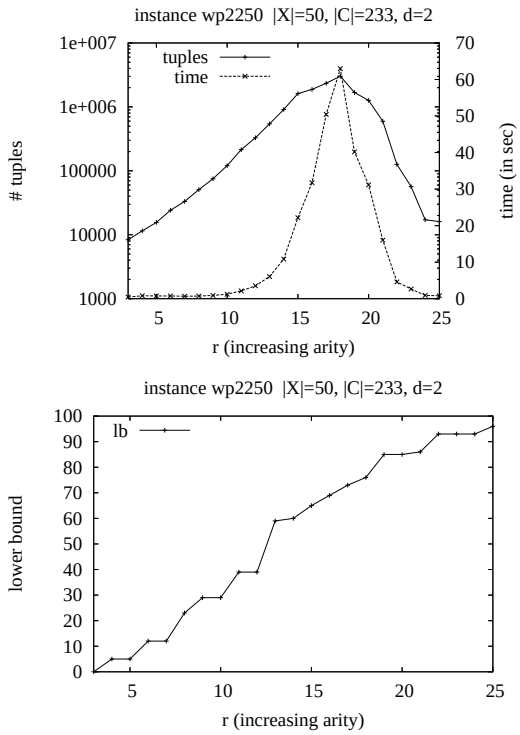


Figure 7.6: IMCTEf execution in Borchers instance wp2250. On the left, y-axis is the total number of computed tuples and time respectively. On the right, y-axis is the lower bound achieved for each arity r.

7.5 Related Work

The trade-off between time and memory has emerged in many areas of problem solving: [Culberson and Schaeffer, 1996] for example suggests the use of the so called *pattern databases* that stores the precomputed exact values of the heuristic. [Pearl, 1988, Darwiche, 2001] present the idea of caching that is also a way to store lower bound values in a search context. The algorithm BTD [Jégou and Terrioux, 2004] (outlined in Section 4.6.2) is another example of time/space trade-off. All these methods come from search contexts. In a complete inference context it is unusual to find algorithms where we are faced with a time/space trade-off (like the described IMCTE Section 7.3.1). Nevertheless similar ideas can be found. For example [Sachenbacher and Williams, 2005] defines a filtering operation called *sinking*. This operation is used inside a variant of branch and bound that maintains a set of assignments, instead of a single current assignment as usual branch and bound. Then the sinking operation is used to refine the assignments in this set that no longer consistent as they have

reached the upper bound. In our opinion the described algorithm has a great theoretical interest as it can behave as usual branch and bound if only one assignment is kept, but also it can behave as complete inference methods if all the assignments are kept.

In [Koster et al., 1999] a dynamic programming algorithm that exploits a tree decomposition is used to solve the Frequency Assignment Problem. In each cluster of the tree decomposition a set of solutions is computed and then refined with techniques that reminds us of the filtering operation. An iterative process based on merging values is also described.

Hyper tree decomposition

The tree decomposition for a CSP including n-ary constraints defined like definition 7.2 is equivalent to the so called hyper-tree decomposition (see [Gottlob et al., 2002]), called like that for the distinction of graphs (having only binary edges) and hyper-graphs (having edges of any arity). In [Gottlob et al., 2002] the so called hyper tree width is computed in terms of maximal number of hyperedges in a cluster. Following [Dechter, 2003] we use the concept of tree decomposition of a CSP referring to an hyper-tree decomposition of the hyper-graph formed by the functions of the CSP. We also extend this definition for WCSP imposing that every constraint must appear exactly once in all clusters.

7.6 Perspectives of future work

The iterative algorithm IMCTE exchanges memory consumption for time in a way that was not explored before as far as we know. In the successive iterations the algorithm increases the maximal arity up to which it is able to perform implicit computations reusing previous iterations to spend less memory. Limiting the arity of the computations is a possible way of approximating the problem, but as we saw in Chapter 3 when dealing with RDS techniques (see for an sketch of this idea Section 3.7.4) there exist other ways of approximation that could be useful to be exploited by this iterative schema. For example one could think of merging all values of a variable into one single value and at each iteration perform successive refinements of the domains of variables. In this way IMCTE would also be able to reuse previous iterations to reduce memory storage.

7.7 Conclusions

We have presented the idea of function filtering for WCSP case, where constraints are cost functions, inside a complete inference schema. This idea has been nicely combined with tree decomposition algorithms, producing new algorithms which experimentally require far less memory than their original counterparts. This represent an important step forward the practical applicability of complete inference for WCSP solving.

So far, the use of upper and lower bounds for WCSP solving was limited to search methods, namely branch-and-bound search. This is the first time that are used inside complete inference methods, to speed up their execution and to reduce their memory consumption. As results show, this combination has been quite beneficial. Combining other inference methods with bounds usage seems a promising line of research and deserves further exploration in the future.

Chapter 8

Conclusions

8.1 Conclusions

The soft constraints framework is a novel field with great potential because many problems can be expressed as a WCSP. In the search part of this thesis we have presented a set of new algorithms which improve the state-of-the-art methods for WCSP solving. In the inference part we have presented a set of complete inference algorithms that improve over existing ones for CSP and WCSP. In both parts a step further into narrowing the gap between these two families of methods is done. With respect to search, we present an algorithm that takes into account the global structure of a WCSP problem (the interaction among variables and constraints) and is able to exploit it and to bound the exponential complexity of the solving method by a structural parameter. This direction of work has originated many recent new approaches that seem essential to make search more efficient in particular structured problems. On the inference side, the two main contributions are devoted to reduce the memory spent by complete inference methods. The first idea is to factorize a constraint into smaller size constraints. The second consisted in anticipating combinations of values that will become inconsistent when extended to other parts of the problem. One could say that we are making use of the kind of look-ahead that search does to prune brunches, but we use it instead to reduce memory storage. Our final contribution is an iterative algorithm that performs successive approximations of the problem and reuses previous iterations to reduce the memory storage. This algorithm raises an interesting compromise between the time spent and the space used. Summarizing, we increased the applicability of complete inference methods into various CSP problems, something that was previously considered of theoretical interest only (although complete inference is the main solving method in Bayesian inference).

From this thesis we can extract the following conclusions:

- CSP and WCSP have many common features that can be exploited. In that context, we have extended to WCSP many developed techniques that

were originally developed for CSP. Pseudo-tree search for CSP is extended to WCSP. A new developed complete inference algorithm that includes what we call filtering is as well extended to WCSP.

- The gap between search and inference is narrowed. Firstly we presented a search algorithm that can exploit the global structure of the problem. We found this issue fundamental as it is a major drawback that search algorithms perform badly on problems that have particular global structures. Secondly we have presented complete inference methods that make use of a kind of look-ahead. These contributions can help in the understanding of the intrinsic nature of both families of methods.

- Solving simplifications of a problem to help in the resolution of the whole one has been proven effective in two contributions. On the search side, we explored Russian Doll Search (RDS) algorithm that solves the problem by first solving a simplification of it to help in the whole resolution. We further developed RDS techniques and proved that if we extract more information of the sequence of subproblems to solve we can then solve larger problems because we dispose of more accurate lower bounds. We spent more time to save time in the resolution of the whole problem. On the inference side, we presented an iterative algorithm that solves successive approximations of the problem and reuses previous iterations to delete combinations of values that do not belong to an optimal solution. This latter contribution is another example of how to spend more time to save space.

- A constraint can be factorized. Complete inference methods can exploit this fact to reduce the memory that algorithms spend. A new complete inference operation that eliminates binary domain variables that follows from the idea of factorization is presented. Thus the applicably of complete inference methods is extended.

As general conclusion we can say that we have widen the applicability of WCSP search with new algorithms that are more efficient than state of the art ones. We also have widen the applicability of complete inference methods as we provide some new algorithms that are more efficient in memory consumption and can be applied to larger problems.

8.2 Future Work

This work raises a number of issues that require further research. At the end of each Chapter we describe some lines of continuation of each subject of study. We summarize what we believe are the most important and we relate them in a more general way:

- The relation between Search and Inference Methods is of major interest. This thesis contributes in narrowing the gap between them, but still work remains to be done. We believe that a good direction of doing so is to

start from search algorithms that are able to exploit tree decompositions. In Chapter 4 we developed algorithm PT-PFC that works with a pseudo-tree decomposition. We show that it has the problem of bad quality local upper and lower bounds. We call this problem uncertainty gap. In the same Chapter we talk about BTD [Jégou and Terrioux, 2004], a search algorithm recently developed that combines search and tree decomposition, but still inherits the uncertainty gap problem. It could then be possible to solve this problem with Russian Doll Techniques as we did for pseudo-tree search in the same Chapter. Moreover BTD has the problem that if a tree decomposition with low maximal separator size is found it can spend a lot of memory. We believe at this point that there is a point of connection with the factorization techniques of Chapter 5 and the tree decomposition complete inference methods (MCTE and IMCTE) developed in Chapter 7. We believe that the developed techniques for both contributions can be applied into BTD to reduce the memory spent by the algorithm. This consideration tackles one of the main issues of this thesis.

- In the work of [Sachenbacher and Williams, 2005] it is defined an operation somehow similar to filtering (introduced in Chapters 6 and 7). It also introduces an algorithm that is an hybrid of search and inference but quite different from PT-PFC (in Chapter 4) and IMCTE (in Chapter 7). It would be interesting to establish a relation between these algorithms.

- Soft Arc Consistency (SAC) [Larrosa and Schiex, 2004] is a form of local incomplete inference that is used during search and is of recent development. SAC did not finally get in the scope of this thesis. A very interesting line of research which is the combination of Russian Doll Search Techniques (Chapter 3) with SAC. It is usually assumed that both techniques cannot be combined as they both refer to the constraints that link future variables, and so the same costs could be counted twice leading to an incorrect lower bounds. But we believe that one could use SAC to detect inconsistencies coming from binary constraints and RDS techniques to take into account cost coming from higher arity implicit dependencies. This idea is sketched in Section 3.7.3

- The fact that RDS is based on solving a simplification of the problem to be reused in the resolution of the whole problem, makes us think of using any arbitrary simplification: a constraint oriented RDS, a domain merging RDS, etc. This line of research could convert RDS in a whole family of methods.

Part III

Appendixes

Appendix A

Benchmarks

A.1 CSP benchmarks

A.1.1 Random Binary CSP

A binary random CSP class is defined by the 4-tuple $\langle n, m, p1, p2 \rangle$, where

- n is the number of variables.
- m is the number of values in each variable domain.
- $p1$ is the proportion of pairs of variables which have a constraint between them, i.e., the constraint density (there are $p1n(n-1)/2$ constraints/edges in the graph). Variables involved in a constraint are selected randomly.
- $p2$ is the proportion of pairs of values which are inconsistent for a pair of variables if there is a constraint between them, i.e., the constraint tightness (there are $p2m^2$ incompatible pairs of values in each constraint). The pair of values forbidden are selected randomly.

The problem generator ensures that all problems are generated with connected constraint graphs, so that the resultant problem cannot be decomposed into smaller components: disconnected graphs are simply discarded and new graphs are generated until a connected one is found. This random CSP model can be easily modifiable to generate problems with constraints of any arity. Random CSP are designed to test satisfaction algorithms, where all constraints are considered hard. Random instances where introduced in [Smith, 1994].

A.1.2 SAT

[Walsh, 2000] describes different SAT formulations into a CSP. The one that is more convenient to our purpose is the so called non-binary encoding. The CSP variables x_i have domains $D_i = \{t, f\}$. A non-binary contraint is posted between

those variables that occur together in a clause. This constraint has as nogoods those partial assignments that fail to satisfy the clause. For example, associated with clause $x_1 \vee x_2 \vee \neg x_3$ is a non-binary constraint on x_1, x_2 and x_3 that has a single nogood fft.

A.1.3 Schur's Lemma

The problem is to put n balls labeled from 1 to n into 3 boxes so that for any triple of balls i_1, i_2, i_3 with $i_1 + i_2 = i_3$, not all are in the same box. 23 is the greater number of balls that can be placed into three boxes. The problem can modeled in the following way: $3n$ binary variables each one indicating whether there is a ball in a particular box.

Variables: A variable x_{ij} is associated to each ball $i = 1, ..., n$ and each box $j = 1, 2, 3$ so we have a total of $3n$ variables.

Domain: Variables x_{ij} have binary domains indicating whether the ball i is in that box j or not.

Constraints: There are ternary hard constraints that forbid a ball being into more than one box. There also ternary hard constraints that forbid three balls i_1, i_2, i_3 that satisfy $i_1 + i_2 = i_3$ to be in the same box.

A.2 WCSP benchmarks

A.2.1 Random Problems

Binary random WCSP have been used in the context of optimization by many researchers [Meseguer and Sanchez, 2001, Larrosa and Schiex, 2003] and are the extension of random CSP [Smith, 1994]. A random WCSP is defined by a tuple $\langle n, m, p1, p2, w1, w2 \rangle$, where

- n is the number of variables.
- m is the number of values in each variable's domain.
- $p1$ is the proportion of tuples of variables which have a constraint among them, i.e., the constraint density (there are $\frac{p1*n(n-1)}{2}$ constraints/edges in the graph). Variables involved in a constraint are selected randomly.
- $p2$ is the proportion of tuples of values which are inconsistent for a constraint, the constraint tightness (there are $p2 * m^2$ incompatible tuples of values in each constraint). Values involved in a forbidden tuple are selected randomly. An inconsistent or forbidden tuple t for constraint c is such that $c(t) > 0$ (that is, every tuple that does not satisfies completely c).
- The cost of a forbidden tuple is selected randomly in the interval defined by $[w1...w2]$.

A.2.2 Earth Observation Satellite Management

The problem consists on planing a daily management of a satellite to take a set of images from at least one of three instruments. The problem can be cast as a satisfaction plus optimization with binary, ternary an one n-ary constraint [Bensana et al., 1999].

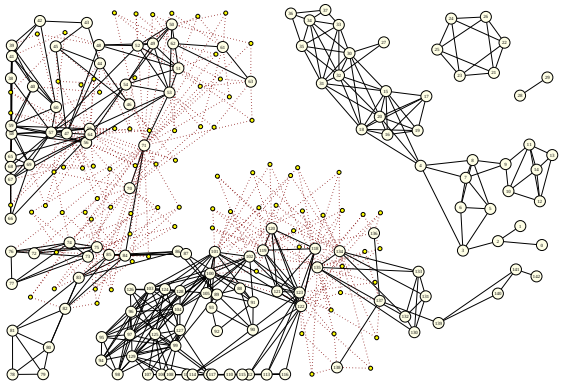


Figure A.1: Spot 503 instance. Small dots represent ternary constraints.

Problem Formulation

Variables: A variable is associated with each image. A positive integer weights each variable.

Domain: It is formed by the possible assignment of the different instruments to take the image: three possible values for a mono image and only one value for a stereo image.

Constraints: A set of hard binary constraints expresses the non overlapping and minimum transition time constraints. A set of hard binary or ternary constraints expresses the limitation of the instantaneous data flow through the satellite telemetry. An n-ary hard constraint involving all the variables expresses the limitation of the on-board recording capacity.

Optimization criteria: The weight of a partial assignment is defined as the sum of the weights of the assigned variables. A partial assignment is said feasible if and only if it satisfies all the hard constraints (with the n-ary constraint restricted to the assigned variables). The problem consists in finding a partial feasible assignment whose weight is maximum.

Solving Methods and Results

Systematic methods are able to optimally solve problem instances without any n-ary recording capacity constraint. However, they fail to do so when this type of constraint is included. All the proven optimal results have been obtained either using an ILP problem formalization and the CPLEX commercial software, or by using a Valued CSP formalization and Russian Doll Search. The original article of RDS [Verfaillie et al., 1996] implemented in LISP, reports very good results compared to Depth First Branch and Bound with backward and forward checking on several instances, and notices the negative influence of increasing graph bandwidth to the cpu solving time. The ILOG Solver has been tried on smaller instances [Lemaitre and Verfaillie, 1997]. All the approximating results have been obtained with Tabu Search algorithms. In an operational context, an hour is currently considered as a maximum time to decide which images will be taken the next day and how to take them. Although this constraint cannot be considered as hard in the context of this benchmark it may be important to be aware that a program taking more than one day, would not be very useful.

A.2.3 Radio Link frequency assignment problems (FAP) of CELAR)

The problem consists in assigning frequencies to a set of radio links defined between pairs of sites in order to avoid interferences. Each radio link is represented by a variable whose domain is the set of all frequencies that are available for this link. All the constraints are binary, non linear, and have finite domains. All problem instances have been built from a unique real instance with 916 links and 5744 constraints, and include both feasible (a solution exists satisfying all the constraints) and unfeasible (a solution exists satisfying all hard constraints and minimizing the cost of violating some of the soft constraints) [Cabon et al., 1999].

Problem Formulation

Variables: The links $i \in X$.

Domains: The finite set of frequencies available for each link D_i.

Constraints:

1. For each link $x_i \in X$ a frequency f_i has to be chosen from $D_i : f_i \in D_i$.
2. Two links can define a duplex link $|f_i - f_j| = d_{ij}$ d stands for distance.
3. Some links may already have a pre-assigned frequency $f_i = p_i$. There is a mobility cost for violating this soft constraint specified by m_i.
4. Two links may interfere together $|f_i - f_j| > d_{ij}$. There is a interference cost for violating this soft constraint specified by c_i.

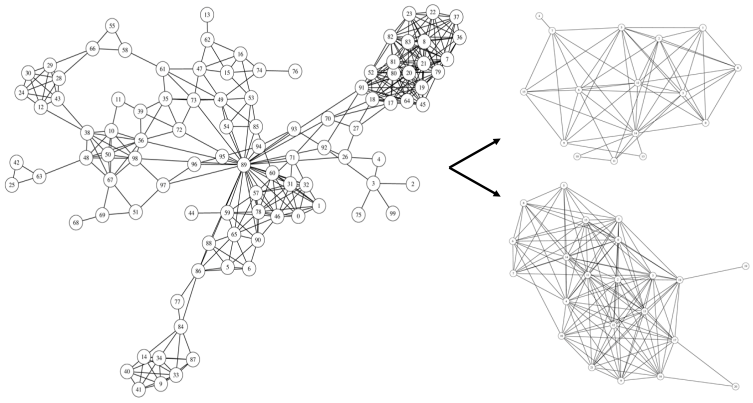


Figure A.2: On the left celar6 instance. On the top right celar6 subinstance 0, and celar 6 subinstance 4 below.

Constraints (1) and (2) are hard, while (3) and (4) are soft.

Several version of problems to solve can be defined but we focus on the Maximum Feasibility problem: if all the constraints cannot be satisfied simultaneously, find the assignment that minimizes the sum of all the violation cost. It can be cast as a WCSP $min(\sum c_{ij} violation(|f_i - f_j| > d_{ij}) + \sum m_{ij} violation(f_i = p_i))$.

Solving Methods and Results

There is a simplification pointed out by Thomas Schiex for eliminating all the hard equality constraints of the problem and reducing the number of variables by two (because of the bijective property of the equality). From now we will assume this simplification. We have checked the following solving approaches:

- Anytime Lower Bounds: In [Cabon et al., 1998] combinations of RDS with some iterative deepening techniques are shown to be very effective in the calculation of anytime lower bounds.
- Russian Doll Search: In [Givry et al., 1997] RDS was used to solve one of the hardest FAP instances in 32 days with a Sparc 5 workstation. The subinstances of Celar6 where optimally solved and the sum of this optimal costs was equal to best upperbound found at that moment for the global instance, so it was directly proven optimal. The enhancements of RDS techniques explained in this thesis solved the four subinstances in an order of 10^4 seconds on a Pentium IV computer [Meseguer and Sanchez, 2001, Meseguer et al., 2002].
- Pure Tree Decomposition Methods: In [Koster et al., 1999] the whole instance 6 is solved to optimality in no more than 3 hours using graph de-

composition techniques combined with a dynamic programming algorithm.

- Tree Decomposition plus Search: BTD algorithm was used to optimally solve subcelar6 instances in 10^5 seconds with a Pentium IV computer [Jégou and Terrioux, 2004].
- Search plus local soft consistency: Subinstances are solved using directed arc-inconsistency techniques in [Larrosa et al., 1999]. The CPU time given correspond to the time to prove optimality, that means the DAC algorithm is initialized with the optimal cost as upperbound. Latest developed local soft consistency techniques have also been tested with celar6 subinstances, not yet with satisfying results.

A.2.4 Combinatorial Auctions

Combinatorial auctions [Sandholm, 2002] are a special kind of auctions where bidders can bid on combinations of objects. Determining the winners so as to maximize the revenue is NP-complete. A generator of combinatorial auctions instances is described in [Cramton et al., 2006]. The instances can then be easily translated to WCSP.

Problem Formulation

Variables: A variable $x_i \in X$ represents a bid.

Domains: Variables have binary domains. If either we select that bid or not, the variable takes the value 1 or 0, $D_i = \{0, 1\}$.

Constraints:

- Binary hard constraints: if two bids share some object, they cannot be taken, they cannot both take value 1.
- Unary soft constraints: Every value 0 of a bid has an associated cost, that is the one that the auctioneer loses for not taking that bid.

Solving Methods

CATS [Cramton et al., 2006] is a generator of combinatorial auctions instances that has been used by many researches. The most used and efficient method seams to be integer linear programming.

A.2.5 Weighted Max-SAT

When a logical formula is unsatisfiable, the MAX-SAT problem tries to find the assignment that satisfies as many clauses as possible. We can assign a cost to each clause corresponding to the cost that we pay if we violate that clause. The objective is then to minimize the total pay-off. A weighted Max-SAT instance

is directly expressed as a WCSP as follows. WCSP variables x_i are the logical variables of the Max-SAT instance, with the domain $D_i = \{t, f\}$. Each clause C_k of the instance with cost w_k, generates a cost function which assigns 0 to those tuples satisfying C_k and assigns cost w_i to the only tuple violating C_k. When two cost functions involve the same variables, they can be added together. The WCSP solution, the complete assignment with minimum cost, corresponds to the solution of the Max-SAT instance. In [Givry et al., 2003] Max-SAT instances are solved using WCSP techniques. In [Givry et al., 2003] one can find several generated Weighted Max-SAT instances.

Appendix B

Search and nary constraints

Non-binary constraint are fundamental in WCSP formalization and resolution and they are present in many real-life problems. In this chapter we are concerned in how mentioned algorithms up to the moment can deal with explicit nary constraints representations. When Partial Forward Checking algorithm was introduced in Section 2.4.1 the look-ahead procedure that propagates costs between past and future variables was assumed to work only with binary constraints. Russian Doll Search and all its presented extensions of Chapter 3 do not need to make any assumption of the constraint arity. [Meseguer et al., 2001].

B.1 The binary case

For binary WCSP several lower bound computations have been used:

- PFC (see Definition 2.14, [Freuder and Wallace, 1992])

$$LB^{PFC}(t, F) = \texttt{cost}(t) + \sum_{x_j \in F} \min_{b \in D_j} \{ic_{jb}\}$$

- DAC [Wallace, 1995]

$$LB^{DAC}(t, F) = \texttt{cost}(t) + \sum_{x_j \in F} \min_{b \in D_j} \{ic_{jb}\} + \sum_{j \in F} \min_{b} \{dac_{jb}\}$$

- PFC-DAC [Larrosa and Messeguer, 1996]

$$LB^{PFC-DAC}(t, F) = \texttt{cost}(t) + \sum_{x_j \in F} \min_{b \in D_j} \{ic_{jb} + dac_{jb}\}$$

- RDS (see Definition 3.4 [Verfaillie et al., 1996])

$$LB^{RDS}(t, F = \{x_i, ..., x_1\}) = \texttt{cost}(t) + \sum_{x_j \in F} \min_{b \in D_j} \{ic_{jb}\} + rds_i$$

- Weighted AC [Affane and Bennaceur, 1998]

$$LB^{WAC}(t,F) = \texttt{cost}(t) + \sum_{x_j \in F} \min_{b \in D_j} \{ic_{jb} + \frac{1}{2} ac_{jb}\}$$

- PFC-MRDAC [Larrosa et al., 1999]

$$LB^{PFC-MRDAC}(t,F) = \texttt{cost}(t) + \sum_{x_j \in F} \min_{b \in D_j} \{ic_{jb} + dac_{jb}(G^F)\}$$

- SRDS (see Chapter 3 Section 3.2.1)

$$LB^{SRDS}(t,F,j) = \texttt{cost}(t) + \min_{a \in D_j} \{ic_{ja} + rds_{ja}\} + \sum_{x_k \in F, k \neq j} \min_{b \in D_j} \{ic_{jb}\}$$

- FSRDS (see Chapter 3 Section 3.3.1)

$$LB^{FSRDS}(t, F = \{x_i, ..., x_1\}, k) = \texttt{cost}(t) + \min_{a \in D_k} \{ic_{ka} + rds^i_{ka}\} + \sum_{x_l \in F, l \neq k} \min_{a \in D_l} \{ic_{la}\} \quad \forall x_k \in F$$

Note that the only difference between SRDS and FSRDS lower bound is that in the latter the combination of the ic's and rds contribution is done in the best table of lower bounds, the one of subproblem $\mathcal{W}^i$, where x_i is the first future variable.

B.2 The non-binary case

We will call C^P completely instantiated constraints, constraints where all his variables belong to the past variables in P. We will call C^F completely instantiated constraints, constraints where all his variables belong to the past variables F. We will call C^{PF} the constraints that link past and future variables. All sets obviously change during search. In the binary case, all constraints in C^{PF} have one future variable in their scope. This is no longer true in the non-binary case: a constraint $f \in C^{PF}$ may have more than one future variable in its scope. If we propagate the effect of one constraint f because of one assignment in all future variables, we may repeat costs in the future because the cost of the same inconsistency could be counted as many times as future variables are involved in the constraint. To prevent this, costs of inconsistencies of f have to be recorded in one of its future variables only. The same problem appears when considering a constraint $f' \in C^F$: inconsistencies of f' have to be recorded in one of its future variables, in order to allow for a safe lower bound computation as the addition of contributions of future variables. Therefore, for each $f \in C^{PF} \cup C^F$ we select one of its future variables as the only variable recording the costs of f inconsistencies. This variable, denoted as var_f, may change during the solving process

among the future variables of the constraint. In the Max-CSP context, this idea was presented in [Meseguer, 2000] (Section 4.2), at the ECAI-00 workshop *Modelling and Solving Problems with Constraints*. A similar approach was presented in [Regin et al., 2000b] (Section 4.1), as a poster at the CP-00 conference.

This problem occurs for ic_{ia} and dac_{ia} counters. From a graph point of view, the constraint hypergraph formed by $C^{PF} \cup C^F$ has to be directed, in the sense that each hyperedge points towards one of the nodes that it connects. The hyperedge representing constraint f points towards the node representing variable var_f. Denoting by G^{PF} the directed hypergraph formed by C^{PF}, and by G^F the directed hypergraph formed by C^F, we generalize binary ic_{ia} and dac_{ia} counters as follows.

Definition B.1 [inconsistency cost, Reversible DAC] *Given a WCSP $\mathcal{W} = \langle X, D, C \rangle$, the current assignment t, the set of future variables F, G^{PF} a directed hypergraph on C^{PF}, G^F a directed hypergraph on C^F,*

- Inconsistency costs*:*

$$ic_{ia}(G^{PF}) = \sum_{f \in C^{PF}, i = var_f} \min_{\forall_{t' \in f} | t \cdot t' \, is \; defined} f(t')$$

- Reversible DAC*:*

$$dac_{ia}(G^F) = \sum_{f \in C^F, i = var_f} \min_{\forall_{t' \in f} | t \cdot t' \, is \; defined} f(t')$$

We maintain the *ic*s and *dac* names. Observe, however, that in the non-binary case both counters record costs of directed inconsistencies, otherwise the cost of the same inconsistency could be recorded more than once. Keeping the meaning of $\texttt{cost}(P)$, ac_{ia} and rds as in the binary case all lower bounds presented in the previous section can be extended to the non-binary case. We show in Fig. B.1 two examples of their extensions. The main difference is that we have to take into account the new definition of *ic*'s. Let's recall that $LB^{RDS}(t, F, G^{PF})$ requires a static variable ordering.

As in the binary case, $dac_{ia}(G^F)$ counters can be maintained during search. In that case, we compute the minimum cost over constraints involving variable i taking into account that the current domains of every variable that can change during search because of value pruning.

B.2.1 Partial Forward Checking

A PFC algorithm to be used with the proposed non-binary lower bounds is presented in Fig. B.2, where a generic lower bound is computed by the function `LB`. It follows the PFC algorithm of [Larrosa et al., 1999]. First, it checks if no more future variables exists and returns the cost of the current assignment (lines 1). Otherwise, it selects a future variable x_i (line 2) and iterates on its feasible

$$LB^{PFC}(P,F,G^{PF}) = \texttt{cost}(P) + \sum_{x_i \in F} \min_{a \in D_i} \{ic_{ia}(G^{PF})\}$$

$$LB^{MRDAC}(P,F,G^{PF},G^{F}) = \texttt{cost}(P) + \sum_{x_i \in F} \min_{a \in D_i} \{ic_{ia}(G^{PF}) + dac_{ia}(G^{F})\}$$

Figure B.1: Lower bounds for non-binary WCSPs.

```
function PFC(t, F, D, C, ub, G^PF, G^F)
1   if F = ∅ return cost(t)
2   i ← choose-variable(F)
3   for each a ∈ D_i do
4     D' ← look-ahead(i, a, P, F, D, ub)
5     if LB(i, a, t, F, D', G^PF, G^F) < ub
6       newG^PF, newG^F ← GreedyOpt(G^PF, G^F)
7       if LB(i, a, P, F, D', G^PF, G^F) < ub
8         D' ← prune(i, a, t, F, D', ub, newG^PF, newG^F)
9         if ∅ ∉ D' then
10          ub' ← PFC(t · {⟨i, a⟩}, F, D', newG^PF, newG^F)
11          if ub' < ub then ub ← ub'
12  return ub
```

Figure B.2: Partial Forward Checking for non-binary constraints.

values (line 3). It then performs lookahead (line 4) and checks if the lower bound has reached the upper bound (line 5). Then a greedy optimization procedure is executed (line 6), redirecting the hypergraphs G^{PF} and G^F in order to increase the lower bound (getting the optimum redirection of hypergraphs is a NP-hard problem [Schiex, 1998], so we redirect hyperedges aiming at a *good* contribution to the lower bound). If the new lower bound does not reach the upper bound (line 7), it tries to remove value b in future variable j using the pruning value procedure (line 8). If no empty domain has been produced (line 9), the process goes on with the recursive call (line 10).

We have evaluated the performance of our PFC algorithm using the proposed lower bounds on random WCSP of arity 5. Results can be found in [Meseguer et al., 2001].

Bibliography

[Affane and Bennaceur, 1998] Affane, M. and Bennaceur, H. (1998). A weighted arc consistency technique for max-csp. In *Proc. 13th ECAI*, pages 209–213.

[Bacchus, 2002] Bacchus, F. (2002). Enhancing davis putnam with extended binary clause reasoning. In *AAAI/IAAI*, pages 613–619.

[Bahar, 1993] Bahar, R. (1993). Algebraic decision diagrams and their applications. In *Proceedings of the IEEE/ACM international conference on Computer-aided design*, pages 188–191.

[Bayardo and Miranker, 1995] Bayardo, R. and Miranker, D. (1995). On the space-time trade-off in solving constraint satisfaction problems. In *IJCAI*, pages 558–562.

[Bensana et al., 1999] Bensana, E., Lemaitre, M., and Verfaillie, G. (1999). Earth observation satellite management. *Constraints*, 4:293–299.

[Bertele and Brioschi, 1972] Bertele, U. and Brioschi, F. (1972). *Nonserial Dynamic Programming*. Academic Press.

[Bistarelli et al., 1995] Bistarelli, S., Montanari, U., and Rossi, F. (1995). Constraint solving over semirings. In *IJCAI (1)*, pages 624–630.

[Bryant, 1986] Bryant, R. (1986). Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, volume c-35 n8.

[Cabon et al., 1999] Cabon, B., Givry, S., Lobjois, L., Schiex, T., and Warners, J. (1999). Radio link frequency assignment problems. *Constraints*, 4:79–89.

[Cabon et al., 1998] Cabon, B., Givry, S., and Verfaillie, G. (1998). Anytime lower bounds for constraint violation minimization problems. In *Proceedings of the 4th Conference on Principles and Practice of Constraint Programming*, volume 1520, pages 117–131.

[Cramton et al., 2006] Cramton, P., Shoham, Y., and Steinberg, R. (2006). *A Test Suite for Combinatorial Auctions*, chapter 18. MIT Press.

[Culberson and Schaeffer, 1996] Culberson, J. and Schaeffer, J. (1996). Searching with pattern databases. In *Proc. CSCSI Canadian AI conference*, pages 402–416.

[Darwiche, 2001] Darwiche, A. (2001). Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397.

[Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *J. ACM*, 7(3):201–215.

[de Givry et al., 2006] de Givry, S., Schiex, T., and Verfaillie, G. (2006). Exploiting tree decomposition and soft local consistency in weighted csp. In *AAAI, Boston (MA) USA*, page to appear.

[Dechter, 1990] Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312.

[Dechter, 1999] Dechter, R. (1999). Bucket elimination: A unifying framework for reasoning. *Artifical Intelligence*, 113:41–85.

[Dechter, 2003] Dechter, R. (2003). Elsevier Science.

[Dechter et al., 2001] Dechter, R., Kask, K., and Larrosa, J. (2001). A general scheme for multiple lower bound computation in constraint optimization. In *Proceedings of the 6th Conference on Principles and Practice of Constraint Programming*, pages 346–360.

[Dechter and Mateescu, 2003] Dechter, R. and Mateescu, R. (2003). A simple insight into iterative belief propagation's success. In *UAI*, pages 175–183.

[Dechter and Mateescu, 2004] Dechter, R. and Mateescu, R. (2004). The impact of and/or search spaces on constraint satisfaction and counting. In *Proceedings of the Conference on Principles and Practice of Constraint Programming*, pages 731–736.

[Dechter and Pearl, 1987] Dechter, R. and Pearl, J. (1987). Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38.

[Doig and Land, 1960] Doig, A. and Land, A. (1960). An automatic method for solving discrete programming problems. *Econometrica*, 28:497.

[Freuder and Quinn, 1985] Freuder, E. and Quinn, M. (1985). Taking advantage of stable sets of variables in constraint satisfaction problems. *IJCAI*, pages 1076–1078.

[Freuder and Wallace, 1992] Freuder, E. and Wallace, R. (1992). Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70.

[Gaschnig, 1978] Gaschnig, J. (1978). Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisfying assignment problems. In *Proc. 2nd National Conference of the Canadian Society for Computational Studies of Intelligence*, pages 268–277.

[Gent et al., 1999] Gent, I., Stergiou, K., and Walsh, T. (1999). Decomposable constraints. In *New Trends in Constraints*, pages 134–149.

[Givry et al., 2003] Givry, S., Larrosa, J., Meseguer, P., and Schiex, T. (2003). Solving max-sat as weighted csp. In *CP*, pages 363–376.

[Givry et al., 1997] Givry, S., Verfaillie, G., and Schiex, T. (1997). Bounding the optimum of constraint optimization problems. In *Proceedings of the 3th Conference on Principles and Practice of Constraint Programming*, pages 405–419, Schloss Hagenberg, Austria.

[Gottlob et al., 2000] Gottlob, G., Leone, N., and Scarcello, F. (2000). A comparison of structural csp decomposition methods. *Artificial Intelligence*, 124(2):243–282.

[Gottlob et al., 2002] Gottlob, G., Leone, N., and Scarcello, F. (2002). Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627.

[Harlick, 1980] Harlick, E. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313.

[Jégou and Terrioux, 2004] Jégou, P. and Terrioux, C. (2004). Decomposition and good recording. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-2004)*, pages 196–200.

[Kask et al., 2006] Kask, K., Dechter, R., Larrosa, J., and Dechter, A. (2006). Unifying cluster-tree decompositions for reasoning in graphical models. *Artifical Intelligence*, page to appear.

[Koster et al., 1999] Koster, A., Hoesel, C., and Kolen, A. (1999). Optimal solutions for a frequency assignment problem via tree-decomposition. In *Lecture Notes in Computer Science*, volume 1665, pages 338–349. Springer.

[Larrosa, 2000] Larrosa, J. (2000). Boosting search with variable elimination. In *Proceedings of the Conference on Principles and Practice of Constraint Programming*, pages 291–305.

[Larrosa et al., 2002] Larrosa, J., Meseguer, P., and Sanchez, M. (2002). Pseudo-tree search with soft constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-2002)*.

[Larrosa et al., 1999] Larrosa, J., Meseguer, P., and Schiex, T. (1999). Maintaining reversible dac for max-csp. *Artificial Intelligence*, 107:149–163.

[Larrosa and Messeguer, 1996] Larrosa, J. and Messeguer, P. (1996). Exploiting the use of dac in maxcsp. In *Proceedings of the 2th Conference on Principles and Practice of Constraint Programming*, pages 308–322.

[Larrosa and Schiex, 2003] Larrosa, J. and Schiex, T. (2003). In the quest of the best form of local consistency for weighted csp. In *Proc. of the 18h IJCAI*.

[Larrosa and Schiex, 2004] Larrosa, J. and Schiex, T. (2004). Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159.

[Lemaitre and Verfaillie, 1997] Lemaitre, M. and Verfaillie, G. (1997). Daily management of an earth observation satellite: comparaison of ilog solver with dedicated algorithms for valued csp. In *Proceedings of the 3rd ILOG International Users Meeting*.

[Mackworth, 1977] Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8:99–118.

[Marinescu and Dechter, 2005] Marinescu, R. and Dechter, R. (2005). And/or branch and bound for graphical models. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI*, pages 224–229.

[Meseguer, 2000] Meseguer, P. (2000). Lower bounds for non-binary maxcsp. In *Modelling and Solving Constraint Problems Workshop ECAI-00*.

[Meseguer et al., 2003] Meseguer, P., Bouhmala, N., Bouzouba, T., Irgens, M., and Sanchez, M. (2003). Current approaches for solving overconstrained problems. *Constraints Journal*, 8:9–39.

[Meseguer et al., 2001] Meseguer, P., Larrosa, J., and Sanchez, M. (2001). Lower bounds for non-binary constraint optimization. In *Proceedings of Conference on Principles and Practice of Constraint Programming*, pages 317–331.

[Meseguer and Sanchez, 2001] Meseguer, P. and Sanchez, M. (2001). Specializing russian doll search. In *Proceedings of Conference on Principles and Practice of Constraint Programming*, pages 464–478.

[Meseguer et al., 2002] Meseguer, P., Sanchez, M., and Verfaillie, G. (2002). Opportunistic russian doll search. In *Proceedings of the 7th Conference on Principles and Practice of Constraint Programming*, pages 264–279.

[Minton et al., 1992] Minton, S., Johnston, M., Philips, A., and Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205.

[Pearl, 1988] Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc. San Mateo, California.

[Petit et al., 2001] Petit, T., Regin, J., and Bessiere, C. (2001). Specific filtering algorithms for over-constrained problems. In *Proceedings of Conference on Principles and Practice of Constraint Programming*, pages 451–463.

[Prosser, 1993] Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):262–267.

[R.Dechter and J.Pearl, 1989] R.Dechter and J.Pearl (1989). Tree clustering for constraint networks. *Artif. Intell.*, 38(3):353–366.

[Regin et al., 2000a] Regin, J., Petit, T., Bessiere, C., and Puget, J. (2000a). An original constraint based approach for solving over constrained problems. In *Proceedings of Conference on Principles and Practice of Constraint Programming*, pages 543–548.

[Regin et al., 2000b] Regin, J., Petit, T., Bessiere, C., and Puget, J. (2000b). An original constraint based approach for solving over constrained problems. In *In Proceedings of the 6th Conference on Principles and Practice of Constraint Programming*, pages 543–548.

[Régin, 1994] Régin, J.-C. (1994). A filtering algorithm for constraints of difference in csps. In *AAAI*, pages 362–367.

[R.E.Tarjan and M.Yannakakis, 1984] R.E.Tarjan and M.Yannakakis (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579.

[R.M.Haralick and L.G.Shapiro, 1979] R.M.Haralick and L.G.Shapiro (1979). The consistent labeling problem. *IEEE Trans. Pattern Anal. Machine Intell.*, PAMI-1:173–203.

[Rosenfeld et al., 1976] Rosenfeld, A., Hummel, R., and Zucker, S. (1976). Scene labeling by relaxation operations. *IEEE Transactions on Systems, Man and Cybernetics*, 6:420–433.

[Sabin and Freuder, 1994] Sabin, D. and Freuder, E. (1994). Contradicting conventional wisdom in consraint satisfaction. In *Proc. ECAI*, pages 125–129.

[Sabin and Freuder, 1997] Sabin, D. and Freuder, E. (1997). Understanding and improving the mac algorithm. In *CP*, pages 167–181.

[Sachenbacher and Williams, 2005] Sachenbacher, M. and Williams, B. (2005). Bounded search and symbolic inference for constraint optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI)*, pages 286–291.

[Sanchez et al., 2005a] Sanchez, M., Larrosa, J., and Meseguer, P. (2005a). Improving tree decomposition methods with function filtering. In *International Join Conference on Artificial Intelligence, IJCAI*, pages 1537–1538.

[Sanchez et al., 2005b] Sanchez, M., Larrosa, J., and Meseguer, P. (2005b). Tree decomposition with function filtering. In *Proceedings of the 11th Conference on Principles and Practice of Constraint Programming*, Sitges, Spain.

[Sanchez et al., 2004a] Sanchez, M., Meseguer, P., and Larrosa, J. (2004a). Improving the applicability of adaptive consistency. In *Proceedings of the 10th Conference on Principles and Practice of Constraint Programming*, Toronto, Canda.

[Sanchez et al., 2004b] Sanchez, M., Meseguer, P., and Larrosa, J. (2004b). Using constraints with memory to implement variable elimination. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-2004)*, Valencia, Spain.

[Sandholm, 1999] Sandholm, T. (1999). An algorithm for optimal winner determination in combinatorial auctions. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI99*, pages 542–547, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Sandholm, 2002] Sandholm, T. (2002). Algorithm for optimal winner determination in combinatorial auctions. *Artif. Intell.*, 135(1-2):1–54.

[Schaerf, 1999] Schaerf, A. (1999). A survey of automated timetabling. *Artif. Intell. Rev.*, 13(2):87–127.

[Schiex, 1998] Schiex, T. (1998). Maximizing the reversible dac lower bound in max-csp is np-hard.

[Schiex, 1999] Schiex, T. (1999). A note on csp graph parameters. *Technical report 1999/03, INRA*.

[Schiex, 2000] Schiex, T. (2000). Arc consistency for soft constraints. In *Proc. 6th CP*, pages 411–424.

[Schiex et al., 1995] Schiex, T., Fargier, H., and Verfaillie, G. (1995). Valued constraint satisfaction problems: Hard and easy problems. In *IJCAI (1)*, pages 631–639.

[Selman et al., 1992] Selman, B., Levesque, H., and Mitchell, D. (1992). A new method for solving hard satisfiability problems. In Rosenbloom, P. and Szolovits, P., editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California. AAAI Press.

[Shapiro and Haralick, 1981] Shapiro, L. and Haralick, R. (1981). Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:504–519.

[Smith, 1994] Smith, B. (1994). Phase transition and the mushy region in constraint satisfaction problems. In *Proc. of ECAI*, pages 100–104.

[Toolbar, 2003] Toolbar (2003). http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/softcsp.

[Tounsi and David, 2002] Tounsi, M. and David, P. (2002). Successive Search Methods for Solving Constraint and Optimization Problems. *International Journal of Artificial Intelligence Tools*, 11(3).

[Verfaillie et al., 1996] Verfaillie, G., Lemaître, M., and Schiex., T. (1996). Russian doll search for solving constraint optimization problems. In *Proc. 13th AAAI*, pages 181–187.

[Wallace, 1995] Wallace, R. (1995). Directed arc consistency preprocessing. In *Selected papers from the ECAI-94 Workshop on Constraint Processing*, volume 923, pages 121–137. Springer.

[Walsh, 2000] Walsh, T. (2000). Sat vs. csp. In *In Proceedings of the 6th Conference on Principles and Practice of Constraint Programming*, pages 441–456.