Distributed Constraint Optimization Related with Soft Arc Consistency

Universitat Autònoma de Barcelona

Patricia Gutiérrez Faxas
Departament de Ciències de la Computació
Universitat Autònoma de Barcelona

Director: **Pedro Meseguer González**
Tutor: **Josep Puyol-Gruart**

A thesis submitted for the degree of
PhD in Computer Science

Work done at
Institut d'Investigació en Intel.ligència Artificial
Consejo Superior de Investigaciones Científicas
September, 2012

Abstract

Distributed Constraint Optimization Problems (DCOPs) can be used for modeling many multi-agent coordination problems. DCOPs involve a finite number of agents, variables and cost functions. The goal is to find a complete variable assignment with minimum global cost. This is achieved among several agents handling the variables and exchanging information about their cost evaluation until an optimal solution is found. Recently, researchers have proposed several distributed algorithms to optimally solve DCOPs. In the centralized case, techniques have been developed to speed up constraint optimization solving. In particular, search can be improved by enforcing soft arc consistency, which identifies inconsistent values that can be removed from the problem. Some soft consistency levels proposed are AC, FDAC and EDAC.

The goal of this thesis is to include soft arc consistency techniques in DCOP resolution. We show that this combination causes substantial improvements in performance. Soft arc consistencies are conceptually equal in the centralized and distributed cases. However, maintaining soft arc consistencies in the distributed case requires a different approach. While in the centralize case all problem elements are available in the single agent performing the search, in the distributed case agents only knows some part of the problem and they must exchange information to achieve the desired consistency level. In this process, the operations that modify the problem structures should be done in such a way that partial information of the global problem remains coherent on every agent.

In this thesis we present three main contributions to optimal DCOP solving. First, we have studied and experimented with the complete solving algorithm BnB-ADOPT. As result of this work, we have improved it to a large extent. We show that some of BnB-ADOPT messages are redundant and can be removed without compromising optimality and termination. Also, when dealing with cost functions

of arity higher than two, some issues appear in this algorithm. We propose a simple way to overcome them obtaining a new version for the n-ary case. In addition, we present the new algorithm ADOPT(k), which generalizes the algoritms ADOPT and BnB-ADOPT. ADOPT (k) can perform a search strategy either like ADOPT, like BnB-ADOPT or like a hybrid of both depending on the k parameter.

Second, we have introduced soft arc consistency techniques in DCOPs, taking BnB-ADOPT$^+$ as our base solving algorithm. During the search process, we enforce the soft arc consistency levels AC and FDAC, under the limitation that only unconditional deletions are propagated, obtaining important benefits in communication and computation. We enforce FDAC considering multiple orderings of the variables obtaining savings in communication. Also, we propose DAC by token passing, a new way to propagate deletions during distributed search. Experimentally, this strategy turned out to be competitive when compared to FDAC.

Third, we explore the inclusion of soft global constraints in DCOPs. We believe that soft global constraints enhance DCOP expressivity. We propose three different ways to include soft global constraints in DCOPs and extend the solving algorithm BnB-ADOPT$^+$ to support them. In addition, we explore the impact of soft arc consistency maintenance in problems with soft global constrains.

Experimentally, we measure the efficiency of the proposed algorithms in several benchmarks commonly used in the DCOP community.

Acknowledgements

First of all, I would like to thank my advisor Pedro Meseguer for counting on me at the beginning of this project, for his persevering guidance and his very much helpful experience. He has transmitted me calmness and confidence in several occasions and has crucially contributed to my growth as a scientist and as a professional.

I want to thank the members of the jury Christian Bessiere, Javier Larrosa and William Yeoh for their precious time. Christian Bessiere was kind enough to receive me at the LIRMM Institute in Montpellier two years ago. I appreciate very much our joint work and collaboration. It has been a real treat working with him, his smile and quick mind makes everything look easy. Javier Larrosa is one of the main references of this thesis and his valuable work has been fundamental in a great portion of it. William Yeoh has been an excellent feedback for me, I want to thank him for our stimulating and fruitful work, for all his helpful comments and suggestions.

I want to thank Thomas Schiex for receiving me at the INRA Institute in Toulouse last year, for being patient and generous with all my questions and for our collaborations.

I want to thank all the IIIA people for these great four years and for making me feel at home. I feel very lucky and grateful to have had the chance to meet great researches, PhD students and human beings. These years in the IIIA have exceeded all my expectations. I want to thank the administrative staff. I want to thank Carles Sierra for his fabulous costelladas! And specially to Ramón Lopez de Mántaras, for helping me in more than one crucial moment.

Finally, I want to thank my family for being confident and supportive with every decision in my life. Specially I want to thank my husband Yanko, because I am certain that without his help this thesis would have not been possible. He helped

me to concentrate in every deadline, encouraged me whenever I felt down, and cheered with me every little accomplishment.

Thank you all!

Contents

1

Introduction

1.1 Problem Statement

A classical problem in computer science is to optimize an aggregation of cost functions involving several variables and domain values. These problems are widely used in a variety of decision making applications. Many real life problems can be modeled as a collection of cost functions over a set of variables –representing constraints or penalty relations– that define a cost for every variable assignment or combination of variable assignments. Such problems are called *Constraint Optimization Problems* (COP). A constraint optimization algorithm is a solver able to find the optimal assignment for every variable in a COP such that the resulting cost is minimal.

Recently, there is an increasing interest in solving constraint optimization problems in a distributed way. When the variables and cost functions of the problem are not centralized in a single hardware but information is distributed among several independent and automated agents this is called a *Distributed Constraint Optimization Problem* (DCOP) (Modi et al., 2005). There are several cases where information requires to be distributed, for example consider problems where data is naturally distributed in different devices and the cost of translation is high, or problems where some information of the decision variables is desired to keep private. Then, this kind of problems commonly require a distributed resolution rather than a centralized solving. In these cases, variables, domain values and cost functions are distributed among several agents. These agents explore the search space assigning variables and exchanging messages about their cost evaluation in order to cooperate towards a global optimal solution.

DCOPs provide a framework for modeling many multi-agent coordination tasks used in

many practical applications. Some of them are meeting scheduling (Maheswaran et al., 2004), sensor networks (Jain et al., 2009; Stranders et al., 2009), traffic control (Junges and Bazzan, 2008), allocating tasks in a teamwork model (Schurr et al., 2005), power networks (Miller et al., 2012; Petcu and Faltings, 2008), among others.

Several distributed search algorithms have been proposed to optimally solve DCOPs. Some of them are: Synchronous Branch and Bound (SBB) (Hirayama and Yokoo, 1997), Asynchronous Distributed Constraint Optimization with Quality Guarantees (ADOPT) (Modi et al., 2005), Dynamic Programming Optimization (DPOP) (Petcu and Faltings, 2005), Asynchronous Partial Overlay (OptAPO) (Mailler and Lesser, 2006), No Commitment Branch and Bound (NCBB) (Chechetka and Sycara, 2006), Asynchronous Forward Bounding (AFB) (Gershman et al., 2009), Branch-and-Bound ADOPT (BnB-ADOPT) (Yeoh et al., 2010), among others.

Distributed algorithms can be either complete (they assure to find the optimal solution) or incomplete (they sacrifice optimality to obtain fast solutions). Also, they can be partially centralized (some distributed information is centralized) or decentralized. In this thesis, we focus on complete decentralized algorithms, which may have different levels of synchronization. In completely synchronous algorithms, only one agent is active at any time so agents wait for other agents execution to become active. In completely asynchronous algorithms, every agent is active at any time. Asynchronous algorithms have a degree of parallelism which can be an advantage in a distributed setting. In addition, they are robust because if some agent is disconnected the algorithm is still able to provide a solution for the connected part. On the other hand, they are usually more complex and they need to exchange a large number of messages to maintain coordination, also they have to deal with obsolete information.

In this thesis we consider BnB-ADOPT (Yeoh et al., 2010) as our base solving algorithm for DCOPs. Our interest for this algorithm comes from the fact that it is complete, requires only polynomial memory, it is asynchronous and communication is limited to neighboring agents.

1.2 Thesis Goal

DCOPs are NP-hard, so an exponential time is needed in the worst case to find the optimal solution. This severely limits the application of existing solving approaches. In the centralized case, several techniques have been developed to speed up constraint optimization solving. In particular, search can be improved by enforcing soft arc consistency, which identifies inconsistent values that can be removed from the problem. Several soft arc consistency levels have

been proposed in centralized (Cooper et al., 2008; de Givry et al., 2005; Larrosa and Schiex, 2003). By enforcing them it is possible to detect sub-optimal values that can be removed. As result, the domain of variables is reduced and the search space becomes smaller although more computational work must be done to maintain consistency. Globally, the overall effect has been very beneficial for efficiency, specially considering the first levels of soft arc consistency.

The goal of this thesis is to include soft arc consistency techniques in DCOP resolution. Such as it happens in the centralized case, we expect that this combination would cause performance improvements.

Soft arc consistencies are conceptually equal in the centralized and distributed cases. However, maintaining soft arc consistencies during distributed search requires a different approach. While in the centralize case all problem elements are available in the solver, in the distributed case agents only know some part of the problem and must exchange information in order to achieve the desired consistency level. In this process, the operations that modify the problem structures should be done in such a way that partial information of the global problem remains coherent on every agent. Maintaining soft arc consistencies during search must not compromise optimality and termination of the solving algorithm.

1.3 Contributions

In this thesis we present three main contributions to optimal DCOP solving, which are:

- **Contributions to Distributed Search**: We have studied and experimented with the complete solving algorithm BnB-ADOPT. As result of this work, we have improved it to a large extent, as explained in the following. On its execution BnB-ADOPT exchanges a large number of messages, which is a major drawback for its practical use. Aiming at increasing BnB-ADOPT efficiency, we show that some of these messages are redundant and can be removed without compromising optimality and termination. Removing most of those redundant messages we obtain the new version BnB-ADOPT$^+$. When tested on commonly used benchmarks, BnB-ADOPT$^+$ obtains large reductions in the number of messages sent by the agents –which is divided by a number often larger than 3– and moderate reductions in computation. A comparison with other state-of-the-art DCOP algorithms is also presented, showing the competitiveness of BnB-ADOPT$^+$.

BnB-ADOPT was initially defined for binary cost functions (involving only two variables). When dealing with cost functions of arity higher than two, some issues appear considering termination and efficiency. We propose a simple way to avoid these issues obtaining the new version n-ary BnB-ADOPT$^+$. Experimental results show the benefits of the proposed approach with respect to the original algorithm.

In addition, we present the new algorithm ADOPT(k), which generalizes ADOPT and BnB-ADOPT. These two algorithms share similar data and message structures, but differ on their search strategies: the former uses best-first search and the latter uses depth-first branch-and-bound search. ADOPT(k) generalizes ADOPT and BnB-ADOPT in the following way. Its behavior depends on the k parameter. It behaves like ADOPT when $k = 1$, like BnB-ADOPT when $k = \infty$ and like a hybrid of ADOPT and BnB-ADOPT when $1 < k < \infty$. We prove that ADOPT(k) is a correct and complete algorithm and experimentally show that ADOPT(k) outperforms ADOPT and BnB-ADOPT in terms of computation and communication on several benchmarks. Additionally, ADOPT(k) provides a good mechanism for balancing the trade-off between runtime and network load.

- **Connection with Soft Arc Consistency**: We have introduced soft arc consistency techniques in DCOPs, taking BnB-ADOPT$^+$ as our base solving algorithm. During the search process, we enforce the soft arc consistency levels AC and FDAC, under the limitation that only unconditional deletions are propagated. As result, agents are able to detect and unconditionally delete sub-optimal values. Unconditional deletions are propagated to neighbor agents, this propagation may generate new deletions on neighbors, that will also be propagated, reducing the search space. We present the new algorithms BnB-ADOPT$^+$- AC and BnB-ADOPT$^+$-FDAC. Experiments on several benchmarks show that maintaining AC level (BnB-ADOPT$^+$-AC) obtains a decrement in the number of messages exchanged and also in computation. Maintaining FDAC level (BnB-ADOPT$^+$-FDAC) enhances this reduction. Although agents need to perform more local computation to maintain consistency and some new messages are introduced to propagate deletions, this is largely compensated by a decrement in the number of messages used to solve the problem and as result, the computational effort shows important reductions as well.

Moving to higher consistency levels, agents need to have a wider knowledge about the global problem. Stronger consistency levels require agents to know more information about other agents. This may compromise privacy, which is an issue to resolve. As alternative, we propose the following two approaches, combined with the solving algorithm BnB-ADOPT$^+$. The first one is to enforce FDAC consistency level on multiple representations following different orderings. Inconsistent values can be detected in any of the ordering causing more deletion opportunities. Experimental results show significant savings in communication and a higher effort in computation, since more work must be done to maintain FDAC in every ordering. Our second approach proposes a heuristic form of propagation which we call DAC by token passing. This strategy does not maintain any soft local consistency property, so theoretically it can not be compared to AC or FDAC. However experimentally it turned out to be competitive when compared with FDAC considering communication and computational effort.

- **Including Global Constraints in DCOPs**: We explore the inclusion of soft global constraints in DCOPs, which have associated a specific semantic. A global constraints is a class of constraint defined on an unbounded number of variables. In centralized problems, they have been largely studied. However, to the best of our knowledge, no connection between DCOPs and soft global constraints have been established so far. We propose three different ways to include soft global constraints in DCOPs and extend the solving algorithm BnB-ADOPT$^+$ to support them. In addition, we explore the impact of soft arc consistency maintenance in problems with global constrains. In this case, performance can be improved, since often soft arc consistency can be achieved more efficiently when global constraints are involved than when working with an equivalent decomposition in smaller, fixed arity constraints.

1.4 Thesis Structure

This thesis is organized as follows.

Chapter 1: This Chapter introduces Distributed Constraint Optimization Problems, which are the target problems of this thesis. It also describes the main goal of this thesis and its contributions.

Chapter 2: This Chapter provides a background for the reader where we specify the main definitions and notation used in the following Chapters and summarize related work made by other authors.

Chapter 3: This Chapter details our contributions to the distributed search algorithm BnB-ADOPT for optimal DCOP solving. An important part of its content has been published in:

- Patricia Gutierrez, Pedro Meseguer *"Removing Redundant Messages in N-ary BnB-ADOPT"*, Journal of Artificial Intelligence Research (JAIR), in press, 2012
- Patricia Gutierrez, Pedro Meseguer, William Yeoh *"Generalizing ADOPT and BnB-ADOPT"*, Proc. of 22nd International Joint Conference on Artificial Intelligence (IJCAI-2011), pp. 554-559, 2011
- Patricia Gutierrez, Pedro Meseguer *"Saving Redundant Messages in BnB-ADOPT"*, Proc. of 24th American Conference on Artificial Intelligence (AAAI-2010), AAAI Press, pp. 1259-1260, 2010

Chapter 4: This Chapter details our contributions connecting soft arc consistency with DCOPs. An important part of its content has been published in:

- Patricia Gutierrez, Pedro Meseguer *"A Novel Way to Connect BnB-ADOPT$^+$ with Soft AC"*, Proc. of 20th European Conference on Artificial Intelligence (ECAI-2012), pp. 903-904, 2012
- Patricia Gutierrez, Pedro Meseguer *"Improving BnB-ADOPT+-AC"*, Proc. of 11th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2012), pp. 273-280, 2012
- P. Gutierrez, P. Meseguer *Connecting BnB-ADOPT with Soft Arc Consistency: Initial Results*. Recent Advances in Constraints. Lecture Notes in Artificial Intelligence 6384, 19-37, ed. J. Larrosa, B. O'Sullivan, 2011.
- Patricia Gutierrez, Pedro Meseguer *"Enforcing Soft Local Consistency on Multiple Representations for DCOP Solving"*, Proc. of CP 2010 workshop: Preferences and Soft Constraints, pp. 98-113, 2010.

- Patricia Gutierrez, Pedro Meseguer *"BnB-ADOPT+ with Several Soft AC Levels"*, Proc. of 19th European Conference on Artificial Intelligence (ECAI-2010), pp. 67-72, 2010.

Chapter 5: This Chapter details our contributions including soft global constraints in DCOPs. An important part of its content has been published in:

- Christian Bessiere, Patricia Gutierrez, Pedro Meseguer *"Including Soft Global Constraints in DCOPs"*, 18th International Conference on Principles and Practice of Constraint Programming (CP-2012), in press, 2012
- Christian Bessiere, Ismel Brito, Patricia Gutierrez, Pedro Meseguer *"Soft Global Constraints in Distributed Constraint Optimization"*, AAMAS 2012 workshop: International Workshop on Optimization in Multi-Agent Systems, pp. 43–50, 2012

Chapter 6: This Chapter provides conclusions and future work.

Appendix A: This Appendix presents results for saving redundant messages in the ADOPT algorithm. An important part of its content has been published in:

- Patricia Gutierrez, Pedro Meseguer *"Saving Messages in ADOPT-based Algorithms"*, Proc. of AAMAS 2010 workshop: Distributed Constraint Reasoning, pp. 53-64, 2010.

Appendix B: This Appendix presents results for introducing global constraints in Distributed Constraint Satisfaction Problems. An important part of its content has been published in:

- Christian Bessiere, Ismel Brito, Patricia Gutierrez, Pedro Meseguer *"Global Constraints in Distributed Constraint Satisfaction"*, Proc. of 11th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2012), pp. 1263-1264, 2012

Since the work contained in the above publications also appears in this document (in some cases with more detail), these publications are not cited in the following Chapters.

2

Background

In this Chapter we summarize the main concepts and algorithms that will be used as reference in the rest of this thesis. We do not aim at covering the basis of constraint reasoning, topics that can be found in books containing most of constraint aspects, as in (Dechter, 2003; Rossi et al., 2006).

Here, we focus on constraint optimization. First, we consider constraint optimization in the centralized case, where all the information of the problem instance is contained in a single agent. After describing the basic methods for solving these problems, we present several soft local consistencies. These soft local consistencies, when maintained inside a branch and bound search, allow to detect and remove inconsistent values, leading to efficiency gains. Second, we present distributed constraint optimization, describing some of the most relevant solving algorithms. We also discuss some issues regarding the experimental evaluation of distributed algorithms.

2.1 Constraint Optimization Problems

A *Constraint Optimization Problem* (COP) involves a finite set of variables with finite domain and a set of cost functions (Dechter, 2003). Variables are related by cost functions, which are also called soft constraints. [1] Cost functions are defined over a subset of variables and they specify the cost of value assignments in this subset. Costs are positive natural numbers (including 0 and ∞). Formally, a *COP* is a tuple (*X, D, C*) 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 variable x_i takes values in D_i.
- C is a set of cost functions; $f \in C$ specifies the cost of every combination of values on the ordered set of variables $var(f) = (x_1, \ldots, x_r)$, that is:
  $f : \prod_{j=1}^{r} D(x_j) \mapsto N \cup \{0, \infty\}$. The arity of f is $|var(f)|$.

A cost function f evaluated on a particular value tuple t gives the cost of assigning the values of t in the variables $var(f)$. The cost of a complete assignment (involving all variables) is the sum of all cost functions evaluated on that assignment. A *solution* is a complete tuple with acceptable cost, and it is *optimal* if its cost is minimal.

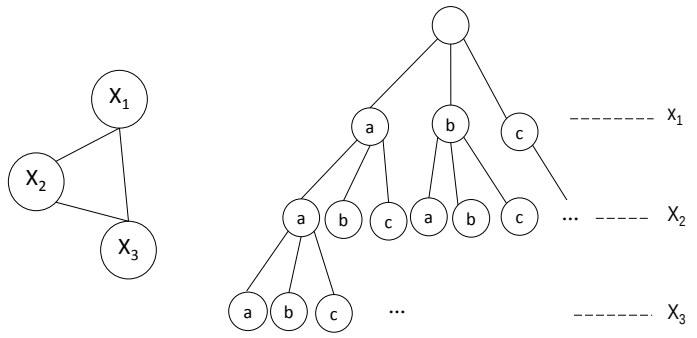
COP generalizes the Constraint Satisfaction Problem framework (CSP) in which complete value assignments (involving all problem variables) are characterized as satisfactory or unsatisfactory. In CSPs, all constraints must be satisfied in the final solution so it does not model problems where solutions have degrees of quality cost. In COPs, satisfactory tuples have 0 cost while unsatisfactory tuples (completely forbidden tuples) have ∞ cost. Intermediate costs are associated to tuples which are neither completely satisfactory nor completely forbidden.

2.1.1 COP Representations

A COP instance can be represented by a *constraint graph*, where nodes in the graph correspond to variables and edges connect pairs of variables appearing in the same cost function. This corresponds with the primal graph in (Dechter, 2003). Variables connected by the same cost functions (connected by the same edge in the constraint graph) are called neighbor variables.

[1] Strictly speaking, we are considering the weighted model (Meseguer et al., 2006)

The search space of a COP can be represented in a tree structure called *search tree*, where each level of the tree corresponds to a variable and each node to a variable assignment. A COP solution is a complete assignment of variables represented as a branch from the root to a leaf node in the search tree. The search space can be explored in different ways, in particular with a best-first search strategy or with a depth-first search strategy, following a best-first traversal or a depth-first traversal of the search tree. In Figure 2.1, a COP with variables x_1, x_2 and x_3, domains $\{a, b, c\}$ and cost functions between every pair of variables is represented by its constraint graph, along with the search tree generated by exploring its search space.



(a) Constraint Graph (b) Search Tree

Figure 2.1: COP Constraint Graph and Search Tree

2.1.2 Solving Algorithms

Several methods have been proposed to optimally solve COPs (Meseguer et al., 2006). In the following we describe two of the most known: Depth-First Branch and Bound, a search method with exponential complexity in time but linear in space, and Bucket Elimination, a complete inference method with exponential complexity, both in time and space, in the induced width of the constraint graph.

2.1.2.1 Systematic Search: Branch and Bound

Depth-First Branch and Bound (DFBB) performs a depth-first traversal of the search tree. This algorithm keeps two bounds, lb and ub, for each node of the tree which corresponds to a particular assignment. The lower bound at node t, $lb(t)$, is an underestimation of the cost of any complete assignment below t. The upper bound ub is the highest acceptable cost. When

$lb(t) \geq ub$, the subtree rooted at t can be pruned because it contains no solution with cost lower than ub. If a complete assignment is founded with cost lower than ub, this cost becomes the new ub. After exhausting the tree, DFBB returns the current ub, which is the optimum cost. DFBB has an exponential temporal complexity. Since it performs depth-first search, its spatial complexity is linear.

The efficiency of DFBB depends largely on its pruning capacity and the quality of its bounds: the higher lb and the lower ub, the better DFBB performs, since it can perform more pruning, exploring a smaller part of the search tree.

In the last 20 years many efforts have been made to improve (that is, to increase) the lower bound. The simplest lower bound one can think of is the sum of costs of the cost functions which have their variables assigned. These cost functions can be evaluated and their costs added. More sophisticated lower bounds also evaluate the impact of assigned variables on cost functions which contain unassigned variables. This was often implemented using counters (Freuder and Wallace, 1992; Larrosa et al., 1999).

An alternative lower bound was developed in the Russian Doll Search approach (Verfaillie et al., 1996). Assuming a static variable ordering, let us consider when the variables assigned are from x_1 to x_{i-1}. Sub-problem i is defined by variables $x_i, \ldots, x_n$ and cost functions among them. Then, a lower bound can be defined that includes the optimum cost of sub-problem i. The solving method replaces one search by n searches on nested sub-problems, each solving optimally sub-problems $n, n-1, \ldots, 1$. The optimal cost of sub-problem j is stored and reused when solving sub-problem $i < j$.

After the development of soft arc consistency (Cooper and Schiex, 2004; Schiex, 2000), lower bounds have been based on the maintenance of some soft local consistency. These bounds have shown to be superior to previous approaches. Then, we mention the levels AC, FDAC (Larrosa and Schiex, 2003), EDAC (de Givry et al., 2005), VAC (Cooper et al., 2008) and OSAC (Cooper et al., 2007). For a comprehensive summary of all these approaches see (Larrosa and Schiex, 2004) and (Cooper et al., 2010).

2.1.2.2 Complete Inference: Bucket Elimination

Bucket Elimination (BE) (Dechter, 1999, 2003) is a complete inference algorithm for optimally solving COPs. In contrast with search strategies, whose basic operation is assigning a variable, complete inference methods are based on operations on cost functions. As consequence, BE is able to compute all optimal solutions, while search methods usually compute a single solution.

This brief description of BE requires to previously define two operations: the *sum of two cost functions* and *projecting out a variable* from a cost function. The sum of two cost functions is a new cost function defined over the union of variables of the two cost functions; the cost of a value tuple of this sum is the sum of costs of the two cost functions evaluated on that tuple (under the assumption that variables not appearing in the scope of a cost function are ignored). Projecting out a variable from a cost function produces another cost function in which values of that variable have been eliminated, keeping the cost of value tuples. As result of this process, some value tuples may be repeated; among those repeated value tuples, only the one with minimum cost is maintained in the new cost function (all the other tuples are eliminated).

BE requires a static variable ordering $x_1, ..., x_n$. It processes variables from last to first, as follows. Let us consider that it processes variable x_i. First, it computes the bucket of x_i as the set of cost functions that have x_i in their scope (x_i is the last variable of the ordering in that scope). Second, it adds the cost functions of the bucket producing a single cost function. And third, it projects out variable x_i from this single cost function. The resulting cost function, that does not mention x_i, is located in the bucket of the last variable (according to the ordering) that appear in its scope. These three consecutive steps are also called "variable elimination", because they perform the necessary process to compute a problem with one less variable that the original one, keeping the same optimum cost. When all variables have been processed in that way, the resulting cost function has arity zero (it does not mention any variable); it is a single number, which is the optimum cost of the problem.

This phase of BE computes the optimum cost. To recover an optimal assignment, some extra work is needed. BE constructs an optimal solution by assigning variables from the first to the last in the ordering and by reusing the intermediate cost functions built to replace buckets. Variable x_i assigns the value that has the best extension of the current partial solution $x_1, \ldots, x_{i-1}$ with respect to that cost function.

The complexity of BE is exponential, both in time and space, in the induced width of the constraint graph. This parameter basically measures graph cyclicity. Although exponential time complexity is not a surprise (the problem to solve is NP-hard), exponential memory represents a real drawback of this method. It comes from the requirement of storing intermediate cost functions in memory.

2.1.3 Soft Local Consistency

Constraint optimization problems are NP-hard, so an exponential time is needed in the worst case to find an optimal solution. This severely limits the application of existing solving approaches. Several techniques have been developed to speed up the solving of constraint optimization problems. In particular, search can be improved by enforcing soft local consistency (also called propagation or incomplete inference techniques). Soft local consistency techniques allows to discover inconsistent values (this is, values that will never appear in an optimal solution). Since we are interested in optimal solutions, sub-optimal values can be removed from the problem. In practical terms, the effect is that the search tree is reduced and there are fewer nodes to explore, but on the other hand more computational work must be done per node. Globally, the overall effect has been very beneficial in terms of CPU time, specially considering the initial levels of soft arc consistency.

Typical solving techniques in constraint programming involve search in the space of all possible solutions. During this search, constraints may not only be used to calculate costs but they can play an active role helping to discover parts of the search space without optimal solutions. Removing such parts from the problem makes the search shorter.

Generally the exploration of a search tree in a COP performs a repetitive exploration of sub-optimal subtrees. In many cases, the search in different parts of the search tree keeps failing for the same reason. For example, repeated failures might be due to a particularly bad value assignment in one variable, differing only in the assignments of other variables that are irrelevant to the sub-optimality of the subtree. This repeated test of irrelevant variable assignments over and over in the search tree is called *thrashing* behavior (Mackworth, 1991). Because there is typically an exponential number of such irrelevant assignments, thrashing is often one of the most significant factors in the runtime of backtracking.

The idea behind consistency algorithms is that some thrashing behavior can be identified and eliminated by performing some reasoning over the cost functions. As result, a value (or combination of values) is explicitly forbidden because a given subset of its constraint costs becomes unaffordable, making this value unacceptable to participate in any solution.

Applying such consistency techniques on a problem P generates a problem P' presumably easier to solve than P, since it reduces the domain of the variables also modifying its cost functions. Operations are done in such a way that it is assured that P' maintains the optimum

solution and the same cost for the remaining solutions. Because of that, these operations are called *equivalence-preserving transformations* (Cooper et al., 2010).

Recently, some soft arc consistency levels have been proposed (Cooper et al., 2008; de Givry et al., 2005; Larrosa and Schiex, 2003). In the following we introduce some local consistency definitions for COPs, independently of the solving strategy used.

2.1.3.1 AC*

Here we present the simplest level of soft arc consistency. First, we consider the binary case, and later we generalize to the n-ary case.

Consider a binary COP: (i, a) means variable x_i taking value a, $\top$ is the lowest unacceptable cost, C_{ij} is the binary cost function between x_i and x_j, C_i is the unary cost function on x_i values, C_ϕ is a zero-ary cost function that represents a necessary global cost of any complete assignment. As in (Larrosa, 2002; Larrosa and Schiex, 2003), we consider the following definition:

- *Node Consistency**: (i, a) is node consistent* (NC*) if $C_\phi + C_i(a) < \top$; x_i is NC* if all its values are NC* and there is a value $a \in D_i$ s.t. $C_i(a) = 0$; a COP is NC* if every variable is NC*.

- *Arc consistency**: (i, a) is arc consistent (AC) wrt. cost function C_{ij} if there is a value $b \in D_j$ s.t. $C_{ij}(a, b) = 0$; b is called a *support* of a; x_i is AC if all its values are AC wrt. every binary cost function involving x_i; a COP is AC* if every variable is AC and NC*.

The AC* property can be assured on every variable applying two operations until the AC* condition is satisfied: forcing supports to NC* values and deleting values not NC*. The systematic application of these operations does not change the optimum cost and maintains the optimal solution as proved in (Larrosa and Schiex, 2003).

Support can be forced by sending (projecting) costs from binary cost functions to unary cost functions. The projection from the binary cost function C_{ij} to the unary cost function C_i for value *a* is a flow of costs defined as follows.

Let α_a be the minimum cost of value *a* with respect to C_{ij} (namely $\alpha_a = min_{b \in D_j} C_{ij}(a,b)$). The projection consists in adding α_a to $C_i(a)$ (namely, $C_i(a) = C_i(a) + \alpha_a, \forall a \in D_i$) and subtracting α_a from $C_{ij}(a, b)$ (namely, $C_{ij}(a, b) = C_{ij}(a, b) - \alpha_a, \forall b \in D_j, \forall a \in D_i$). This

operation can be seen as if we were assuring that, no matter which value variable x_j takes, x_i has to pay at least α_a when value a is assigned.

In the same way, NC* values are assured projecting from unary cost functions to C_ϕ. The projection from unary costs C_i to C_ϕ consists in decrementing the minimum cost from C_i and increment it in C_ϕ. In this way the minimum cost of all unary constraints (of any variable) can be projected to C_ϕ, producing a necessary cost for any complete assignment. So C_ϕ is calculated as a global lower bound of any solution.

A unary constraint $C_i(a)$ can be seen as the minimum cost that variable x_i has to pay if it chooses value a, no matter which are the values of the other variables. So $C_i(a)$ is a lower bound of value a and C_ϕ is a lower bound of any solution. If $C_i(a) + C_\phi \geq \top$ then a can be removed from the problem, since any assignment containing value a for variable x_i costs at least $\top$, which is an unacceptable cost.

When a value a is removed in x_i supports may be lost in neighbors (if a is a support in some neighbors), so the AC* property needs to be rechecked on every variable that x_i is constrained with. As result of this, new projections may be performed from x_i to neighbors and new inconsistent values may appear on neighbors. In this way, a deleted value in one variable might cause further deletions in other variables. This AC* check must be performed until no further values are deleted.

A simple example of AC^* enforcement is presented in Figure 2.2. Consider two variables x_i and x_j with domain $\{a, b, c\}$ and $\top = 10$ (Figure 2.2(a)). Binary costs with costs different from zero are represented with lines and unary costs are represented in parenthesis (initially all unary costs are 0). First, projections are performed from C_{ij} to C_i for all values in D_i (Figure 2.2(b)). As result, $C_i(a) = 1$, $C_i(b) = 5$, $C_i(c) = 11$ and binary costs in C_{ij} are decremented. With the remaining costs in C_{ij}, projections are performed from C_{ij} to C_j for all values in D_j (Figure 2.2(c)). At this point, every value in D_i and D_j have a support. Then, projections are performed from unary costs C_i and C_j to C_ϕ and C_ϕ is incremented in 1 (Figure 2.2(d)). Value c in variable x_i is found inconsistent because it does not satisfy the condition $C_i(c) + C_\phi < \top$ $(1 + 10 < 10)$ so it is deleted from x_i (Figure 2.2(d)). Observe that the deleted value (i, c) is a support in x_j. Therefore, AC* needs to be rechecked in x_j so projections are performed from C_{ij} to C_j (Figure 2.2(e)). As result, $C_j(c)$ is incremented and value c is deleted in x_j, since $C_j(c) + C_\phi < \top$ $(11 + 1 < 10)$ (Figure 2.2(f)). So the deletion of an inconsistent value in x_i produced a new deletion in x_j.

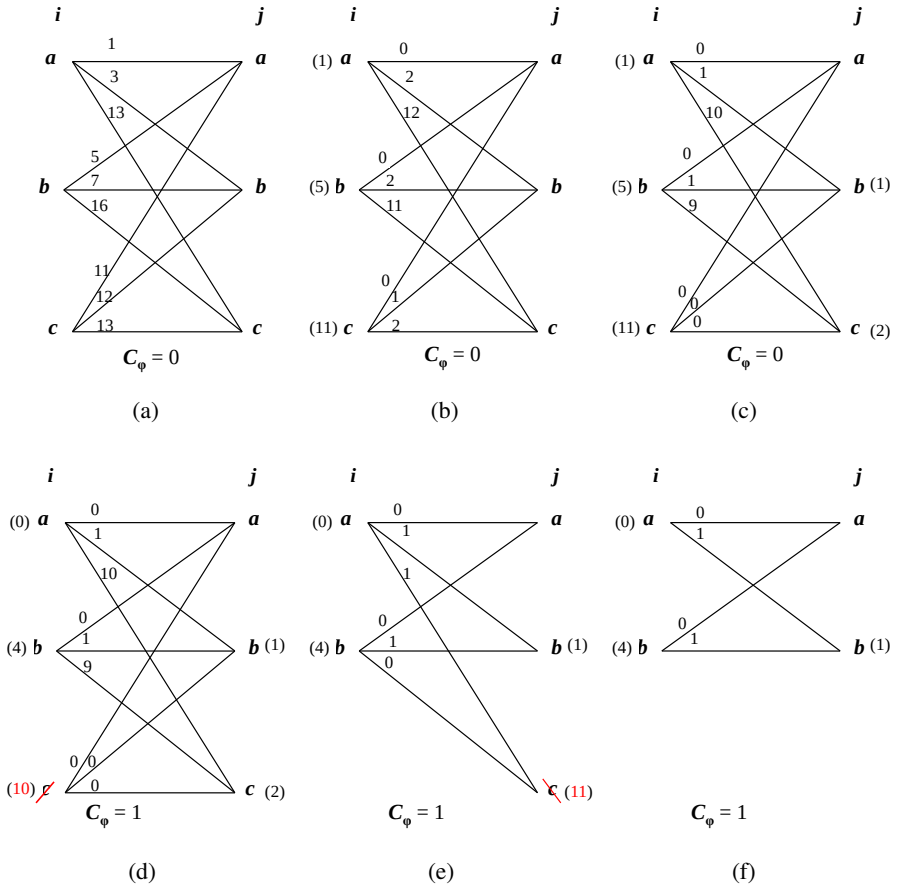


Figure 2.2: AC* example. Lines represent binary costs. Values in parenthesis represent unary costs.

For COPs with cost functions C_S of arity higher than two, where S is the set of variables involved in the constraint, we consider the following definition as in (Lee and Leung, 2009):

- *Generalized Arc Consistency**: (i, a) is generalized arc consistent (GAC) wrt. a non-unary cost function C_S if: there exists values $v_j \in D_j$ for all $x_j \in S$ and $j \neq i$ so that they form a tuple t with $C_S(t) = 0$. $\{v_j\}$ is a simple support of a with respect to C_S. x_i is GAC if all its values are GAC wrt. every cost function involving x_i. A COP is GAC* if every variable is AC and NC*.

Notice that GAC* collapses to AC* for binary constraints.

2.1.3.2 FDAC*

Consider a binary COP: (i, a) means variable x_i taking value a, $\top$ is the lowest unacceptable cost, C_{ij} is the binary cost function between x_i and x_j, C_i is the unary cost function on x_i values, C_ϕ is a zero-ary cost function that represents a necessary global cost of any complete assignment and variables are totally ordered. As in (Larrosa and Schiex, 2003), we consider the following definition:

- *Directional arc consistency**: (i, a) is directional arc consistent (DAC) wrt. cost function C_{ij}, $j > i$, if there is a value $b \in D_j$ s.t. $C_{ij}(a, b) + C_j(b) = 0$; b is called a *full support* of a; x_i is DAC if all its values are DAC wrt. every C_{ij}, $j > i$; a COP is DAC* if every variable is DAC and NC*.
- *Full DAC**: a COP is FDAC* if it is DAC* and AC*.

Full supports can be enforced by first sending costs from a unary constraint C_j to C_{ij} (extension) and then sending the cost from C_{ij} to C_i (projection). Specifically, full supports are enforced in a variable x_i extending costs from unary cost functions of each neighbor x_n, $n > i$, to the binary cost functions and after projecting costs from the binary cost functions to the unary cost functions of x_i.

Extensions from a unary cost $C_i(a)$ to the binary cost C_{ij} consist in decrementing a cost α_a from $C_i(a)$ and increment it in $C_{ij}(a, b)$ $\forall b \in D_j$. In the case of full supports, the α_a cost extended is the the maximum cost that can be projected in the next step into x_j (Larrosa and Schiex, 2003). Projections are performed as explained in the previous Section: a projection into value b of x_j consist in decrementing the minimum cost α_b from binary $C_{ij}(a, b)$ $\forall a \in D_i$ and incrementing this minimum cost in $C_j(b)$.

In FDAC* costs can be passed from one variable to another in the constraint graph. This allows to aggregate costs in one variable originally coming from other variables in the problem, which may lead to more pruning opportunities. For example, consider variables x_i, x_j and x_k in Figure 2.3(a) with domain $\{a, b\}$ where $k < j < i$ and $\top = 5$. Binary costs different from zero are represented with lines and unary costs are represented between parenthesis. To enforce FDAC*, first unary costs C_i are extended to binary costs C_{ij} (Figure 2.3(b)). The cost extended is the minimum cost that can be projected to x_j after extension: for $C_i(a)$ this cost is 4 and for $C_i(b)$ is 0. Then, a projection is done from C_{ij} to C_j and as result $C_j(a)$ is incremented in 4 (Figure 2.3(c)). Now agent j has a full support for every value. To assure full supports

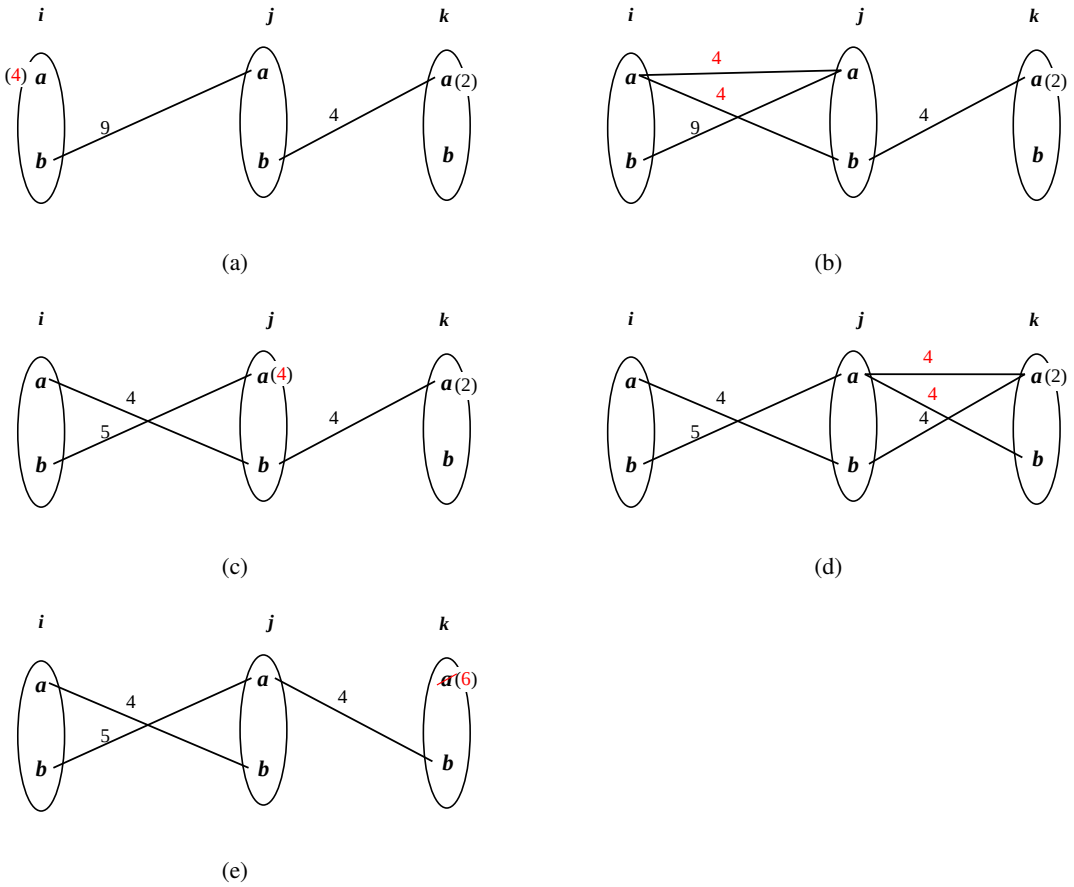


Figure 2.3: FDAC* example. Lines represent binary costs different from zero. Values in parenthesis represent unary costs. $\top = 5$. After enforcing FDAC*, value a of variable x_k can be removed.

in agent k, an extension must be done from unary costs C_j to binary costs C_{jk} (Figure 2.3(d)) and afterwards a projection from C_{jk} to C_k (Figure 2.3(e)). As result, $C_k(a)$ is incremented in 4 and value (k, a) is detected inconsistent since $C_k(a) + C_\phi > \top$ $(6 + 0 > 5)$ so it can be deleted from the problem. Observe that a cost of 4, initially on unary cost $C_i(a)$, has passed from agent i to agent k performing cost extensions and projections, finally reaching $C_k(a)$. By this, it was possible to delete a value that could not be deleted enforcing AC*.

2.1.3.3 EDAC*

Consider a binary COP: (i, a) means variable x_i taking value a, $\top$ is the lowest unacceptable cost, C_{ij} is the binary cost function between x_i and x_j, C_i is the unary cost function on x_i

values, C_ϕ is a zero-ary cost function that represents a necessary global cost of any complete assignment and variables are totally ordered. As in (de Givry et al., 2005), we consider the following definitions:

- *Existential Arc Consistency**: Variable x_i is existential arc consistent if there is at least one value $a \in D_i$ such that $C_i(a) = 0$ and it has a full support in every constraint C_{ij} . A problem is existential arc consistent (EAC*) if every variable is NC* and EAC*.

- *Existential Directional Arc Consistency**: A problem is EDAC* if it is FDAC* and EAC*.

EDAC* requires that every value is fully supported in one direction and simply supported in the other direction (to satisfy FDAC*). Additionally, at least one value per variable must be fully supported in both directions (to satisfy EAC*). This special value is called the *fully supported* value.

EDAC* can be enforced by performing projections and extensions following the established order to satisfy FDAC* and also performing projections and extensions against the established order if as result an increment in C_ϕ occurs. So in EDAC* full supports can be enforced in both directions if their enforcement produces an increment in C_ϕ, observe that this is an increment that would not be achieved with FDAC*.

NC* has become standard in the soft local consistency community, to the point that higher local consistencies using it are named without asterisk (Cooper et al., 2010). Following this trend, in the rest of this thesis we refer to AC* , FDAC* , EDAC* as AC, FDAC, EDAC respectively.

2.1.4 Soft Global Constraints

Global constrains have been essential for the advancement of constraint reasoning. Global constraints are classes of constraints that naturally appear in many practical scenarios involving several problem variables. They capture a relation between a non-fixed number of variables and this relation has associated a clear semantic. For example consider the set of variables T , the well-known *alldifferent(T)* global constraint means that all the variables in the set T must assign a different value (independently of the cardinality of T). Constraints with different arity can be defined by the same class. For instance, *alldifferent*(x_1, x_2, x_3) and *alldifferent*(x_1, x_4, x_5, x_6) are two instances of the *alldifferent* global constraint class.

For COPs, soft global constraints have been defined. In soft global constraints the cost of value assignments is evaluated using a violation measure μ. This violation measure may vary for each soft global constraint (we assume that a soft constraint instance can be expressed and evaluated by a cost function, so we use the terms soft global constraints and global cost functions interchangeably). For example, consider the following soft global constraints:

- *soft-alldifferent*(T). This global constraint expresses that all variable values in T should be different. Costs are defined by violation measures μ_{var} and μ_{dec} (Petit et al., 2001): μ_{var} is the number of variables in T that have to change their values to satisfy that all values are different, while μ_{dec} is the number of pairs of variables with the same value.

- *soft-at-most[k,v]*(T). This global constraint expresses that at most k variables in T should take value v. Costs are defined by violation measure μ_{var}, which is the number of variables in T that have to change to satisfy this condition.

2.1.4.1 Properties

A soft global constraint C with violation measure μ is *contractible* iff μ is a non-decreasing function (Maher, 2009a) [1]. The intuition behind is as follows: C with μ is contractible when $\mu(a, b, c) \leq \mu(a, b, c, d) \leq \mu(a, b, c, d, e)$...., so shortening by the right the sequence of variables on which C is defined gives a valid lower bound to the cost of C.

A soft global constraint C with violation measure μ admits a *binary decomposition without extra variables* iff for any instance $C(x_1, \ldots, x_p)$ of C, there exists a set S of binary soft constraints involving only variables $x_1, \ldots, x_p$ such that for any value tuple t on $x_1, \ldots, x_p$, $\sum_{C_{x_i,x_j} \in S} C_{x_i,x_j}(t[x_i, x_j]) = \mu(t)$. We also say that C is semantically decomposable in S. For example, the *soft-alldifferent* constraint is binary decomposable with violation measure μ_{dec} (Petit et al., 2001). It is easy to see that this constraint can be decomposed in a clique of binary constraints using this violation measure, as shown in the example of Figure 2.4.

Not every global constraint can be decomposed into an equivalent set of binary constraints. For example, the *soft-alldifferent* constraint is not binary decomposable with violation measure μ_{var}. Consider the example of Figure 2.5 where a *soft-alldifferent* constraint defined over variables x_1, x_2, x_3 with violation measure μ_{var} is represented (Figure 2.5, up) along with its binary decomposition with violation measure μ_{var} ((Figure 2.5, down). Observe that the tuple

[1] Function f on a sequence is non-decreasing if $f(a) \leq f(b)$, for every sequence a and b such that a is a prefix of b (Maher, 2009a).

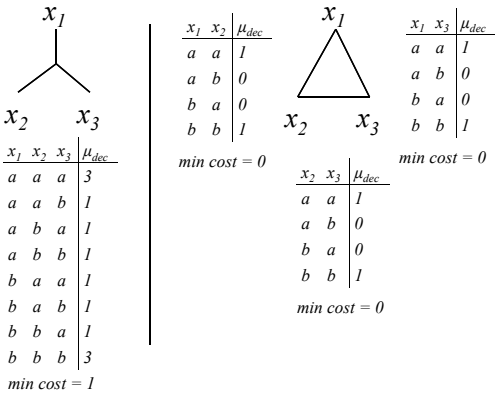


Figure 2.4: (Left) *soft-alldifferent*(x_1, x_2, x_3) with μ_{dec} violation measure; (right) its binary decomposition

$(x_1 = a, x_2 = a, x_3 = a)$ has a different cost in the global formulation (up) and in the binary decomposition (down). Hence, this soft constraint is not binary decomposable with violation measure μ_{var}.

$D_{x_1} = \{a\}$ $D_{x_2} = \{a, b\}$ $D_{x_3} = \{a, b\}$

x_1	x_2	x_3	μ_{var}
a	a	a	2
a	a	b	1
a	b	a	1
a	b	b	1

soft-alldifferent$_{x_1,x_2,x3}(a, a, a) = 2$

x_1	x_2	μ_{var}
a	a	1
a	b	0

x_1	x_3	μ_{var}
a	a	1
a	b	0

x_2	x_3	μ_{var}
a	a	1
a	b	0
b	a	0
b	b	1

soft-alldifferent$_{x_1,x_2}(a, a)$+ *soft-alldifferent*$_{x_1,x_3}(a, a)$+ *soft-alldifferent*$_{x_2,x_3}(a, a) = 3$

Figure 2.5: (*up*) *soft-alldifferent* global constraint with μ_{var} violation measure; (*down*) a decomposition in binary constraints. However, *soft-alldifferent* is not binary decomposable with μ_{var}

Finally, a soft global constraint admits a *decomposition with extra variables* if it can be decomposed in a finite number of fixed-arity constraints using extra variables (this is, variables which are not originally present in the problem but are introduced to allow the decomposition).

Summarizing, considering the soft global constraints *soft-alldifferent* and *soft-at-most[k,v]*:

- *soft-alldifferent*(T) with μ_{dec}. It is contractible and binary decomposable
- *soft-alldifferent*(T) with μ_{var} It is contractible but not binary decomposable.
- *soft-at-most[k,v]*(T) with μ_{var}. It is contractible but not binary decomposable. It allows decomposition with extra variables in a finite amount of constraints of arity 3.

2.1.4.2 Soft Arc Consistency

It is known that when applying soft arc consistency to some global constraints, the quality of the bounds obtained is better than when working with an equivalent decomposition. For example, consider the global constraint *soft-alldifferent(*x_1, x_2, x_3*)* with the domain set $\{a, b\}$ for every variable and violation measure μ_{dec} (Figure 2.4, left) and its binary decomposition (Figure 2.4, right). If GAC is applied on the global formulation it can be inferred a lower bound of 1 for the optimal solution. Since there are three variables and only two domain values, any ternary tuple (with a combination of x_1, x_2, x_3 values) will cost at least 1 (Figure 2.4, left). However in its binary decomposition we can only infer a lower bound of 0, if looking independently every binary tuple (Figure 2.4, right).

Enforcing GAC on soft global constraints can be expensive using generic propagators because all tuples of the domain must be revised. In the worst case, this is exponential in the number of variables. Some efficient techniques have been proposed for some global constraints that exploit constraint semantics, reaching the consistency level with lower complexity (usually polynomial) than with generic propagators (Lee and Leung, 2009).

2.2 Distributed Constraint Optimization Problems

Moving into a distributed context, a *Distributed Constraint Optimization Problem* (*DCOP*), is a *COP* where variables, domains and cost functions are distributed among automated agents. Formally, a *DCOP* is a 5-tuple (*X, D, C, A,* α), where *X, D, C* define a *COP* and:

- *A* = $\{1, \ldots, p\}$ is a set of *p* agents
- α: X $\rightarrow$ *A* is a function that maps each variable to one agent

We make the usual assumption that each agent owns exactly one variable, so we use the terms agents and variables interchangeably (for notation, we connect agents and variables by

subindexes, agent i owns variable x_i). We also assume that a cost function f defined over several variables is known by every agent that owns a variable of $var(f)$ (Yokoo et al., 1998). In a significant part of this thesis we assume unary and binary cost functions only, since most of the DCOP solving algorithms make this assumption. In this case, a cost function is denoted as *C* with the indexes of the variables involved, so C_{ij} is the binary cost function between agent *i* and *j*, and C_i is a unary cost function over agent *i*.

Agents communicate through messages in order to coordinate towards the optimal solution. It is assumed that these messages can have a finite delay but are never lost. For any pair of agents, messages are delivered in the same order that they were sent.

2.2.1 DCOP Representations

A DCOP –like a COP– can be represented by a *constraint graph*, where nodes in the graph correspond to variables and edges connect pairs of variables appearing in the same cost function.

It can also be represented as a *depth-first search* (DFS) *pseudo-tree* arrangement with the same nodes and edges as the constraint graph and the following conditions:

- There is a subset of edges, called *tree-edges*, that form a rooted tree. The remaining edges are called *back-edges*.
- Variables involved in the same cost function appear in the same branch of that tree.

For example, Figure 2.6(a) represents a constraint graph and Figure 2.6(b) a DFS pseudo-tree of this constraint graph (back-edges are represented with dotted lines).

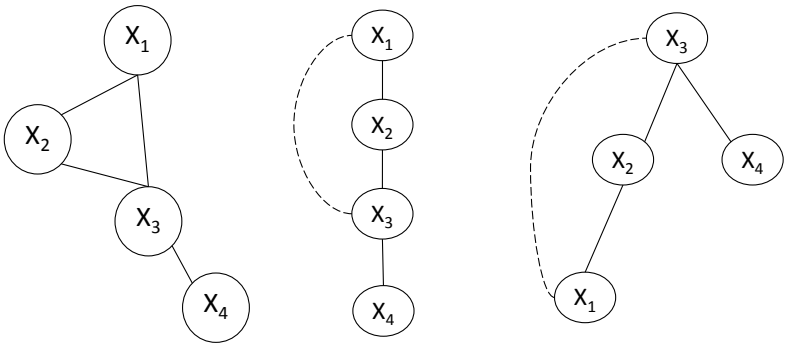
Tree edges connect parent-child nodes in the pseudo-tree: every node in the pseudo-tree has a single parent (except the *root* agent) and may have one or multiple children. Back-edges connect a node with its pseudo-parents (ancestors in the pseudo-tree) and pseudo-children (descendants in the pseudo-tree). In Figure 2.6(b), x_2 is a child of x_1; and x_3 is a pseudo-child of x_1. We assume that agents that share a constraint are neighbors, so x_2 and x_3 are neighbors of x_1 and vice versa.

There are many possible DFS pseudo-trees for a given constraint graph (for example see Figure 2.6(b) and (c)). They can be constructed starting at an arbitrary agent (the *root*) and making a traversal of the graph in a depth-first search order until all vertices are visited. Every edge visited will be an edge of the DFS pseudo-tree and every edge not visited will be a back-edge. DFS pseudo-trees can be constructed using distributed algorithms (Petcu, 2007) where a

token is passed among agents following a depth-first traversal containing the identifiers of the visited agents. From this token, the parent, pseudo-parents, children and pseudo-children can be identified.

In this thesis we generate DFS pseudo-trees following a most-connected heuristic (starting with the most connected agent in the constraint graph and performing the DFS tree traversal selecting first the most connected neighbors).

DFS pseudo-trees are useful representation for some solving algorithm since every branch of the pseudo-tree represents an independent sub-problem of the DCOP instance. Several distributed algorithms exploit this DFS pseudo-tree arrangement of their variables since it allows agents positioned in different branches to perform search in parallel. In the following, we may refer to a DFS pseudo-tree simply as a pseudo-tree, assuming a DFS order among its nodes (variables).



(a) Constraint Graph (b) DFS Pseudo-Tree (c) Another DFS Pseudo-Tree

Figure 2.6: DCOP Representations

2.2.2 Solving Algorithms

In this Section we describe some standard algorithms for DCOP solving. Several solving algorithms has been developed to solve DCOPs, following different approaches: complete or incomplete algorithms, partially centralized or fully decentralized algorithms, inference or search algorithms with different levels of synchronous or asynchronous behaviour.

In the following, we explain some fully decentralized and complete algorithms for DCOP solving. Particularly, we describe with special detail the algorithm BnB-ADOPT since it is the

solving algorithm that we use in the rest of this thesis. Therefore, we encourage the reader to take a closer look at its description.

2.2.2.1 DPOP

DPOP is the acronym for Dynamic Programming Optimization Protocol. It was presented first in (Petcu and Faltings, 2005) and it appears in full detail in (Petcu, 2007).

DPOP is a distributed inference algorithm (it can be seen as the distributed version of the Bucket Elimination algorithm (Dechter, 1999)). It does not exchange individual values among agents, as search algorithms do. DPOP exchanges complete cost functions among agents. It uses the following two operations defined on cost functions (these are the same two operations required for the Bucket Elimination algorithm, described in Section 2.1.2.2; we denote cost functions by f_V, where V is the set of variables on which the function f_V is defined):

1. *Projecting out* a variable $x \in V$ from f_V is a new function with scope $V - \{x\}$ defined as projecting f_V on $V - \{x\}$, that is, first removing the values of x and then removing duplicate tuples, while keeping the minimum cost of the original tuples in f_V.

2. *Summing* two functions f_V and g_W is a new function $f + g$ with scope $V \cup W$ and $\forall t \in \prod_{x_i \in V} D_i, \forall t' \in \prod_{x_j \in W} D_j$ such that they can concatenate (the concatenation of t and t', written as $t \cdot t'$ is a new tuple formed by the values appearing in t and t', requiring that on common variables they should have the same values), $(f + g)_{V \cup W}(t \cdot t') = f_V(t) + g_W(t')$.

DPOP works on a DFS-tree arrangement of the constraint graph and performs three phases in sequence:

1. DFS phase. (DFS-tree generation). An agent is selected as root (for instance, by a leader election process). From this agent starts a distributed DFS traversal of the constraint graph. At the end of this traversal, each agent labels its neighbors as parent, pseudo-parents, children or pseudo-children.

2. UTIL phase. (UTIL message exchange, bottom-up propagation). Each agent (starting from leaves) sends a UTIL message to its parent, that contain an aggregated cost function computed adding received UTIL messages from its children with its own cost functions with parent and pseudo-parents. The sent cost function does not contain the agent's

variable, which is projected out. Observe that UTIL messages have exponential size, since they exchange complete cost functions.

3. VALUE phase. (VALUE message exchange, top-down propagation). Each agent determines its optimal value using the cost function computed in phase 2 and the VALUE message received from its parent. Then, it informs its children using VALUE messages. The agent at the root starts this phase.

We identify each phase as DPOP(*phase*). DPOP(*util*) is a one-pass process from leaves to root: it is the phase responsible for using exponentially large messages. DPOP(*dfs*) is a preprocess to compute the DFS-tree arrangement, while DPOP(*value*) is a one-pass process from root to leaves, with the best assignments for variables in early levels of the DFS-tree. DPOP(*dfs*) and DPOP(*value*) both require a polynomial number of linear size messages.

Originally DPOP considers that agents have utilities associated with value tuples, so an optimal solution is the tuple that maximizes the overall utility. Here, we consider that agents have costs associated with value tuples, so an optimal solution is the tuple that minimizes the overall cost.

It is worth mentioning that, when the DFS pseudo-tree has no back edges (that is, it is a real tree), there is a version of this algorithm (called DTREE in (Petcu and Faltings, 2005)) which uses UTIL messages of linear size.

2.2.2.2 SBB

Synchronous Branch and Bound (SBB) (Hirayama and Yokoo, 1997) is a distributed complete search algorithm for DCOPs. It performs a branch-and-bound search in a synchronous way. In this algorithm only one agent is active at every moment.

SBB uses a total ordering among agents. Agents exchange messages following this ordering to send a *path*, which contains the current partial assignment of variables and its cost. This *path* is exchanged among agents and extended with new value assignments exploring the search space until the optimal solution is found. Also, an upper bound UB is passed among agents containing the cost of the best solution found so far.

SBB executes in the following way:

- The first agent in the ordering starts the search sending a *path* that constains only its value assignment. This *path* is sent to the second agent in the ordering.

- When receiving a *path* from a previous agent in the agent ordering, the receiver agent evaluates the cost of assigning its first value in the value ordering. If this cost is lower than UB the receiver adds its own assignment to the *path* and sends it to the next agent. Otherwise, it continues trying next domain values until the cost is lower than UB. If values are exhausted, agent i it backtracks to the previous agent by returning the *path*.

- When receiving a *path* from the previous agent in the agent ordering, an agent changes its assignment to the next value in its value ordering, calculates the cost of the new *path*, and sends it to the next agent if its cost is less than the UB. Otherwise, it continues to try next values. Another backtrack takes place if values are exhausted.

2.2.2.3 AFB

Asynchronous Forward-Bounding (AFB) (Gershman et al., 2009) is a complete and partially asynchronous algorithm for DCOP solving also based on a branch-and-bound search strategy. In the AFB algorithm agents are totally ordered and messages are exchanged between agents containing a current partial assignment, called CPA, and a unique timestamp so the recency of CPAs can be compared. Agents have an upper bound UB which is the cost of the best solution found so far. Also, agents maintains lower bound estimations for every value assignment considering a particular CPA. This lower bound estimations are calculated according to information they receive from lower priority agents.

Agents try to extend the CPA with new value assignments until a complete solution is found. If an agent succeeds in adding a new value assignment in the CPA, it sends copies of the updated CPA to all unassigned agents, requesting to compute lower bound estimations for that CPA. These estimations are computed and are sent back to the sender agent. In this way, it is posible to discover that the lower bound of the current CPA is higher than the UB and perform an earlier backtrack. In this algorithm more than one CPA may exist at any time so the timestamps are introduced to discard obsolete CPA corresponding to abandoned partial solutions.

Execution of AFB is as follows:

- The first agent starts adding its first value assignment to the CPA and sending a CPA message to the next agent in the ordering. It also sends a FB_CPA message (forward bounding request) to all agents whose assignments are not yet in the CPA.

- When an agent receives a CPA message it makes sure that it is not obsolete comparing timestamps. If the message is not discarded, the agent tries to assign a new value assignment to the CPA. The cost of the new value assignment in the CPA plus its lower bound estimations can not exceed UB. If no such value is found the agent pass the CPA to the previous agent (backtrack). Otherwise, the agent adds the assignment to the CPA, sends a CPA message to the next agent and a FB_CPA message to unassigned agents. If a complete assignment has been reached (the agent is the last agent in the ordering) a NEW_SOLUTION message is broadcasted to all agents with the new UB. This is done for every new complete solution until no further improvements to the UB can be made. Then, the last agent performs backtrack sending the CPA to its previous agent.

- When an agent receives a FB_CPA message it makes sure that it is not obsolete comparing timestamps. If the message is not discarded, the agent computes an estimate lower bound for the received CPA which is calculated considering the cost of its lowest cost assignment considering this CPA. This estimation is sent back to the sender agent with an FB_ESTIMATE message.

- When an agent receives a FB_ESTIMATE message it makes sure that it is not obsolete comparing timestamps. If the message is not discarded, it saves the estimates and checks if the aggregation of lower bound estimations received causes the current partial assignment to exceed UB. In such a case, the agent tries to change its value assignment with the next value or backtracks in case a valid assignment cannot be found.

2.2.2.4 ADOPT

ADOPT (Asynchronous Distributed Constraint Optimization with Quality Guarantees) (Modi et al., 2005) is a reference algorithm for asynchronous distributed search. It can find the optimal solution for a DCOP, or a solution within a user-specified distance from the optimal, using local communication and polynomial space at each agent.

Unlike SBB and AFB, ADOPT uses a pseudo-tree arragenment of its variables that allow agents to work on independent sub-problems (branches of the pseudo-tree) concurrently. According to ADOPT authors, a synchronous method were agents remain idle waiting for a particular message to arrive is unacceptable because it is wasting time when agents coud be doing potencially useful work.

ADOPT traverses the search space in a best-first search order. Each agent change its value assignment whenever it detects there is a possibility that some other value may be better than the one currently assigned. To make such decisions, agents use lower and upper bounds calculated and delivered by children in the pseudo-tree. These lower and upper bounds are initially 0 and ∞ and are refined iteratively as search progresses. Agents change value whenever the lower bound of a domain value is lower than the lower bound of its current value assignment. In this way, agents perform a best-first search strategy where partial solutions may be abandoned to change to the most promising value considering the current information. This allows a high degree of parallelism because agents do not wait to gather all the information of global bounds needed to discard the current value as sub-optimal. However, since partial solutions are abandoned before sub-optimality is proved they may need to be revisited.

ADOPT agents use four types of messages: VALUE, COST, THRESHOLD and TERMINATE. Every parent agent sends a VALUE message to its children and pseudo-children to inform of value assignments. As response, every child sends to its parent a COST message informing the lower and upper bound the child is able to calculate considering the parent current assignment, the assignment of ancestors and also the lower/upper bounds that the child has received from its own children. These bounds can be iteratively refined with further COST messages. THRESHOLD messages allow agents to reconstruct previously abandoned solutions in an efficient way, which is a frequent action due to ADOPT search strategy since agents only store information from the current partial solution. This technique requires only polynomial space in the worst case, which is much better that the exponential space that would be required to simply memorize all partial solutions in case they need to be revisited.

Finally, the algorithm provides a built-in mechanism for termination. For this it uses bound intervals (a lower and upper bound of the optimal solution). When the size of this bound interval shrinks to zero (the lower bound is equal to the upper bound), the cost of the optimal solution has been determined. In the same way, when the bound interval shrinks to a user-specified size, agents can terminate guaranteeing that the cost of the solution found is within the given distance of the optimal solution. This means that agents can find an approximate solution (sub-optimal solution) faster than the optimal one but they provide a theoretical guarantee on the global solution quality.

2.2.2.5 BnB-ADOPT

Branch and Bound ADOPT (BnB-ADOPT) (Yeoh et al., 2010) is a distributed asynchronous algorithm that optimally solves DCOPs using a depth-first branch-and-bound search strategy. It is closely related to ADOPT (Modi et al., 2005), maintaining most of its data structures and communication framework. BnB-ADOPT starts constructing a DFS pseudo-tree arrangement of its agents. After this, each agent knows its parent, pseudo-parents, children and pseudo-children.

Data Structure:

During execution an agent i maintains: its current value assignment d; a timestamp for every value assignment (so their recency can be compared); a current context X_i, which is a set of value assignment that represent its knowledge about the current value assignment of its ancestors; for every value $d \in D_i$ and context X_i, a lower and upper bound $\boldsymbol{LB}_i(d)$ and $\boldsymbol{UB}_i(d)$, which are bounds on the optimal cost $OPT_i(d)$ given that x_i takes on the value d and its ancestors take on their respective values in X_i; and the lower and upper bounds $\boldsymbol{LB}_i$ and $\boldsymbol{UB}_i$, which are bounds on the optimal cost OPT_i given that its ancestors take on their respective values in X_i. Costs and bounds are calculated in the following way:

$$\boldsymbol{OPT}_i(d) = \delta_i(d) + \textstyle\sum_{x_c \in C_i} \boldsymbol{OPT}_c$$
$$\boldsymbol{OPT}_i = \min_{d \in D_i} \boldsymbol{OPT}_i(d)$$

$$\delta_i(d) = \textstyle\sum_{(x_j, d_j) \in X_i} C_{ij}(d, d_j)$$

$$\boldsymbol{LB}_i(d) = \delta_i(d) + \textstyle\sum_{x_c \in C_i} lb_{i,c}(d)$$
$$\boldsymbol{UB}_i(d) = \delta_i(d) + \textstyle\sum_{x_c \in C_i} ub_{i,c}(d)$$

$$\boldsymbol{LB}_i = \min_{d \in D_i} \{\boldsymbol{LB}_i(d)\}$$
$$\boldsymbol{UB}_i = \min_{d \in D_i} \{\boldsymbol{UB}_i(d)\}$$

where C_i is the set of children of agent i, $\delta_i(d)$ is the sum of costs of all cost functions between i and its ancestors given that i assigns value d and ancestors assign their respective values in X_i. Tables $lb_{i,c}(d)$ and $ub_{i,c}(d)$ store the agent i assumption on the bounds LB_c and UB_c of

children c, for all values $d \in D_i$ and current context $X_i \cup (x_i, d)$. Due to memory limitations, agent i can only store lower and upper bounds for *one* context. So agents reinitializes its lower/upper bounds each time there is a context change.

To prune values during search, agents use a threshold value TH, initially ∞. Thresholds represent an estimated upper bound for the current context, which may in some cases be more strong than upper bound UB. In the root agent TH_{root} remains always ∞. Threshold values are sent from parent to children. A threshold th sent from agent i with value d to child c is calculated as:

$$th = min(TH_i, UB_i) - \delta_i(d) - \sum_{ch \in C_i, ch \neq c} lb_{i,ch}(d)$$

th is calculated on parent i based on information obtained exploring previous values (UB_i) and information from exploring the current value d ($\delta_i(d)$, $lb_{i,ch}(d)$). When this information arrives to child c its threshold TH_c, initially infinity, is updated with the th sent from parent i.

Communication:

Some communication is needed in BnB-ADOPT to calculate the global costs of agents assignments and to coordinate search towards the optimal solution. BnB-ADOPT agents use three types of messages: VALUE, COST and TERMINATE, defined as follows:

- VALUE($i; j; val; th$): agent i informs child or pseudo-child j that it takes value val with threshold th;
- COST($k; j; context; lb; ub$): agent k informs parent j that with $context$ its bounds LB_k and UB_k are lb, ub;
- TERMINATE($i; j$): agent i informs child j that i terminates.

As mentioned, BnB-ADOPT associates a timestamp to each value assignment (either travelling in VALUE or COST messages). This permits VALUE and COST messages to update the context of the receiver agent, if their values are more recent. Every time there is a context change in agent i as result of a new value assignment on an ancestor p, the bounds of children $lb_{i,c}$, $ub_{i,c}$ might be reinitialized. This is the case if c is constrained with the ancestor p that changed its value.

Upon reception of a VALUE message, value val is copied in the receiver context if its timestamp is more recent, and threshold th is updated in the receiver if it the message comes from the parent.

Upon reception of a COST message from child c, the more recent values in the $context$ of the COST message are copied in the receiver context. After this, if the receiver context is compatible with the COST message $context$, then the agent updates its lower and upper bounds $lb_{i,c}(d)$ and $ub_{i,c}(d)$ with the lower and upper bounds in the COST message, respectively. Otherwise, the bounds of the COST message are discarded. Contexts are *compatible* iff they agree on common agent-value pairs.

Execution:

The goal of every agent i in BnB-ADOPT is to explore the search space and ultimately chooses the value that minimizes LB_i. BnB-ADOPT changes its value only when it is able to determine that the optimal solution for that value is provably no better than the best solution found so far for its current context. In other words, when $\boldsymbol{LB}_i(d) \geq \boldsymbol{UB}_i$ for current value d. Therefore, it performes a branch-and-bound search. Often, agents also prune values using thresholds sent from their parents, so more precisely agent i changes its value d when $\boldsymbol{LB}_i(d) \geq \min\{\boldsymbol{TH}_i, \boldsymbol{UB}_i\}$.

As search progresses, agent i explores values from its domain and calculates $LB_i(v)$ and $UB_i(v)$ for every value $v \in D_i$. When all values has been explored or pruned, $LB_i = UB_i$, which means that i reaches the optimal solution for context X_i. When $LB_i = UB_i$ is reached in the $root$ agent then the optimal problem solution of the problem has been found and $root$ sends TERMINATE message to children. For a non $root$ agent, the optimal problem solution is found when $LB_i = UB_i$ and a TERMINATE message has been received from its parent.

A BnB-ADOPT agent executes the following loop. It reads and processes all incoming messages. After completely processing the message queue, it changes value if $LB_i(d) \geq \min\{\boldsymbol{TH}_i, \boldsymbol{UB}_i\}$ (being d its current value). After this, the agent sends the following messages: a VALUE message to every child, a VALUE message to every pseudo-child and a COST message to its parent. This process repeats until the root agent r reaches the termination condition $\boldsymbol{LB}_r = \boldsymbol{UB}_r$, which means that it has found the optimal cost. It then sends a TERMINATE message to each of its children and terminate. After agent i receives a TERMINATE message, i sends a TERMINATE message to its children and terminates when $LB_i = UB_i$.

In the following chapters, we assume that the reader has some familiarity with BnB-ADOPT (for a deeper description, see the original source (Yeoh et al., 2010)).

In this thesis we have studied, implemented and used the BnB-ADOPT algorithm as a base search algorithm because of the following reasons. Following a branch-and-bround strategy, BnB-ADOPT has proved to be more efficient than ADOPT and as efficient as NCBB (Yeoh et al., 2010). It is memory bounded since it uses polynomial memory –unlike DPOP, which requires an exponential size in messages– and restricts communication to neighbors –unlike AFB, which broadcast new solutions to every agent in the problem. These are desirable properties to maintain, for example consider a sensor network application where sensors have limited memory capacity or limited communication radius. BnB-ADOPT is also an asynchronous algorithm –unlike SBB– which can be useful in distributed problems because agents perform computation and exchange information in parallel instead of waiting idle for a particular message to arrive. Another advantage of asynchronous algorithms is that they are more robust to failures than synchronous ones. Finally, BnB-ADOPT offers quality guarantees and allows to end the search process at a user-specified distance from the optimum cost when time is limited.

2.2.3 Experimental Evaluation

In this thesis we perform experimental evaluations of distributed solving algorithms in a discrete event simulator. The simulator contains a list of all the agents involved in a particular instance. Every agent has associated a message queue containing the messages sent to this agent. The simulator performs several iterations, called cycles, until the optimal solution is found. One cycle consist in every agent reading all incoming messages from its message queue, performing local computation, and if needed sending messages to other agents.

Performance is evaluated in terms of the communication cost and computational effort required to solve a problem. As a measure of the communication effort needed to solve a problem we use the total number of messages exchanged during its resolution. We assume the usual case where the communication time is higher than the computation time. Then the total elapsed time is dominated by the communication time. In this case, reducing the number of messages is very desirable, also the agents need to process less information.

As a measure of the computational effort needed to solve a problem we use the number of non-concurrent constraint checks (NCCCs) (Meisels et al., 2002). To calculate the non-concurrent constraint checks every agent has a counter that is incremented every time a constraint is evaluated. This counter is sent in every message. When a message is received the

counter of the receiver agent is updated with the higher value between its own counter and the counter of the received message. When the algorithm terminates the number of non-concurrent constraint checks is selected as the highest value among all agent counters. This value can be seen as the longest sequent of constraint checks performed non-concurrently.

The total number of messages exchanged and NCCCs are standard measures widely used in the DCOP community. As a complementary measure, we also provide in experimental evaluations the number of cycles performed by the simulator.

Generally we perform evaluation using the following benchmarks:

- **Random DCOPs**. They are characterized by $\langle n, d, p_1 \rangle$, where n is the number of variables, d is the domain size of variables and p_1 is the network connectivity defined as the ratio of existing cost functions. For example, binary instances contain $p_1 * n(n-1)/2$ binary cost functions, while ternary instances contain $p_1 * n(n-1)(n-2)/6$ ternary cost functions. Constrained variables are selected randomly until the specified network connectivity is reached. Costs are selected randomly from different sets. Random generation assures connected problems, so all agents of the problem belong to the same constraint graph.

- **Meeting Scheduling Instances**. Variables represent meetings, domains represent time slots assigned for meetings, and there are cost functions between meetings that share participants (Maheswaran et al., 2004). These instances are obtained from a public DCOP repository (Yin, 2008) with four hierarchical scenarios among participants (cases A, B, C and D).

- **Sensor Network Instances**. Variables represent areas that need to be observed, domains represent time slots and there are cost functions between adjacent areas (Maheswaran et al., 2004). These instances are obtained from a public DCOP repository (Yin, 2008), with four available topologies (cases A, B, C and D).

Meeting scheduling and sensor network instances are structured problems since they are generated considering specific rules in how variables are constrained (Maheswaran et al., 2004). On the other hand, random DCOPs are non-structured instances since their variables are constrained randomly and costs are selected randomly from a uniform cost distribution.

3

Distributed Search

During this thesis we have studied, implemented and experimented with ADOPT and BnB-ADOPT algorithms. As result of this work, we devised ways to improved them to a large extent. In this Chapter we present our contributions to distributed search:

- We propose a new version of BnB-ADOPT which avoids sending redundant messages.
- We address the issue of handling efficiently n-ary constraints in BnB-ADOPT.
- We present a new algorithm called ADOPT(k) that generalizes ADOPT and BnB-ADOPT search strategy.

3.1 BnB-ADOPT$^+$: A New Version of BnB-ADOPT

In the distributed context, agents performing complete search may send an exponential number of messages (consider that each agent sends a message each time it takes a new value and, in the worst case, the optimal solution is the rightmost leaf of the search tree). We realized that ADOPT (Modi et al., 2005) and –to a lesser extent– BnB-ADOPT (Yeoh et al., 2010) exchange a large number of messages. Often, this is a major drawback for their practical applicability. Every time an agent in BnB-ADOPT processes completely its message queue, it checks if it must change its value and sends a VALUE message to each children and pseudo-children and a COST message to its parent. Even if the agent maintains its value assignment, or if the lower and upper bounds of the current partial solution remain the same, the agent invariably sends VALUE messages to children and pseudo-children and a COST message to its parent. This strategy –sending information that in many cases has already been sent– is certainly safe. However, it could be seen as "sending more than needed" and offers room for improving the communication protocol.

Aiming at decreasing the number of exchanged messages, we show that some of BnB-ADOPT messages are redundant and can be removed from the search process without compromising its optimality and termination properties. As mentioned above, our intuition comes from the fact that BnB-ADOPT agents often send repeated information, which can be avoided under certain conditions. We present a new version, namely BnB-ADOPT$^+$, which avoids sending most of these redundant messages. Experimentally, we show that BnB-ADOPT$^+$ significantly decrements communication cost (the number of exchanged messages is divided by a number often larger than 3) on several widely used DCOP benchmarks.

3.1.1 Removing Redundant Messages in BnB-ADOPT

Initially, we consider the binary DCOP definition, where agents share binary cost functions. These results can be generalized to cost functions of any arity, as explained in Section 3.1.2.3.

Notation: i, j and k are agents executing BnB-ADOPT handling variables x_i, x_j and x_k respectively. (x_i, v) means that agent i holding variable x_i *assigns value* v. $context_i$ is the set of value assignments of agents located before i in its pseudo-tree branch (timestamps are not considered part of the context). $context_i[j]$ is the value assignment of agent j in $context_i$. For each value d and children c in agent i, the lower and upper bounds informed by c are

$lb_{i,c}(d)$/$ub_{i,c}(d)$. We recall that a message msg that arrives to j containing the assignment (x_i, v) with timestamp t *updates* $context_j[i]$ which has timestamp t' if and only if $t > t'$.

A message msg sent from i to j is *redundant* if at some future time t, the effect of other messages arriving j between msg and t would cause the same effect, so msg could have been avoided.

Lemma 1 *In BnB-ADOPT, if agent i sends two consecutive VALUE messages to agent j with timestamps t_1 and t_2 for x_i assignment, there is no message containing an x_i assignment with timestamp t such that $t_1 < t < t_2$.*

Proof. There is no VALUE message containing a timestamp between t_1 and t_2 for x_i assignment, since both VALUE messages are consecutive and sent from agent i. COST messages build their contexts from the information in VALUE messages. Since no VALUE message contains a timestamp between t_1 and t_2 for x_i assignment, no COST message will contain it. □

Theorem 1 *In BnB-ADOPT, if agent i sends to agent j two consecutive VALUE messages with the same val and th, the second message is redundant.*

Proof. Let V_1 and V_2 be two consecutive VALUE messages sent from agent i to agent j with the same value val and threshold th with timestamps t_1 and t_2, $t_1 \leq t_2$. Observe that between V_1 and V_2 agent i may have processed messages and its value assignment may have been reinitialized several times (by `InitSelf` procedure (Yeoh et al., 2010)), therefore even if V_1 and V_2 contain the same val and th values, their timestamps may be different. Observe also that between V_1 and V_2 any messages may arrive to j (coming from other agents).

Considering th, when V_1 reaches j:

1. If i is the parent of j, the th is copied in j.
2. If i is not the parent of j, the threshold th is ignored.

Assuming i is the parent of j and th is always copied, copying the same th value in j is redundant. It might be the case that j has reinitialized its th to ∞ as result of a context change in some higher neighbor. Observe that any higher neighbor connected with j is also an ancestor of i in the pseudo-tree (because i and j are on the same branch), so i will eventually receive a message as result of this context change and then it will send a new VALUE message to j with an updated th and val. Notice that maintaining $th = \infty$ in agent j during some time does not

compromises termination or completeness of the algorithm. To see this, it is enough to realize that thresholds are not used in (Yeoh et al., 2010) in any proof of Section 5 but in the proof of Lemma 8. However, the proof of Lemma 8 remains valid replacing threshold by ∞. Generally speaking, since th is an estimated upper bound of the current partial solution, when $th = \infty$ this property holds so it does not affect optimality or termination, although it could affect the pruning potentiality of the algorithm (thresholds are included to increase efficiency). For a more detailed explanation of maintaining efficient threshold management see Section 3.1.2.2.

Considering val, when V_1 reaches j the following cases are possible:

1. V_1 does not update $context_j[i]$. When V_2 arrives to j the following cases are possible:

   (a) V_2 does not update $context_j[i]$. Future messages will be processed as if V_2 has not been received, so V_2 is redundant.

   (b) V_2 updates $context_j[i]$ which has timestamp t. There are two options: (i) $t_2 > t > t_1$ and (ii) $t_2 > t = t_1$. Option (i) is impossible according to Lemma 1. Option (ii) is possible, but since $t = t_1$ the value val contained in V_2 is already in $context_j[i]$, so V_2 updates timestamps only. In future messages, every message accepted with timestamp t_2 in $context_j[i]$ would also be accepted with timestamp t_1 in $context_j[i]$. Therefore we conclude that V_2 is redundant.

2. V_1 updates $context_j[i]$. When V_2 arrives to j the following cases are possible:

   (a) V_2 does not update $context_j[i]$: as case (1.a).

   (b) V_2 updates $context_j[i]$: since V_1 updated $context_j$ and Lemma 1, the timestamp of $context_j[i]$ must be t_1. Updating with V_2 does not change $context_j[i]$ value but updates the timestamp of $context_j[i]$ from t_1 to t_2. Since there are no messages with timestamp between t_1 and t_2 (Lemma 1), any future message that could update $context_j$ with t_2 would also update it with t_1. So V_2 is redundant.

□

Lemma 2 *In BnB-ADOPT, if agent k sends two consecutive COST messages C_1 and C_2 with the same context and timestamps t_1 and t_2 respectively for the x_i assignment, and k has not detected a context change between C_1 and C_2, then there is no message with a timestamp t between t_1 and t_2 for x_i assignment incompatible with C_1 and C_2.*

Proof. Each time agent i changes value it sends a VALUE message to *all* its children and pseudo-children. This context change is eventually detected by k, either by a direct VALUE message from i (in this case k is constrained with i) or by a COST message from a descendant. Since there is no context change between C_1 and C_2, no message with timestamp between t_1 and t_2 can contain a x_i assignment incompatible with C_1 and C_2; otherwise agent k would have necessarily detected the context change. □

Theorem 2 *In BnB-ADOPT, if agent k sends to agent j two consecutive COST messages with the same information (context, lower/upper bound) and k has not detected a context change, the second message is redundant.*

Proof. Let C_1 and C_2 be two consecutive COST messages sent from k to j with the same context and lower/upper bounds, and $context_k$ has not changed between sending them. Any message may arrive to j between C_1 and C_2 (coming from other agents). Upon reception, the more recent values of C_1 (and later of C_2) are copied in $context_j$ (by `PriorityMerge` (Yeoh et al., 2010)). If j detects a context change, tables $lb_{j,c}(d)$ and $ub_{j,c}(d)$ could be reinitialized. If, after the priority merge, $context_j$ is still compatible with the COST context, tables $lb_{j,c}(d)$ and $ub_{j,c}(d)$ are updated with the information contained in the COST message.

Copying C_2 more recent values in $context_j$ is not essential. Let us assume that these values are not copied. Since there is no context change between C_1 and C_2, any message with a timestamp in between the timestamps of C_1 and C_2 will necessarily include a context compatible with C_2, according to Lemma 2. Therefore when C_2 arrives it updates timestamps only.

Since C_2 do not cause a context change in j, because it updates timestamps only, it does not cause a reinitialization of $lb_{j,c}(d)$, $ub_{j,c}(d)$. Because of that, our proof concentrates on bounds update.

When C_1 arrives, the following cases are possible:

1. After priority merge, C_1 is not compatible with $context_j$, its bounds are discarded. When C_2 arrives the following cases are possible:

   (a) After priority merge, C_2 is not compatible with $context_j$, its bounds are discarded. Bounds provided by C_2 are based on outdated information. So C_2 is redundant.

(b) C_2 is compatible with $context_j$, its bounds are included in j. Since C_1 was not compatible, there is at least one agent above j that changed its value. This value change has been received by j between C_1 and C_2 but has not yet been received by k (otherwise k would have detected a context change). Therefore there are one or several VALUE messages on its/their way towards k or k descendants. Upon reception, one or several COST messages will be generated. The last of them will be sent from k to j with more updated information. C_2 could have been avoided because a more updated COST will arrive to j. So C_2 is redundant.

2. After priority merge, C_1 is compatible with $context_j$, its bounds are included. When C_2 arrives to j the following cases are possible:

   (a) After priority merge, C_2 is not compatible with $context_j$, its bounds are discarded. Bounds provided by C_2 are based on outdated information. Since C_1 was compatible, there is at least one agent above j that changed its value. This information reached j between C_1 and C_2 but is still not detected by k. In the future a more updated COST will reach j (same reasons as previous case 1.b). So C_2 is redundant.

   (b) C_2 is compatible with $context_j$, it bounds are included but this causes no change in j bounds, unless bounds have been reinitialized. In the case of bounds reinitialization there is at least one agent above j that changed its value. The situation is the same as case (1.b). So C_2 is redundant. □

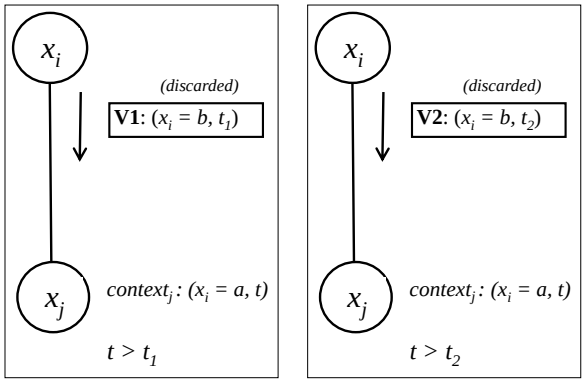
3.1.1.1 Example of Redundant VALUE messages

We present an example to illustrate some details of Theorem 1. Consider agents i and j, holding variables x_i and x_j respectively. Agent i sends two consecutive VALUE messages V_1 and V_2 to j with the same value and threshold. As explained in the previous Section, if V_2 does not update $context_j$ then V_2 is redundant. Now, we will consider two possible scenarios in which message V_2 changes $context_j$.

Case V_1 does not update $context_j$, and V_2 updates $context_j$:

As proved in Theorem 1, this scenario is impossible. Consider i sending message V_1 to j informing of the assignment $x_i = b$ with timestamp t_1 (Figure 3.1(a)). Agent j has in its context the assignment $x_i = a$ with timestamp $t > t_1$ (more updated) , so V_1 is discarded. Then, message V_2 is sent to j with timestamp t_2 (Figure 3.1(b)). If V_2 is accepted, then timestamp t_2 has to be more updated that t ($t_2 > t$). Then we get the case: $t_2 > t > t_1$ which

is impossible, because we know, from Lemma 1, that there can be no message with timestamp between t_1 and t_2.



(a) (b)

Figure 3.1: Case V_1 does not update $context_j$, and V_2 updates $context_j$

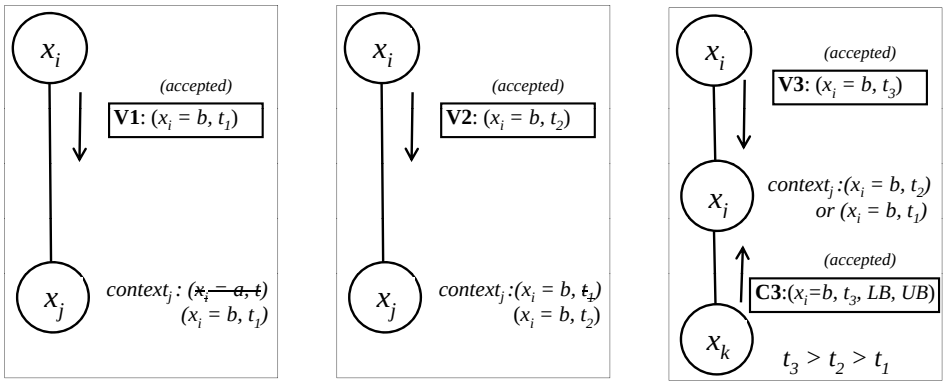
Case V_1 and V_2 update $context_j$:

Let us consider that i sends message V_1 to j informing the assignment $x_i = b$ with timestamp t_1, and is accepted (Figure 3.2(a)). Then $context_j[i]$ is updated and its timestamp is set to t_1. Between V_1 and V_2 many messages may arrive to j with different timestamps, and $context_j$ might be updated. But if V_2 is accepted, we can assure that timestamp in $context_j[i]$ must be t_1 when message V_2 arrives (otherwise V_2 would not have been accepted). Upon V_2 reception, $context_j[i]$ remains the same (because V_1 and V_2 contains the same assignment) and only timestamp t_2 is updated (Figure 3.2(b)). Since there is no message with timestamp between t_1 and t_2 (Lemma 1) is easy to see that any VALUE message V_3 or COST message C_3 that would update $context_j[i]$ having timestamp t_2, will also update it if $context_j[i]$ would have timestamp t_1 (Figure 3.2(c)). So updating t_2 in $context_j[i]$ makes no difference on future message processing. Therefore, V_2 is redundant.

3.1.1.2 Example of Redundant COST messages

In the following we present an example to illustrate some details of Theorem 2 when messages are delayed during execution.

Consider agents k and j, holding variables x_k and x_j respectively. Agent k sends two consecutive COST messages C_1 and C_2 to j. As explained in the previous Section, if after



(a) (b) (c)

Figure 3.2: Case V_1 and V_2 update $context_j$

priority merge C_2 bounds are discarded then C_2 is redundant since bounds LB and UB are not updated and its reception has no effect in future message processing. Now, we will consider two possible scenarios in which message C_2 bounds are accepted.

Case C_1 bounds are discarded, and C_2 bounds are accepted:

Consider a higher agent i connected with j and k. Variable x_i changes its value from a to b and i sends the correspondent VALUE messages (informing $x_i = b$) to j and k. Message sent to j is received and $context_j[i]$ is updated, but message sent to k is delayed (Figure 3.3 (a)). Now, agent k sends a COST message C_1 to j and this message is discarded because contexts are incompatible on variable x_i, since k has not received the last VALUE message from i yet (Figure 3.3 (b)). Between messages C_1 and C_2 other messages may arrive, so let us assume that i changes its value again to a and sends the correspondent VALUE messages to j and k, but message to k is once again delayed (Figure 3.3 (c)). With these messages delayed, there is no context change in k and message C_2 is sent. Message C_2 is accepted in j (since contexts are now compatible) and j bounds are updated (Figure 3.3(d)). However, as there are still 2 delayed VALUE messages from i to k, we can assure that when they arrive to k there will be a context change, so a new COST message C_3 will be generated with more updated information (Figure 3.3(e)). As we can see, message C_2 can be ignored since a message C_3 will eventually arrive to j and update its bounds with more recent information.

Case C_1 and C_2 bounds accepted:

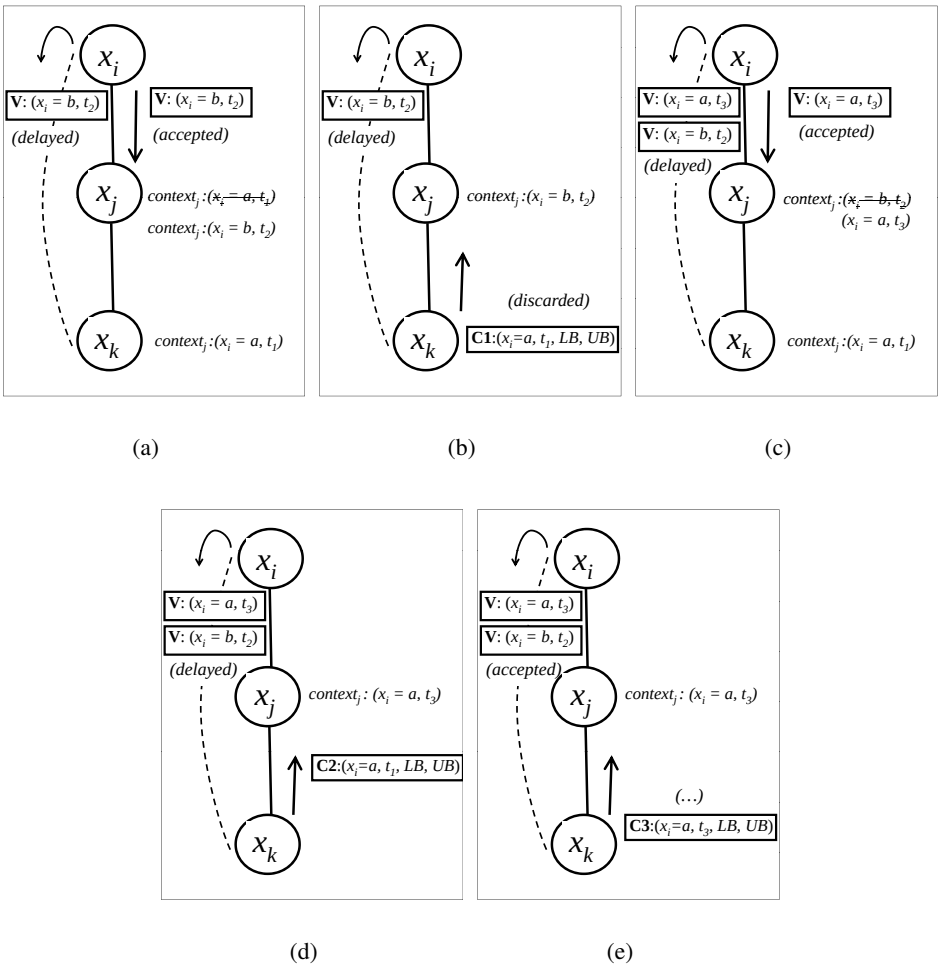


Figure 3.3: Case C_1 bounds are discarded, and C_2 bounds are accepted

Consider that agent k sends message C_1 to j and its bounds are accepted because contexts are compatible. In this case bounds LB and UB are updated in j. If message C_2 arrives right after C_1, it is easy to realize that is redundant, since it would copy the same information in j. However, between C_1 and C_2 some messages may arrive, and as result of this the bounds informed by C_1 might be reinitialized. This could only happen if a higher agent i changes its value. So if a higher agent i changes its value from a to b, when the corresponding VALUE messages arrives to j its bounds are reinitialized. After this i must change again its value to a if we want a scenario where C_2 message is accepted. In this case, as there is no context change in k, we can assure that there are delayed VALUE messages on their way to k (as in Figure

3.3 (b) and (c)). When those delayed messages arrive to k there will be a context change, generating a new COST message C_3 with more updated information (same case as Figure 3.3 (e)). So message C_2 can be ignored since a message C_3 will eventually arrive to j and update its bounds.

3.1.1.3 Correctness and Completeness

Temporarily, we define BnB-ADOPT$^+$ as our version of BnB-ADOPT with the following changes:

- The second of two consecutive VALUE messages with the same i, j, val and th is not sent.
- The second of two consecutive COST messages with the same k, j, $context$, lb and ub when k detects no context change is not sent.

These changes do not affect optimal solving, as proved next.

Theorem 3 *BnB-ADOPT$^+$ terminates with the cost of a cost-minimal solution.*

Proof. By Theorem 1 and 2 messages not sent by BnB-ADOPT$^+$ are redundant. At some future time, the effect of other messages arriving to the receiver agents will cause the same effect, so these messages can be removed. BnB-ADOPT terminates with the cost of a cost-minimal solution (Yeoh et al., 2010), so BnB-ADOPT$^+$, without redundant messages, also terminates with the cost of a cost-minimal solution. □

3.1.1.4 Efficient Threshold Management

Experimentally, BnB-ADOPT$^+$ as described in 3.1.1.3 caused minor benefits. We realized that we have ignored threshold management. Although a VALUE message is sent every time the th value changes in the sender agent, we did not considered that this th may be reinitialized to ∞ in the receiver agent. Thresholds are reinitialized to ∞ after a context change (caused by VALUE or COST messages); this causes no special difficulty in the original BnB-ADOPT algorithm because VALUE messages are sent invariably to children and pseudo-children every time the message queue is processed, so thresholds are soon updated in children. Now, if some

of these VALUE messages are not sent, children may run the algorithm with an ∞ threshold during some periods (until its parent changes its val or th).

Having an ∞ threshold does not affect optimality and termination. Observe that thresholds are used for pruning under the condition: $LB(d) \geq min\{TH, UB\}$. If threshold is ∞ the pruning condition simply reduces to: $LB(d) \geq UB$. But if agents do not use the tightest bound for pruning, performance can decrease substantially.

To avoid this, children should have a way to ask for the threshold to their parents after reinitialization. This is done using a flag in COST messages, which are sent from children to parents. Thus, we define BnB-ADOPT$^+$ as our BnB-ADOPT version as follows:

1. Agents remember the last VALUE and COST messages sent to every neighbor.
2. COST messages include a boolean $ThRequest$ set to true if the sender threshold has been reinitialized.
3. If i has to send j a VALUE message equal to the last message sent (ignoring timestamps), the new VALUE message is sent if and only if the last COST message that i received from j had $ThRequest = true$ and i threshold is not ∞.
4. If j has to send i a COST message equal to the last message sent (ignoring timestamps), the new COST message is sent if and only if j has detected a context change between them.

It is immediate to see that this version maintains the optimality and termination condition of BnB-ADOPT$^+$. Some messages are sent to update children thresholds more rapidly when they have been reinitialized. Original BnB-ADOPT, that includes redundant messages, terminates with the minimal solution cost (Yeoh et al., 2010); then, sending *some* of those redundant messages the algorithm remains optimal and terminates.

The proposed changes to avoid redundant messages can also be applied to the ADOPT algorithm, as we show in Appendix A, producing the new algorithm ADOPT$^+$. However in this thesis we work with BnB-ADOPT$^+$ because it is more efficient in practice.

3.1.2 N-ary Cost Functions in BnB-ADOPT

In DCOPs, it is somehow natural to use binary cost functions. It is usually assumed that each agent holds a single variable and that agents communicate through messages from a sender to

a receiver agent, this naturally brings to a binary relation between agents and to binary cost functions. However, in some cases an agent may have a cost function of higher arity with a subset of agents. ADOPT and BnB-ADOPT can be extended to support cost functions of any arity. The original ADOPT (Modi et al., 2005) proposes a way to deal with n-ary cost functions (with arity higher than two), and BnB-ADOPT takes the exact same strategy (Yeoh et al., 2010). The extension proposed for ADOPT to handle n-ary cost functions, described in (Modi et al., 2005), is as follows [1] :

> ...a ternary constraint f_{ijk} ... defined over three variables x_i, x_j, x_k ... Suppose x_i and x_j are ancestors of x_k... With our ternary constraint, both x_i and x_j will send VALUE messages to x_k. x_k then evaluates the ternary constraint and sends COST messages back up the tree as normal ... Thus, we deal with an n-ary constraint by assigning responsibility for its evaluation to the lowest agent involved in the constraint. The only difference between evaluation of an n-ary constraint and a binary one is that the lowest agent must wait to receive all ancestors' VALUE messages before evaluating ...

In other words (replacing "constraint" by "cost function"), in the proposed extension agents must send their VALUE messages to the lowest agent of the pseudo-tree involved in a cost function. In the case of a binary cost function, the lower agent in the pseudo-tree (of the two involved in the cost function) always receives VALUE messages. In the case of n-ary cost functions (involving more than two agents), intermediate agents do not receive VALUE messages from the rest of the agents involved in that function. The lowest agent k must receive all VALUE messages before evaluating the cost function. Because of this, it is called the *evaluator* agent. Upon reception of all these messages, x_k evaluates the n-ary cost function and sends a COST message to its parent, which receives and processes it as any other COST message.

When applying this technique to BnB-ADOPT some issues appear. This strategy may cause inefficiency and an incorrect final assignment on BnB-ADOPT agents. In the following we details these issues and provide a simple way to correct them, which can be easily integrated in BnB-ADOPT.

[1] It must be assured that all agents involved in an n-ary cost function lie on the same branch of the pseudo-tree. This is guaranteed since agents sharing n-ary cost functions form a clique in the constraint graph. When performing a depth first traversal to construct the pseudo-tree, agents of the clique will necessarily lie on the same branch.

3.1.2.1 Termination

In binary BnB-ADOPT, each time an agent changes its value assignment it sends VALUE messages to its children and pseudo-children. A *non-root* agent terminates when it reaches the termination condition $LB = UB$ after receiving a TERMINATE message from its parent (for the *root* agent no TERMINATE message is needed). As a side-effect, the last value taken by all agents is the optimal value. This feature is very appreciated in distributed environments, because optimal values are distributed among agents without requiring a central agent in charge of the whole solution.

In n-ary BnB-ADOPT, although the *root* agent computes the optimum cost, a direct implementation may not terminate with an optimal value assigned to every agent. Let us consider a ternary cost function among agents i, j and k (as in Figure 3.4); i is at the root of the pseudo-tree, and k evaluates the cost function. Agent i may explore its last value, jump back to its best value where it reaches termination condition $LB = UB$, sends a VALUE message to k and a TERMINATE message to its child j. Upon reception of VALUE message, k will send a COST message to j. This COST message contains the last assignment (optimal value) made by i. But if j has processed before the TERMINATE message from i, it will end without processing that COST message, which means that j will end with an outdated context: [1] this causes j to end with an assigned value which may not be an optimal one, since it does not minimize the cost of the global solution. [2]

A simple way to correct this is to include in TERMINATE messages the last assignment made by the sender agent. In this way, the receiver can update its context and terminate with the value that minimizes the lower bound.

3.1.2.2 Efficient Threshold Management

If the ADOPT strategy for n-ary cost functions is applied literally to BnB-ADOPT, there are scenarios which are inefficiently solved. For instance, consider Figure 3.4, where variables of agents i, j, k share a ternary cost function. Agent i is the root of the pseudo-tree and k is the lowest agent (therefore the evaluator) of the ternary cost function. Suppose j is constrained with further agents of the problem, represented as a gray subtree. Since VALUE messages are sent from i and j to k to inform value changes, but no VALUE messages are sent from i to

[1] This would not happen if agent i would have sent VALUE messages to any child j informing of value changes, but this is not the original BnB-ADOPT strategy to deal with n-ary cost functions.

[2] Technically speaking, j might terminate with an *incorrect* context.

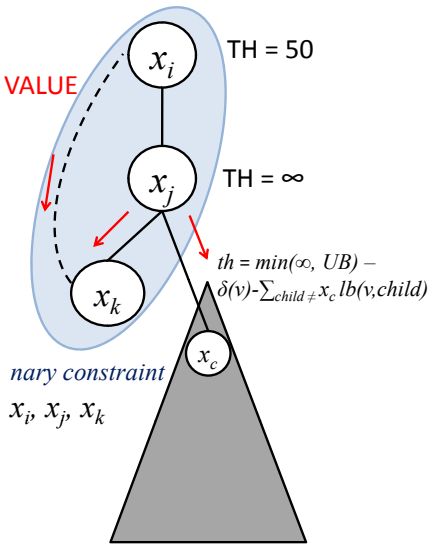


Figure 3.4: Original BnB-ADOPT dealing with n-ary constrains, use of VALUE messages.

j, agent j will not have a threshold provided by its parent. Furthermore, j will not be able to calculate for its children the tightest possible threshold since its own TH is ∞ and this effect propagates to all its subtree. Thresholds were introduced in BnB-ADOPT to speed up problem resolution and increase pruning opportunities, so not having the tightest threshold on agent j –and on j subtree– is clearly a drawback for performance.

A simple way to avoid this issue is to send VALUE messages to all descendants (children or pseudo-children). However, this is more than needed. We can avoid unnecessary messages by only sending VALUE messages to the lowest agent in charge of evaluating the cost function –to generate COST messages with updated LB and UB– and to children –to propagate the TH value down in the pseudo-tree–. Any agent involved with i in a cost function which is neither the evaluator of the cost function nor a child of i does not need to receive a VALUE message from i. This is our proposed extension for BnB-ADOPT to deal with n-ary cost functions. We define n-ary BnB-ADOPT as follows:

- As in (Yeoh et al., 2010), each agent sends VALUE messages to the evaluator agent of the cost functions it is involved in.
- Each agent sends its VALUE messages to all its children in the pseudo tree.

Observe that in the binary case, this proposal collapses into the existing operation for both algorithms. It is easy to show that our n-ary BnB-ADOPT terminates with the optimal solution, as proved next.

3.1.2.3 Correctness and Completeness

First, we prove that if agents send VALUE messages to their children and pseudo-children, this extended n-ary BnB-ADOPT terminates with the optimum cost. Second, we show that VALUE messages send to pseudo-children are redundant (except if the pseudo-child is the evaluating agent of a cost function that involved the sender). Third, we demonstrate that redundant messages in the binary case remain redundant in the n-ary case. Combining these results we obtain the desired output: n-ary BnB-ADOPT$^+$ terminates with the optimum cost.

Theorem 4 *N-ary BnB-ADOPT terminates with the minimum solution cost.*

Proof. Imagine an extended version of n-ary BnB-ADOPT where agents send VALUE messages to all their neighbors (children and pseudo-children) below the pseudo-tree. This extended version of n-ary BnB-ADOPT is working as in the binary case: each agent sends VALUE messages to all its descendants (children or pseudo-children) and it sends a COST message to its parent. In this case, it is easy to check that all results of Section 5 in (Yeoh et al., 2010) apply here (observe that no result of Section 5 in (Yeoh et al., 2010) use the fact that cost functions are binary). In particular, in (Yeoh et al., 2010) it is proved that binary BnB-ADOPT terminates with the minimal solution cost. Therefore, this extended version of n-ary BnB-ADOPT terminates with the minimal solution cost.

Now, we consider VALUE messages sent from agent i to pseudo-children, such that none of these pseudo-children is the evaluator of a cost function involving agent i. We show that these messages are redundant.

After receiving a VALUE message from i, agent j does the following:

1. Agent j updates its context.
2. If j detected a context change, j may reinitialize some $lb_{j,c}(d)$ and $ub_{j,c}(d)$ if c is a pseudo-child of i.
3. If the message comes from its parent, j rewrites its own threshold with the message threshold.

We know that i is not the parent of j, so we consider point (1) and (2) only. Agent j is not the evaluating agent of a cost function involving i; then, there is another agent k –in the same branch and below i and j– in charge of such evaluation. This agent k will receive –for sure– the VALUE messages coming from its ancestors, and then it will send COST messages up the tree. When these COST messages reach j, they will update its context and perform the required reinitialization in lb/ub exactly in the same way as after receiving j a VALUE message from i. Original VALUE messages are redundant because the same effect can be obtained with the COST messages arriving from k. Therefore, we can remove these VALUE messages from our extended version of n-ary BnB-ADOPT, and the algorithm will terminate with the minimal solution cost. This algorithm is the proposed n-ary BnB-ADOPT. □

Next we prove that redundant messages in the binary case remain redundant in the n-ary case.

Lemma 3 *VALUE and COST messages found redundant for binary BnB-ADOPT, remain redundant for n-ary BnB-ADOPT.*

Proof. To proof this lemma, it is enough to realize that Theorems 1 and 2 remain valid for n-ary BnB-ADOPT. Observe that in the proofs of Lemma 1, Theorems 1 and 2 it is not required the use of binary cost functions. The proof of Lemma 2 can be easily generalized to the n-ary case replacing "to all its children and pseudo-children" by "to all its children and evaluator pseudo-children". □

We define n-ary BnB-ADOPT$^+$ as n-ary BnB-ADOPT removing redundant VALUE and COST messages, as in Section 3.1.1.

Corolary 1 *N-ary BnB-ADOPT$^+$ terminates with the minimal solution cost.*

Proof. Combining Theorem 4 and Lemma 3 we prove that n-ary BnB-ADOPT not sending redundant VALUE or COST messages terminates with the minimal solution cost.

As in Section 3.1.1.4, a child may ask its parent to resend the threshold in a VALUE message to improve performance if its threshold has been reinitialized.

3.1.3 Experimental Results

We evaluated experimentally the performance of original BnB-ADOPT against our new version BnB-ADOPT$^+$ in the binary and n-ary case.

First, we evaluate the reduction caused by removing redundant messages in the binary case, comparing original BnB-ADOPT against BnB-ADOPT$^+$.

Second, we evaluate performance in n-ary instances, comparing:

1. N-ary BnB-ADOPT with the modification described in Section 3.1.2.1 to avoid an incorrect final assignment of agents.

2. Our proposal of n-ary BnB-ADOPT as described in Section 3.1.2.2 (efficient threshold management).

3. N-ary BnB-ADOPT$^+$ (correct final assignment of agents, efficient threshold management and removing redundant messages).

Lastly, we compare in binary instances BnB-ADOPT$^+$ against two other well-known algorithms for DCOP solving: Synchronous Branch and Bound (SBB) (Hirayama and Yokoo, 1997) and Asynchronous Forward Bounding (AFB) (Gershman et al., 2009). SBB is a completely synchronous algorithm whereas AFB performs synchronous value assignments but computes asynchronously bounds used for pruning. SBB and AFB maintain a total order of variables to perform assignments while BnB-ADOPT$^+$ uses a partial ordering following the pseudo-tree structure. We present this last comparison to provide an overall picture of BnB-ADOPT$^+$ and how its asynchronous nature affects the number of messages exchanged and computation.

Experiments are performed in three different benchmarks: random DCOPs (binary and ternary cases), meeting scheduling and sensor networks (both are binary, obtained from a public DCOP repository (Yin, 2008)). Random DCOPs are characterized by $\langle n, d, p_1 \rangle$, where n is the number of variables, d is the domain size and p_1 is the network connectivity. Binary instances contain $p_1 * n(n-1)/2$ binary cost functions, while ternary instances contain $p_1 * n(n-1)(n-2)/6$ ternary cost functions. Costs are selected randomly from the set {0,..., 100}.

Results of the first experiment comparing BnB-ADOPT and BnB-ADOPT$^+$ appear in Table 3.1 and 3.2. Table 3.1 shows results on binary random problems averaged over 50 instances. Table 3.1(a) shows results varying network connectivity with $\langle n = 10, d = 10, p_1 = 0.2...0.8 \rangle$. Table 3.1(b) shows results varying domain size with $\langle n = 10, d = 6...12, p_1 = 0.5 \rangle$. Table 3.1(c) shows results varying the number of variables with $\langle n = 6...12, d = 10, p_1 = 0.5 \rangle$. Table 3.2 (a) shows meeting scheduling instances in 4 cases with different hierarchical scenarios: case A (8 variables), B (10 variables), C (12 variables) and D (12 variables). Table 3.2 (b) shows sensor network instances in 4 cases with different topologies: cases A (16 variables), B

(16 variables), C (10 variables) and D (16 variables). In these two last benchmarks, results are averaged over 30 instances.

Experiments with binary random DCOPs show that our algorithm BnB-ADOPT$^+$ obtains important savings in communication with respect to original BnB-ADOPT. Messages are reduced by a factor from 3 to 6 when connectivity and domain size increases, also showing a consistent reduction, between a factor of 4 and 5, when increasing the number of variables. For meeting scheduling instances, messages are reduced by a factor between 3 and 9, and for sensor network by a factor between 5 and 8. The standard deviation of messages also decreases in all problems considered.

Regarding NCCCs, the mean is moderately reduced in all instances (around 10%). In the binary random benchmark, the standard deviation is also slightly reduced. In meeting scheduling and sensor network instances, the standard deviation of NCCCs increases however if looking at every problem separately, the number of NCCC of BnB-ADOPT$^+$ is always smaller in every instance. Cycles remain practically unchanged.

These results clearly indicate that, in the binary case, removing redundant messages is very beneficial for enhancing communication, achieving also moderated gains in computation.

In addition, we took a particular random binary instance $\langle n = 10, d = 10, p_1 = 0.5\rangle$, and solved it repeatedly by original BnB-ADOPT and BnB-ADOPT$^+$, varying the order in which agents are activated in the simulator (using the same DFS pseudo-tree in all executions). Results were quite similar across executions. Regarding saved messages, BnB-ADOPT always required between 4.3 and 4.4 times more messages than BnB-ADOPT$^+$ (considering individual executions). These results show that the activation order of agents in the simulator has no impact in the message reduction caused by BnB-ADOPT$^+$.

Results of the second experiment appear in Table 3.3, which contains results of ternary random instances with $\langle n = 8, d = 5, p_1 = 0.4...0.8\rangle$ averaged over 50 instances. First row contains results of original BnB-ADOPT (including the modification of Section 3.1.2.1) to assure a correct termination. Second row contains results for our proposal for n-ary BnB-ADOPT (Section 3.1.2), where thresholds are propagated to children to assure an efficient threshold management. Third row contains n-ary BnB-ADOPT$^+$ results, which enhances this last version removing redundant messages.

(a) $< n = 10, d = 10, p_1 >$

p_1	Algorithm	#Messages	#NCCC	#Cycles
	BnB-ADOPT	1068 (274)	904 (23)	**62 (15)**
0.2	BnB-ADOPT$^+$	**416 (74)**	**881 (23)**	**62 (15)**
	BnB-ADOPT	39,158 (36,578)	68,882 (62,180)	**1,751 (1,625)**
0.3	BnB-ADOPT$^+$	**11,774 (10,105)**	**62,031 (53,085)**	1,753 (1,629)
	BnB-ADOPT	270,379 (432,782)	504,373 (796,625)	**10,313 (16,478)**
0.4	BnB-ADOPT$^+$	**69,277 (92,291)**	**475,534 (776,820)**	10,317 (16,483)
	BnB-ADOPT	2,273,768 (2,149,369)	4,311,524 (3,923,577)	**73,715 (69,676)**
0.5	BnB-ADOPT$^+$	**493,137 (422,360)**	**4,112,299 (3,760,583)**	73,792 (69,808)
	BnB-ADOPT	11,439,563 (10,231,971)	23,759,356 (22,468,476)	**331,947 (299,259)**
0.6	BnB-ADOPT$^+$	**2,205,848 (1,802,655)**	**22,783,209 (21,040,893)**	332,841 (300,784)
	BnB-ADOPT	60,221,283 (34,121,853)	134,868,051 (90,469,274)	**1,526,394 (862,540)**
0.7	BnB-ADOPT$^+$	**8,930,713 (5,092,602)**	**129,143,706 (89,328,458)**	1,527,960 (865,974)
	BnB-ADOPT	161,327,710 (94,398,879)	360,857,244 (212,464,295)	**3,752,164 (2,210,488)**
0.8	BnB-ADOPT$^+$	**22,972,676 (13,464,530)**	**353,180,585 (209,726,371)**	3,755,118 (2,213,631)

(b) $< n = 10, d, p_1 = 0.5 >$

d	Algorithm	#Messages	#NCCC	#Cycles
	BnB-ADOPT	618,005 (573,704)	701,352 (642,821)	**20,305 (18,869)**
6	BnB-ADOPT$^+$	**119,841 (104,980)**	**657,276 (593,830)**	20,342 (18,924)
	BnB-ADOPT	1,362,586 (951,900)	2,090,231 (1,470,631)	**44,507 (31,209)**
8	BnB-ADOPT$^+$	**288,422 (201,488)**	**1,986,430 (1,398,420)**	44,562 (31,238)
	BnB-ADOPT	2,711,719 (2,929,759)	5,092,387 (5,376,943)	**88,224 (96,033)**
10	BnB-ADOPT$^+$	**597,325 (633,879)**	**4,842,265 (5,133,731)**	88,329 (96,195)
	BnB-ADOPT	4,871,563 (9,725,100)	10,969,641 (20,549,608)	**157,856 (314,908)**
12	BnB-ADOPT$^+$	**1,015,541 (1,706,302)**	**10,342,414 (18,679,208)**	157,994 (315,137)

(c) $< n, d = 10, p_1 = 0.5 >$

n	Algorithm	#Messages	#NCCC	#Cycles
	BnB-ADOPT	4,388 (3,272)	13,077 (13,214)	**350 (259)**
6	BnB-ADOPT$^+$	**1,514 (1,020)**	**12,221 (12,439)**	**350 (259)**
	BnB-ADOPT	72,783 (54,772)	173,038 (126,743)	**3,576 (2,679)**
8	BnB-ADOPT$^+$	**20,326 (12,971)**	**159,698 (113,801)**	3,581 (2,689)
	BnB-ADOPT	2,603,727 (3,358,285)	5,289,823 (6,844,174)	**84706 (112,469)**
10	BnB-ADOPT$^+$	**547,079 (656,709)**	**5,005,774 (6,576,047)**	84816 (112,774)
	BnB-ADOPT	111,436,193 (133,362,317)	187,178,211(237,619,542)	**2,633,456 (3,148,339)**
12	BnB-ADOPT$^+$	**20,169,771 (23,877,564)**	**179,110,208 (228,862,664)**	2,636,675 (3,152,543)

Table 3.1: Results (mean and standard deviation between parenthesis) of random binary benchmarks varying network connectivity, domain size and number of variables: BnB-ADOPT (first row), BnB-ADOPT$^+$ (second row).

(a) Meeting Scheduling

	Algorithm	#Messages	#NCCC	#Cycles
A	BnB-ADOPT	178,899 (3,638)	446,670 (2,786)	**8,202 (302)**
	BnB-ADOPT$^+$	**18,117 (627)**	**413,507 (13,446)**	8,203 (302)
B	BnB-ADOPT	65,556 (912)	125,331 (1,963)	**2,663 (43)**
	BnB-ADOPT$^+$	**15,373 (426)**	**120,900 (1,969)**	2,665 (43)
C	BnB-ADOPT	62,707 (741)	80,369 (54)	**2,353 (34)**
	BnB-ADOPT$^+$	**11,343 (347)**	**74,518 (407)**	2,355 (35)
D	BnB-ADOPT	41,282 (862)	60,424 (460)	**1,545 (48)**
	BnB-ADOPT$^+$	**13,354 (455)**	**49,878 (1692)**	1,547 (48)

(b) Sensor Network

	Algorithm	#Messages	#NCCC	#Cycles
A	BnB-ADOPT	9,369 (99)	7,241 (52)	313 (3)
	BnB-ADOPT$^+$	**1,103 (73)**	**450 (232)**	**307 (4)**
B	BnB-ADOPT	12,917 (116)	11,054 (135)	**414 (4)**
	BnB-ADOPT$^+$	**1,569 (77)**	**592 (879)**	409(4)
C	BnB-ADOPT	6,429 (59)	8,786 (52)	**340 (5)**
	BnB-ADOPT$^+$	**1,177 (51)**	**1,495 (2,490)**	**340 (6)**
D	BnB-ADOPT	15,560 (145)	12,641 (57)	**477 (2)**
	BnB-ADOPT$^+$	**2,155 (81)**	**2,137 (3,552)**	**477 (2)**

Table 3.2: Results (mean and standard deviation between parenthesis) of meeting scheduling and sensor network instances: BnB-ADOPT (first row), BnB-ADOPT$^+$ (second row).

Experiments with ternary random DCOPs show that assuring the propagation of threshold values to children produces clear benefits in performance (Table 3.3, second row). Agents send some extra VALUE messages to children containing the threshold, but these extra messages contribute to a better pruning. As a global effect, less communication is required in the overall search, and significant reductions are obtained in all metrics (messages, NCCCs and cycles). Maintaining this positive effect, we remove redundant messages (Table 3.3, third row). Removing redundant messages causes savings up to one order of magnitude in the number of messages exchanged. We consider this result very positive since execution time is often dominated by communication time. Observe that the number of cycles have very little variation between the second and third row. Also, there are slight savings in NCCCs, although they are not very significant. From these results we conclude that, in the n-ary case, our proposal for n-ary BnB-ADOPT causes clear benefits in communication and computation, and removing redundant messages substantially reduces communication.

Finally, we present a third experiment comparing BnB-ADOPT$^+$ with SBB and AFB on Figure 3.5. These experiments were performed on random binary instances with $\langle n = 10, d = 10, p_1 = 0.2...0.8 \rangle$ (up), meeting scheduling (center) and sensor network (down)

p_1	Algorithm	#Messages	#NCCC	#Cycles
	n-ary BnB-ADOPT	257,238 (294,137)	1,522,580 (1,863,696)	10,880 (12,295)
	our n-ary BnB-ADOPT	147,379 (139,646)	829,594 (852,754)	**5,793 (5,357)**
0.2	n-ary BnB-ADOPT$^+$	**28,614 (23,822)**	**764,516 (783,253)**	5,797 (5,360)
	n-ary BnB-ADOPT	648,217 (413,580)	4,029,045 (2,807,389)	24,026 (15,085)
	our n-ary BnB-ADOPT	401,306 (239,271)	2,414,946 (1,641,836)	**13,938 (8,271)**
0.3	n-ary BnB-ADOPT$^+$	**63,778 (33,025)**	**2,237,786 (1,520,761)**	13,943 (8,270)
	n-ary BnB-ADOPT	1,642,247 (975,131)	12,585,339 (8,483,693)	55,156 (32,553)
	our n-ary BnB-ADOPT	1,210,143 (523,825)	9,194,309 (4,938,582)	**39,373 (17,188)**
0.4	n-ary BnB-ADOPT$^+$	**164,901 (68,876)**	**8,744,465 (4,813,577)**	39,381 (17,197)
	n-ary BnB-ADOPT	2,321,729 (1,106,146)	19,424,669 (10,248,817)	74,279 (36,150)
	our n-ary BnB-ADOPT	1,771,256 (678,893)	14,775,187 (6,678,667)	**55,101 (21,280)**
0.5	n-ary BnB-ADOPT$^+$	**225,836 (69,630)**	**14,337,952 (6,372,442)**	55,103 (21,280)
	n-ary BnB-ADOPT	3,666,514 (1,316,782)	35,743,718 (15,729,105)	115,523 (43,001)
	our n-ary BnB-ADOPT	2,973,239 (1,406,400)	29,252,469 (16,146,978)	**92,020 (45,010)**
0.6	n-ary BnB-ADOPT$^+$	**311,524 (93,062)**	**27,614,749 (14,316,979)**	**92,020 (45,010)**
	n-ary BnB-ADOPT	4,013,891 (897,046)	41,469,157 (11,804,005)	124,408 (29,353)
	our n-ary BnB-ADOPT	3,537,027 (1,119,242)	36,966,889 (13,604,359)	**108,858 (36,028)**
0.7	n-ary BnB-ADOPT$^+$	**348,199 (73,620)**	**35,314,718 (12,331,316)**	**108,858 (36,028)**
	n-ary BnB-ADOPT	4,892,733 (788,897)	55,151,742 (11,209,964)	150,077 (25,114)
	our n-ary BnB-ADOPT	4,616,032 (1,135,830)	52,221,369 (14,948,424)	**140,472 (35,672)**
0.8	n-ary BnB-ADOPT$^+$	**399,662 (70,572)**	**49,189,230 (13,197,765)**	**140,472 (35,672)**

Table 3.3: Results (mean and standard deviation between parenthesis) on random ternary DCOPs: original n-ary BnB-ADOPT with correct termination (first row), our proposal of n-ary BnB-ADOPT for efficient threshold management (second row), and n-ary BnB-ADOPT$^+$ (third row).

instances. SBB variables were statically ordered using the width heuristic described in (Hirayama and Yokoo, 1997), AFB variables were ordered following this same heuristic and BnB-ADOPT$^+$ variables were partially ordered using a most-connected heuristic when constructing the pseudo-tree. We restrict the solving time to one hour; in the case of SBB and AFB if a problem instance could not be solved in this amount of time, we present the amount of messages exchanged and NCCCs performed until timeout.

We can observe from this results that, for random instances, BnB-ADOPT$^+$ is significantly more efficient in low connected problems, however in tightly connected problems it requires more messages and computational effort than SBB and AFB. We explain this behavior given that BnB-ADOPT$^+$ is an asynchronous algorithm designed to benefit from a pseudo-tree structure, where non-connected agents lying on different branches of the pseudo-tree can explore the search space in parallel. When connectivity increases, the wide of the pseudo-tree decreases (in a fully connected problem the pseudo-tree has only one single branch where all agents are totally ordered). This makes BnB-ADOPT$^+$ asynchronous potential to decrease. At the same time, a higher number of reinitialization (bounds and context) are performed, since agents have

more links to ancestors, which reduces pruning effectiveness.

For the meeting scheduling and sensor network instances, we can see that BnB-ADOPT$^+$ is several orders of magnitude more efficient than SBB and AFB. The structured nature of these problems (with a different topology for every case A,B,C or D (Maheswaran et al., 2004) suitable to build balanced pseudo-trees) allows BnB-ADOPT$^+$ to benefit from asynchronous

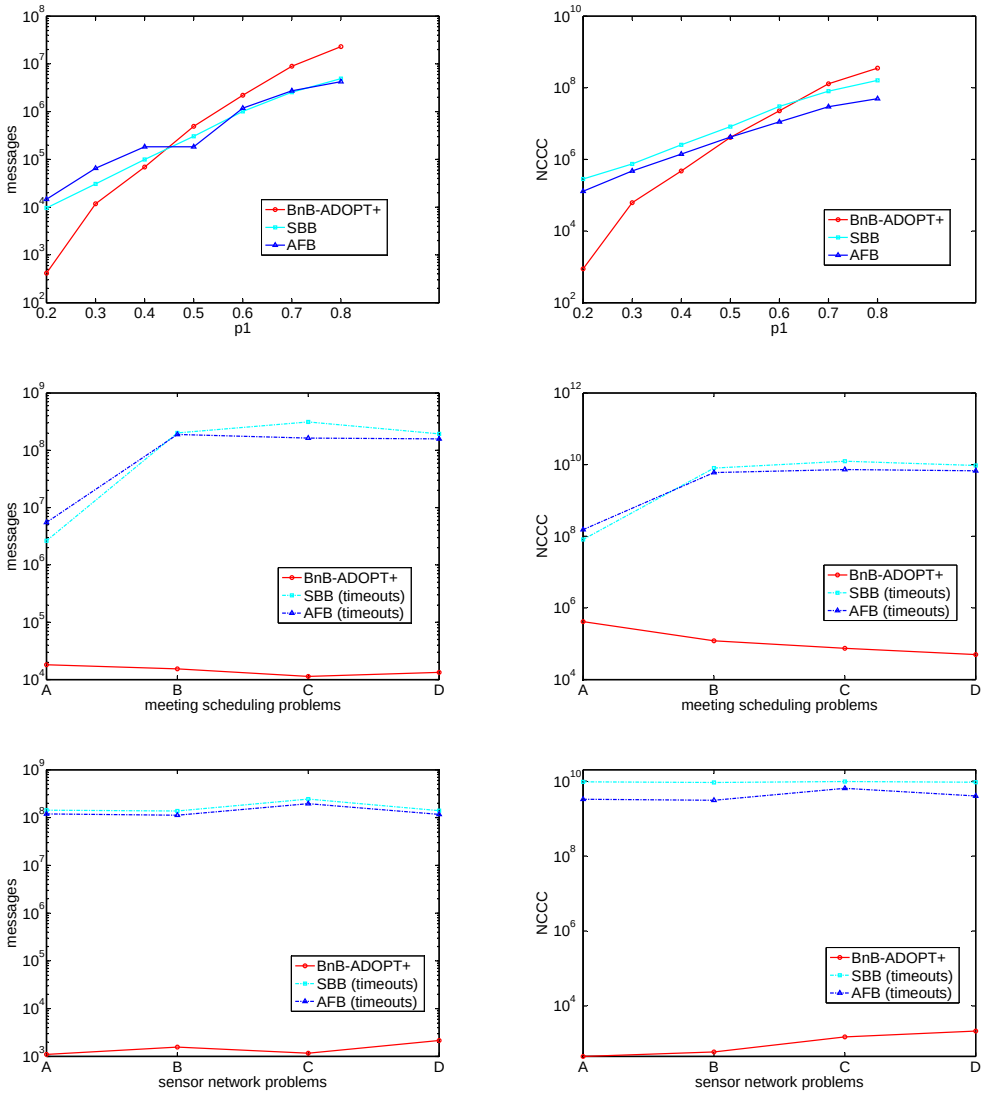


Figure 3.5: Comparison of algorithms BnB-ADOPT$^+$, SBB, AFB. Up: binary random instances. Center: meeting scheduling. Down: sensor networks.

search. In addition, we observed that in these instances the variability of costs is smaller than in random problems: most costs are quite similar, while some others are clearly larger. In these cases, an upper bound close to the optimum cost is reached early in the execution. However to satisfy the pruning condition, lower bounds contributions from almost all cost functions are needed. We observed that SBB and –to a lesser extent– AFB have to go deep in the search tree to obtain such contributions (pruning is usually done in the last agents of the ordering) and finally they are subject to thrashing. On the other hand, BnB-ADOPT$^+$ computes local bounds on every agent since all agents are assigned at every moment of the execution. By the use of thresholds, specialized upper bounds are propagated to every node of the pseudo tree. If the amount of bound reinitialization is not very high, this allows BnB-ADOPT$^+$ to reduce the search space faster than AFB and SBB. We have confirmed this fact empirically testing on several instances with small or large costs but with very small variability (including artificial cases where all tuples have the same cost) obtaining the same behavior.

In summary, we can see that the proposed BnB-ADOPT$^+$ algorithm is clearly more efficient than original BnB-ADOPT. In binary instances, BnB-ADOPT$^+$ processes a third (or less) of the total number of messages required by BnB-ADOPT (in some instances, messages are reduced by a factor of 8 or 9), and still reaches an optimal solution in almost the same number of cycles. In ternary instances, savings reach up to one order of magnitude in communication for almost all cases. Regarding the comparison with SBB and AFB, the new BnB-ADOPT$^+$ outperforms them (in number of messages and NCCCs) for low connected random instances, while the contrary occurs for highly connected ones. Regarding meeting scheduling and sensor network instances, BnB-ADOPT$^+$ outperforms SBB and AFB by a large margin. These results indicate that BnB-ADOPT$^+$ is a competitive algorithm for optimal DCOP solving.

3.2 ADOPT(k): Generalizing ADOPT and BnB-ADOPT search

As explained in Chapter 2, ADOPT and BnB-ADOPT algorithms share most of its data structures and communication protocol. Both algorithms send VALUE and COST messages to explore the search space, TERMINATE messages to end the execution and maintain lower and upper bounds of the current solution in every agent. An agent i calculates $LB_i(d)$ and $UB_i(d)$ as the lower and upper bounds of value $d \in D_i$ in the current context, while LB_i and UB_i are the minimum lower and upper bound for all values in D_i for the current context.

The main difference between these two algorithms lies on their search strategies. ADOPT employs a *best-first search* strategy while BnB-ADOPT employs a *depth-first branch-and-bound search* strategy. This difference in search strategies is reflected by how the agents change their values. While each agent i in ADOPT eagerly assigns the value that minimizes its lower bound $LB_i(d)$, each agent i in BnB-ADOPT changes its value only when it is able to determine that the optimal solution for that value is provably no better than the best solution found so far for its current context. In other words, BnB-ADOPT agent i changes value when $LB_i(d) \geq UB_i$ for its current value d.

The role of thresholds in the two algorithms is also different. As described earlier, each agent in ADOPT uses thresholds to store previously computed lower bounds for the current context. These thresholds are sent from parent to children, so children can reconstruct the partial solution found earlier more efficiently. This changes the condition for which an agent changes its value in ADOPT. Each agent i changes its value d only when $LB_i(d) \geq TH_i$. On the other hand, each agent in BnB-ADOPT uses thresholds to store the cost of the best solution found so far for all contexts and uses them to change value more efficiently. Therefore, each agent i changes its value d only when $LB_i(d) \geq \min\{TH_i, UB_i\}$.

BnB-ADOPT has several optimizations over ADOPT:

1. Agents in BnB-ADOPT process messages differently compared to agents in ADOPT. Each agent in ADOPT assigns a new value, if necessary, after each received message. On the other hand, each agent in BnB-ADOPT does so only after processing completely the message queue.

2. BnB-ADOPT includes thresholds in VALUE messages such that THRESHOLD messages are no longer required.

3. BnB-ADOPT includes a timestamp for each value in contexts such that their recency can be compared.[1]

In this Section we present a new algorithm, called ADOPT(k), that generalizes both ADOPT and BnB-ADOPT.

[1]The first two optimizations were in the implementation of ADOPT (Yin, 2008) but not in the publication (Modi et al., 2005).

3.2.1 Search Strategy

An agent in ADOPT always changes its value to the most promising value. This strategy causes that agents change value quite often as local information is updated with the arrival of new messages. As consequence of these frequent value changes, ADOPT agents have to repeatedly reconstruct partial solutions that they previously abandoned, which can be computationally inefficient. On the other hand, a BnB-ADOPT agent take values following a linear order. Once a value is assigned, an agent changes this value only when its optimal solution is provably no better than the best solution found so far, which can be computationally inefficient if the agent takes on bad values before good values. Therefore, we believe that there should be a good trade off between the two extremes, where an agent keeps its value longer than it otherwise would as an ADOPT agent and shorter than it otherwise would as a BnB-ADOPT agent.

With this idea in mind, we developed ADOPT(k), a new algorithm that generalizes ADOPT and BnB-ADOPT. Its behavior depends on parameter k. It behaves like ADOPT when $k = 1$, like BnB-ADOPT when $k = \infty$ and like a hybrid of ADOPT and BnB-ADOPT when $1 < k < \infty$. ADOPT(k) uses mostly identical data structures and messages as ADOPT and BnB-ADOPT. Each agent i in ADOPT(k) maintains two thresholds, TH_i^A and TH_i^B, which are the thresholds in ADOPT and BnB-ADOPT, respectively. They are initialized and updated in the same way as in ADOPT and BnB-ADOPT.

The main difference between ADOPT(k) and its predecessors is the condition by which an agent changes its value. Each agent i in ADOPT(k) changes its value d_i when:

$$LB_i(d_i) > TH_i^A + (k-1) \text{ or } LB_i(d_i) \geq \min\{TH_i^B, UB_i\}.$$

If $k = 1$, then the first condition reduces to $LB_i(d_i) > TH_i^A$, which is the condition for agents in ADOPT. The agents use the second condition, which remains unchanged, to determine if the optimal solution for their current value assignment is provably no better than the best solution found so far.

If $k = \infty$, then the first condition is always false and the second condition, which remains unchanged, is the condition for agents in BnB-ADOPT.

If $1 < k < \infty$, then each agent in ADOPT(k) keeps its current value assignment until the lower bound of that value is at least k units larger than the lower bound of the most promising value, at which point it takes on the most promising value.

3.2.2 Pseudocode

In Figures 3.6 and 3.7 we present the pseudocode of ADOPT(k), where x_i is a generic agent, C_i is its set of children, PC_i is its set of pseudo-children and SCP_i is the set of agents that are either ancestors of x_i or parent and pseudo-parents of either x_i or its descendants. The pseudocode uses the predicate `Compatible`(X,X') to determine if two contexts X and X' are compatible and the procedure `PriorityMerge`(X,X') to replace the values of agents in context X' with more recent values, if available, of the same agents in context X (see (Yeoh et al., 2010) for more details). The pseudocode is similar to ADOPT's pseudocode with the following changes:

- The pseudocode includes the optimizations described in Section 3.2 for BnB-ADOPT (message queue processing, removing THRESHOLD messages and including timestamps) that can also be applied to ADOPT [lines 03, 08-12, 35-36 and 40-41].
- In ADOPT, the `MaintainThresholdInvariant`, `MaintainChildThreshold-Invariant` and `MaintainAllocationInvariant` procedures are called after each message is processed. Here, they are called in the `Backtrack` procedure [lines 28 and 38-39]. The invariants are maintained only after all incoming messages are processed.
- In addition to TH_i^A, each agent maintains TH_i^B. It is initialized, propagated and used in the same way as in BnB-ADOPT (lines 21, 33-34, 40 and 59).
- The condition by which each agent i changes its value is now $LB_i(d_i) > TH_i^A + (k-1)$ or $LB_i(d_i) \geq \min\{TH_i^B, UB_i\}$ (lines 31 and 33). Thus, the agent keeps its value until the lower bound of that value is k units larger than the lower bound of the most promising value or the optimal solution for that value is provably no better than the best solution found so far.
- In ADOPT, the `MaintainAllocationInvariant` procedure ensures that the invariant $TH_i^A = \sum_{x_c \in C_i} TH_c^A$ always hold. This procedure assumes that $TH_i^A \geq LB_i(d_i)$ for the current value d_i of agent i, which is always true since the agent would change its value otherwise. However, this assumption is no longer true in ADOPT(k). Therefore, the pseudocode includes a new threshold $TH_i^A(d_i)$, which is set to TH_i^A and updated such that it satisfies the invariant $LB_i(d_i) \leq TH_i^A(d_i) \leq UB_i(d_i)$ in the `MaintainCurrentValueThreshold-Invariant` procedure (lines 84-89). This new threshold then replaces TH_i^A in the `Maintain-AllocationInvariant` procedure (lines 91 and 93).

```
01 procedure Start()
02   X_i := {(x_p, ValInit(x_p), 0) | x_p ∈ SCP_i};
03   ID_i := 0;
04   for each x_c ∈ C_i and d ∈ D_i InitChild(x_c, d);
05   InitSelf();
06   Backtrack();
07   loop forever
08     if message queue is not empty then
09       while message queue is not empty do
10         pop msg off message queue;
11         When Received(msg);
12       Backtrack();

13 procedure InitChild(x_c, d)
14   lb_i^c(d) := h_i^c(d);
15   ub_i^c(d) := ∞;
16   th_i^c(d) := lb_i^c(d);

17 procedure InitSelf()
18   d_i := arg min_{d∈D_i} {δ_i(d) + Σ_{x_c∈C_i} lb_i^c(d)};
19   ID_i := ID_i + 1;
20   TH_i^A := min_{d∈D_i} {δ_i(d) + Σ_{x_c∈C_i} lb_i^c(d)};
21   TH_i^B := ∞;

22 procedure procedure Backtrack()
23   for each d ∈ D_i do
24     LB_i(d) := δ_i(d) + Σ_{x_c∈C_i} lb_i^c(d);
25     UB_i(d) := δ_i(d) + Σ_{x_c∈C_i} ub_i^c(d);
26   LB_i := min_{d∈D_i} {LB_i(d)};
27   UB_i := min_{d∈D_i} {UB_i(d)};
28   MaintainThresholdInvariant();
29   if (TH_i^A = UB_i) then
30     d_i := arg min_{d∈D_i} {UB_i(d)}
31   else if (LB_i(d_i) > TH_i^A + (k − 1)) then
32     d_i := arg min_{d∈D_i|LB_i(d)=LB_i} {UB_i(d)}
33   else if (LB_i(d_i) ≥ min{TH_i^B, UB_i}) then
34     d_i := arg min_{d∈D_i|LB_i(d)=LB_i} {UB_i(d)}
35   if (a new d_i has been chosen) then
36     ID_i := ID_i + 1;
37   MaintainCurrentValueThresholdInvariant();
38   MaintainChildThresholdInvariant();
39   MaintainAllocationInvariant();
40   Send(VALUE, x_i, d_i, ID_i, th_i^c(d_i), min(TH_i^B, UB_i) − δ_i(d_i) − Σ_{x_c'∈C_i|x_c'≠x_c} lb_i^{c'}(d_i)) to each x_c ∈ C_i;
41   Send(VALUE, x_i, d_i, ID_i, ∞, ∞) to each x_c ∈ PC_i;
42   if (TH_i^A = UB_i) then
43     if (x_i is root or termination message received) then
44       Send(TERMINATE) to each x_c ∈ C_i;
45       terminate execution;
46   Send(COST, x_i, X_i, LB_i, UB_i) to parent;

47 procedure When Received(TERMINATE)
48   record termination message received;
```

Figure 3.6: Pseudocode of ADOPT(k) (1).

49 **procedure** `When Received`(VALUE, x_p, d_p, ID_p, TH_p^A, TH_p^B)
50 $X' := X_i$;
51 `PriorityMerge`((x_p, d_p, ID_p), X_i);
52 **if** (!`Compatible`(X', X_i)) **then**
53 **for each** $x_c \in C_i$ and $d \in D_i$ **then**
54 **if** ($x_p \in SCP_c$) **then**
55 `InitChild`(x_c, d);
56 `InitSelf`();
57 **if** (x_p is parent) **then**
58 $TH_i^A := TH_p^A$;
59 $TH_i^B := TH_p^B$;

60 **procedure** `When Received`(COST, x_c, X_c, LB_c, UB_c)
61 $X' := X_i$;
62 `PriorityMerge`(X_c, X_i);
63 **if** (!`Compatible`(X', X_i)) **then**
64 **for each** $x_c \in C_i$ and $d \in D_i$ **then**
65 **if** (!`Compatible`($\{(x_p, d_p, ID_p) \in X' \mid x_p \in SCP_c\}$,$X_i$)) **then**
66 `InitChild`(x_c, d);
67 **if** (`Compatible`(X_c, X_i)) **then**
68 $lb_i^c(d) := \max\{lb_i^c(d), LB_c\}$ for the unique $(a', d, ID) \in X_c$ with $a' = a$;
69 $ub_i^c(d) := \min\{ub_i^c(d), UB_c\}$ for the unique $(a', d, ID) \in X_c$ with $a' = a$;
70 **if** (!`Compatible`(X', X_i)) **then**
71 `InitSelf`();

72 **procedure** `MaintainChildThresholdInvariant`()
73 **for each** $x_c \in C_i$ and $d \in D_i$ **do**
74 **while** ($th_i^c(d) < lb_i^c(d)$) **do**
75 $th_i^c(d) := th_i^c(d) + \epsilon$;
76 **for each** $c \in C_i$ and $d \in D_i$ **do**
77 **while** ($th_i^c(d) > ub_i^c(d)$) **do**
78 $th_i^c(d) := th_i^c(d) - \epsilon$;

79 **procedure** `MaintainThresholdInvariant`()
80 **if** ($TH_i^A < LB_i$) **then**
81 $TH_i^A = LB_i$;
82 **if** ($TH_i^A > UB_i$) **then**
83 $TH_i^A = UB_i$;

84 **procedure** `MaintainCurrentValueThresholdInvariant`()
85 $TH_i^A(d_i) := TH_i^A$;
86 **if** ($TH_i^A(d_i) < LB_i(d_i)$) **then**
87 $TH_i^A(d_i) = LB_i(d_i)$;
88 **if** ($TH_i^A(d_i) > UB_i(d_i)$) **then**
89 $TH_i^A(d_i) = UB_i(d_i)$;

90 **procedure** `MaintainAllocationInvariant`()
91 **while** ($TH_i^A(d_i) > \delta_i(d_i) + \sum_{x_c \in C_i} th_i^c(d_i)$) **do**
92 $th_i^{c'}(d_i) := th_i^{c'}(d_i) + \epsilon$ for any $x_{c'} \in C_i$ with $ub_i^{c'}(d_i) > th_i^{c'}(d_i)$;
93 **while** ($TH_i^A(d_i) < \delta_i(d_i) + \sum_{x_c \in C_i} th_i^c(d_i)$) **do**
94 $th_i^{c'}(d_i) := th_i^{c'}(d_i) - \epsilon$ for any $x_{c'} \in C_i$ with $lb_i^{c'}(d_i) < th_i^{c'}(d_i)$;

Figure 3.7: Pseudocode of ADOPT(k) (2).

3.2.3 Correctness and Completeness

The proofs for the following lemmata and theorem closely follow those in (Modi et al., 2005; Yeoh et al., 2010). We relay in ADOPT(k) pseudocode lines and the optimum cost definitions:

$$OPT_i(d) = \delta_i(d) + \textstyle\sum_{x_c \in C_i} OPT_c \qquad \text{(Eq. 1)}$$
$$OPT_i = \min_{d \in D_i} OPT_i(d) \qquad \text{(Eq. 2)}$$

Lemma 4 *For all agents x_i and all values $d \in D_i$, $LB_i \leq OPT_i \leq UB_i$ and $LB_i(d) \leq OPT_i(d) \leq UB_i(d)$ at all times.*

Proof: We prove the lemma by induction on the depth of the agent in the pseudo-tree. It is clear that for each leaf agent i, $LB_i(d) = OPT_i(d) = UB_i(d)$ for all values $d \in D_i$ (Lines 24-25 and Eq. 1). Furthermore,

$$\begin{aligned}
LB_i &= \min_{d \in D_i} \{LB_i(d)\} && \text{(Line 26)} \\
&= \min_{d \in D_i} \{OPT_i(d)\} && \text{(see above)} \\
&= OPT_i && \text{(Eq. 2)} \\
UB_i &= \min_{d \in D_i} \{UB_i(d)\} && \text{(Line 27)} \\
&= \min_{d \in D_i} \{OPT_i(d)\} && \text{(see above)} \\
&= OPT_i && \text{(Eq. 2)}
\end{aligned}$$

So, the lemma holds for each leaf agent. Assume that it holds for all agents at depth q in the pseudo-tree. For all agents x_i at depth $q-1$,

$$\begin{aligned}
LB_i(d) &= \delta_i(d) + \sum_{x_c \in C_i} LB_c && \text{(Lines 24 and 68)} \\
&\leq \delta_i(d) + \sum_{x_c \in C_i} OPT_c && \text{(induction ass.)} \\
&= OPT_i(d) && \text{(Eq. 1)} \\
UB_i(d) &= \delta_i(d) + \sum_{x_c \in C_i} UB_c && \text{(Line 25 and 69)} \\
&\geq \delta_i(d) + \sum_{x_c \in C_i} OPT_c && \text{(induction ass.)} \\
&= OPT_i(d) && \text{(Eq. 1)}
\end{aligned}$$

for all values $d \in D_i$. The proof for $LB_i \leq OPT_i \leq UB_i$ is similar to the proof for the base case. Thus, the lemma holds. ■

Lemma 5 *For all agents x_i, if the current context of x_i is fixed, then $LB_i = TH_i^A = UB_i$ will eventually occur.*

Proof: We prove the lemma by induction on the depth of the agent in the pseudo-tree. The lemma holds for leaf agents x_i since $LB_i = UB_i$ (see proof for the base case of Lemma 4) and $LB_i \leq TH_i^A \leq UB_i$ (lines 79-83). Assume that the lemma holds for all agents at depth q in the pseudo-tree. For all agents x_i at depth $q-1$ with a fixed context,

$$\begin{aligned}
LB_i &= \min_{d \in D_i} \{\delta_i(d) + \sum_{x_c \in C_i} lb_i^c(d)\} && \text{(Lines 24 and 26)} \\
&= \min_{d \in D_i} \{\delta_i(d) + \sum_{x_c \in C_i} LB_c\} && \text{(Line 68)} \\
&= \min_{d \in D_i} \{\delta_i(d) + \sum_{x_c \in C_i} UB_c\} && \text{(induction ass.)} \\
&= \min_{d \in D_i} \{\delta_i(d) + \sum_{x_c \in C_i} ub_i^c(d)\} && \text{(Line 69)} \\
&= UB_i && \text{(Line 25 and 27)}
\end{aligned}$$

Additionally, $LB_i \leq TH_i^A \leq UB_i$ (Lines 79-83). Therefore, $LB_i = TH_i^A = UB_i$. ■

Lemma 6 *For all agents x_i, $TH_i^A(d) = TH_i^A$ on termination.*

Proof: Each agent i terminates when $TH_i^A = UB_i$ (Line 42). After $TH_i^A(d_i)$ is set to TH_i^A for the current value d_i of x_i (Line 85) in the last execution of the MaintainCurrentValueThresholdInvariant procedure,

$$\begin{aligned}
TH_i^A &= UB_i && \text{(Line 42)} \\
&= UB_i(d_i) && \text{(Lines 29-30)} \\
&\geq LB_i(d_i) && \text{(Lemma 4)}
\end{aligned}$$

Thus, $LB_i(d_i) \leq TH_i^A = UB_i$ and $TH_i^A(d_i)$ is not set to a different value later (Lines 86-89). Then, $TH_i^A(d) = TH_i^A$ on termination. ■

Theorem 5 *For all agents x_i, $TH_i^A = OPT_i$ on termination.*

Proof: We prove the theorem by induction on the depth of the agent in the pseudo-tree. The theorem holds for the root agent i since $TH_i^A = UB_i$ on termination (Lines 42-45), $TH_i^A = LB_i$ at all times (Lines 20 and 28), and $LB_i \leq OPT_i \leq UB_i$ (Lemma 4). Assume that the theorem holds for all agents at depth q in the pseudo-tree. We now prove that the theorem holds for all agents at depth $q+1$. Let x_p be an arbitrary agent at depth q in the pseudo-tree and d_p is its current value assignment on termination. Then,

$$\begin{aligned}
\sum_{x_c \in C_p} ub_p^c(d_p) &= UB_p(d_p) - \delta_p(d_p) && \text{(Line 25)} \\
&= UB_p - \delta_p(d_p) && \text{(Lines 29-30)} \\
&= TH_p^A - \delta_p(d_p) && \text{(Lines 42-45)} \\
&= TH_p^A(d_p) - \delta_p(d_p) && \text{(Lemma 6)} \\
&= \sum_{x_c \in C_p} th_p^c(d_p) && \text{(Lines 91-94)}
\end{aligned}$$

Thus, $\sum_{x_c \in C_p} ub_p^c(d_p) = \sum_{x_c \in C_p} th_p^c(d_p)$. Furthermore, for all agents $x_c \in C_p$, $th_p^c(d_p) \leq ub_p^c(d_p)$ (Lines 77-78). Combining the inequalities, we get $th_p^c(d_p) = ub_p^c(d_p)$. Additionally, $TH_c^A = th_p^c(d_p)$ (Lines 57-58) and $UB_c = ub_p^c(d_p)$ (Line 69). Therefore, $TH_c^A = UB_c$. Next,

$$\begin{aligned}
\sum_{x_c \in C_p} OPT_c &= OPT_p - \delta_p(d_p) && \text{(Eq. 1)} \\
&= TH_p^A - \delta_p(d_p) && \text{(induction ass.)} \\
&= TH_p^A(d_p) - \delta_p(d_p) && \text{(Lemma 6)} \\
&= \sum_{x_c \in C_p} th_p^c(d_p) && \text{(Lines 91-94)} \\
&= \sum_{x_c \in C_p} TH_c^A && \text{(Lines 57-58)}
\end{aligned}$$

Thus, $\sum_{x_c \in C_p} OPT_c = \sum_{x_c \in C_p} TH_c^A = \sum_{x_c \in C_p} UB_c$ (see above). Furthermore, for all agents $x_c \in C_p$, $OPT_c \leq UB_c$ (Lemma 4). Combining the inequalities, we get $OPT_c = UB_c$. Therefore, $TH_c^A = UB_c = OPT_c$. ■

3.2.4 Tie-breaking in BnB-ADOPT$^+$

When comparing ADOPT and BnB-ADOPT search strategies we noticed that ADOPT provides a tie-breaking mechanism, that is not present in BnB-ADOPT, when more than one value minimizes $LB(v)$. In this case, ADOPT agents choose the value that minimizes $UB(v)$. We introduced this tie-breaking strategy in ADOPT(k) (Figure 3.6, line 34). In BnB-ADOPT on the other hand, when more than one value minimizes $LB(v)$ agents choose the previous value, if possible (Yeoh et al., 2010).

Taking a closer look to ADOPT(k) behavior we noticed that, in case of having several values that minimize the LB (a tie), choosing the value that minimizes the UB can provide savings in search. Consider the following example: The $root$ agent i is executing a BnB-ADOPT search and has explored value a, b, c in that order and its current value assignment is c. During its execution, agent i has calculated the following bounds:

$$LB(a) = 10, UB(a) = 10$$
$$LB(b) = 10, UB(b) = \infty$$
$$LB(c) = 10, UB(c) = \infty$$

In this case, a is an optimal solution with cost 10, since $LB = UB$ and $LB(a) = UB(a)$. However, the BnB-ADOPT agent i might not jump directly to value a (the one that minimizes the UB) but to value b (the previous value). Only when $LB(b) > 10$ or $LB(b) = UB(b) = 10$ the agent jumps to value a. This behavior creates unnecessary computation in agent i and in other agents of the pseudo-tree. If we apply the tie-breaking mechanism, agent i would assign directly value a because is the one that minimizes $UB(v)$.

In the next Section we compare ADOPT(k) with BnB-ADOPT$^+$ and to better assess the impact of the type of search used, we also compare it with a version of BnB-ADOPT$^+$ that includes this tie-breaking strategy used in ADOPT and ADOPT(k).

3.2.5 Experimental Results

We compare ADOPT$^+$(k) to ADOPT$^+$ and BnB-ADOPT$^+$. ADOPT$^+$(k) is an optimized version of ADOPT(k) with the message reduction techniques explained in Section 3.1. In addition, we include a version of BnB-ADOPT$^+$ including the tie-breaking strategy explained in Section 3.2.4. All the algorithms use the DP2 heuristic values (Ali et al., 2005) calculated in a preprocess step. We report also, as a reference measure, the trivial upper bound *UB* calculated

in every instance as the sum of the maximums over cost functions. Tables 3.4, 3.5 and 3.6 shows the results. We omit ADOPT$^+$ in Tables 3.4 and 3.5 since BnB-ADOPT$^+$ performs better than ADOPT$^+$ across all metrics.

Table 3.4 shows the results on random binary DCOP instances with 10 variables of domain size 10. The costs are randomly chosen over the range $\langle 1000, \ldots, 2000 \rangle$. We impose $n(n-1)/2 * p_1$ cost functions, where n is the number of variables and p_1 is the network connectivity. We vary p_1 from 0.5 to 0.8 in 0.1 increments and average our results over 50 instances for each value of p_1. The table shows that ADOPT$^+(k)$ requires a large number of messages, cycles and NCCCs when k is small. These numbers decrease as k increases until a certain point where they increase again. For the best value of k, ADOPT$^+(k)$ performs significantly better than BnB-ADOPT$^+$ across all metrics and is slightly better than BnB-ADOPT$^+$ with tie-breaking in NCCCs.

p_1	Trivial *UB*	Algorithm	#Messages	# Cycles	# NCCC
0.5	45,766	BnB-ADOPT$^+$	262,812	23,646	5,353,423
		BnB-ADOPT$^+$ with tie-breaking	**197,028**	18,088	3,999,334
		ADOPT$^+(k=1{,}000)$	413,711	34,402	8,491,909
		ADOPT$^+(k=4{,}000)$	197,342	**17,100**	**3,969,104**
		ADOPT$^+(k=6{,}000)$	197,486	17,117	3,972,960
0.6	53,722	BnB-ADOPT$^+$	1,017,939	99,969	26,191,249
		BnB-ADOPT$^+$ with tie-breaking	**693,878**	66,567	16,897,706
		ADOPT$^+(k=1{,}000)$	1,864,165	160,365	45,101,673
		ADOPT$^+(k=4{,}500)$	701,374	**62,977**	**16,454,689**
		ADOPT$^+(k=6{,}000)$	701,529	62,994	16,459,453
0.7	63,654	BnB-ADOPT$^+$	3,716,766	387,744	116,050,941
		BnB-ADOPT$^+$ with tie-breaking	**2,486,103**	255,800	76,497,061
		ADOPT$^+(k=1{,}000)$	6,846,289	591,271	187,172,366
		ADOPT$^+(k=6{,}000)$	2,558,658	**241,102**	**71,495,744**
		ADOPT$^+(k=10{,}000)$	2,559,603	241,169	71,503,823
0.8	71,624	BnB-ADOPT$^+$	9,493,156	1,032,767	324,271,538
		BnB-ADOPT$^+$ with tie-breaking	**6,099,378**	657,104	206,828,853
		ADOPT$^+(k=5{,}000)$	10,911,176	1,056,531	339,276,384
		ADOPT$^+(k=10{,}000)$	6,395,945	**619,431**	**192,355,298**
		ADOPT$^+(k=20{,}000)$	6,484,296	628,362	195,216,439

Table 3.4: The number of messages, cycles and NCCCs of ADOPT$^+$, BnB-ADOPT$^+$ and ADOPT$^+(k)$ on Random Binary DCOP Instances (10 variables).

Table 3.5 shows the results on sensor network instances from a publicly available repository (Yin, 2008). We use instances from all four available topologies (cases A, B, C and D) and average our results over 30 instances for each topology. In relation with the k parameter, we observe the same trend as in Table 3.4 (large number of messages and cycles when k is small,

starting to decrease as as k increases until a certain point where they increase again). We only report the results for the best value of k, where ADOPT(k) is able to outperform BnB-ADOPT$^+$ significantly in all metrics and shows a moderate advantage with respect to BnB-ADOPT$^+$ with tie-breaking.

	Trivial *UB*	Algorithm	# Messages	# Cycles	# NCCC
A	15,234,868,488	BnB-ADOPT$^+$	5,090,410	228,784	43,595,024
		BnB-ADOPT$^+$ with tie-breaking	2,605,016	112,030	24,521,860
		ADOPT$^+$(k = 30,000,000)	**2,005,732**	**88,556**	**24,068,428**
B	15,355,044,866	BnB-ADOPT$^+$	23,911,475	1,024,435	249,771,051
		BnB-ADOPT$^+$ with tie-breaking	13,877,495	586,287	**147,834,791**
		ADOPT$^+$(k = 30,000,000)	**9,869,280**	**459,540**	166,542,715
C	3,997,096,838	BnB-ADOPT$^+$	311,738	17,571	3,386,651
		BnB-ADOPT$^+$ with tie-breaking	193,691	12,263	**2,241,200**
		ADOPT$^+$(k = 15,000,000)	**178,301**	**10,815**	2,625,136
D	15,595,397,524	BnB-ADOPT$^+$	10,722,499	575,613	156,019,351
		BnB-ADOPT$^+$ with tie-breaking	7,407,831	392,941	107,791,651
		ADOPT$^+$(k = 30,000,000)	**3,812,541**	**196,424**	**66,347,439**

Table 3.5: The number of messages, cycles and NCCCs of ADOPT$^+$, BnB-ADOPT$^+$ and ADOPT$^+$(k) on Sensor Network Instances (70, 70, 50 and 70 variables).

Lastly, Table 3.6 shows the results on sensor network instances of 100 variables arranged into a chain following the (Maheswaran et al., 2004) formulation. All the variables have a domain size of 10. The cost of hard constraints is 1,000,000. The cost of soft constraints is randomly chosen over the range $\langle 0, \ldots, 200 \rangle$. Additionally, we use discounted heuristic values, which we obtain by dividing the DP2 heuristic values by two, to simulate problems where well informed heuristics are not available due to privacy reasons. We average our results over 30 instances. The table shows that ADOPT$^+$ terminates with less cycles and computational effort than BnB-ADOPT$^+$ but sends more messages. When $k = 1$, the results for ADOPT$^+$(k) and ADOPT$^+$ are almost the same. Although performing the same search strategy, agents in ADOPT$^+$(k) sends more VALUE messages because they need to send VALUE messages when TH^B changes even if TH^A remains unchanged. These additional messages then trigger the need for more constraint checks. Agents in ADOPT$^+$ do not need to send VALUE messages in such cases. We observe that as k increases, the runtime of ADOPT$^+$(k) increases but the number of messages sent decreases. Therefore, ADOPT$^+$(k) provides a good mechanism in this case for balancing the trade-off between runtime and network load.

Trivial *UB*	Algorithm	# Messages	# Cycles	# NCCC
98,000,000	ADOPT$^+$	25,731	**259**	**10,840**
	BnB-ADOPT$^+$	4,808	1,753	36,913
	BnB-ADOPT with tie-breaking$^+$	**3,764**	827	38,704
	ADOPT$^+(k=1)$	25,915	259	12,381
	ADOPT$^+(k=30)$	22,092	449	20,591
	ADOPT$^+(k=50)$	10,502	550	25,196
	ADOPT$^+(k=100)$	6,290	550	25,196
	ADOPT$^+(k=1{,}000)$	5,050	827	38,441

Table 3.6: The number of messages, cycles and NCCCs of ADOPT$^+$, BnB-ADOPT$^+$ and ADOPT$^+(k)$ on Sensor Network Instances (100 variables).

3.3 Conclusions

In this Chapter we have presented two contributions to improve our reference algorithm for distributed search, namely BnB-ADOPT. First, we present theoretical results to detect redundant messages. Second, we describe some difficulties when dealing with n-ary cost functions and propose some modifications to overcome them. Combining these two contributions, we generate a new version of BnB-ADOPT. This new version, called BnB-ADOPT$^+$, causes substantial savings with respect to the original algorithm when tested on binary and n-ary benchmarks. We also show experimentally that BnB-ADOPT$^+$ is competitive with respect to other DCOP solving algorithms.

Taking a closer look to BnB-ADOPT and ADOPT algorithms, we noticed that ADOPT provides a tie-breaking strategy that when introduced in BnB-ADOPT$^+$ produces a significant boost in performance, so we include this modification in our version of BnB-ADOPT$^+$ as well.

Also, we present the new algorithm ADOPT(k), which combines the search strategies of ADOPT and BnB-ADOPT depending on the k parameter. Our experimental results show that ADOPT(k) can provide a good mechanism for balancing the trade-off between runtime and network load between ADOPT and BnB-ADOPT. Also, it was able to outperforms both algorithm on commonly used benchmarks for a certain k. It is direct to see that the changes indicated in Section 3.1.1 to remove redundant messages can also be applied to ADOPT(k). The version of ADOPT(k) that includes these changes is called ADOPT$^+(k)$.

In order to combine distributed search with soft arc consistency –the main goal of this thesis–, it would be ideal to work with an algorithm that provides lower and upper bounds of the problem solution. These bounds and their quality during execution are crucial to discover sub-optimal values. In the case of lower bounds, both ADOPT$^+(k)$ and BnB-ADOPT$^+$ provide

a mechanism to calculate lower bounds on every agent. In the case of upper bounds, the branch-and-bound search strategy of BnB-ADOPT$^+$ allows to refine a problem UB with every best solution found in the $root$ agent. In ADOPT$^+$(k) when $k \neq \infty$ (this is, using a different search strategy that BnB-ADOPT$^+$), the $root$ agent may jump from one domain value to another before completely exploring its current value d, so it might take longer to calculate $UB(d)$ or in the worst case, if d is not the optimal value, $UB(d)$ may remain always ∞. Then, no matter how high the lower bound of a value can be enhanced, this value could never be pruned using soft arc consistency. Also, the k parametrization of ADOPT$^+$(k) makes complex its evaluation if combining it with soft arc consistency, since we do not know how to decide automatically the best k. For all these reasons, for the rest of this thesis we use the BnB-ADOPT$^+$ algorithm as the distributed search algorithm to combine with soft arc consistency techniques.

4

Distributed Soft Arc Consistency

In the centralized case several techniques have been developed to speed up constraint optimization solving. In particular, search can be improved by enforcing soft arc consistency techniques. Several soft arc consistency levels have been proposed for constraint optimization problems (Cooper et al., 2008; de Givry et al., 2005; Larrosa and Schiex, 2003). By enforcing them it is possible to detect inconsistent values (this is, suboptimal values) and remove them from the problem, shrinking its search space. In practical terms, the effect is that the search tree is reduced and there are fewer nodes to explore. On the other hand, more computational work must be done per node. Globally, the overall effect is very beneficial, causing substantial savings in computational effort.

In this Chapter we include in distributed search for DCOP solving some techniques to enforce soft arc consistency. To the best of our knowledge, this is the first time that soft arc consistency is connected with distributed search. Such as it happens in the centralized case, this combination causes a drastic improvement in communication cost with also a substantial decrement in computation effort. Specifically, we we took as baseline the distributed search algorithm BnB-ADOPT$^+$ on top of which we maintain several consistency levels. Maintaining local consistencies keeps the optimality and termination of asynchronous distributed search.

4.1 Including Soft Local Consistencies in Distributed Problems

In Chapter 2 we have presented soft arc consistency techniques defined for constraint optimization problems in the centralized case. Soft arc consistencies are conceptually equal in the centralized and distributed cases. However, maintaining soft arc consistencies in a distributed environment requires different techniques. While in the centralize case all problem elements are available to the single agent performing the search, in the distributed case agents only know some parts of the problem. For example agents know about cost functions in which they are involved, but they do not know about costs functions that other agents share. This fact brings some challenges in distributed, specially for those consistency levels that need to have a wider scope of the global problem.

In the following, we summarize the main differences between the centralized and distributed cases regarding soft arc consistency techniques:

- *Pruning condition.* In the centralized case, a value $a \in D_i$ can be removed if it is not NC, that is, if $C_i(a) + C_\phi \geq \top$. However, in the distributed case a is removed only if $C_i(a) + C_\phi > \top$, as explained in the following. In both cases, $\top$ is an upper bound ($\geq$) of the optimum cost. In the distributed case, it is expected that agents terminate assigning the optimal value. So the optimal solution is configured among the agents, when all solutions have been explored, without any centralized control. If we prune a value when its cost equals $\top$, we might remove a value that belongs to an optimal solution, so agents could not end execution with the optimal assignment. (An example appears in Figure 4.1) For this reason, we only prune when the value cost exceeds $\top$. In the centralized case, the only agent executing the solving procedure stores the complete "best solution" found as search progresses; so a value of the optimal solution can be pruned from its domain, because that solution was stored somewhere; when the algorithm terminates, that solution will be recalled. However in distributed we do not have this centralized storage.

- *Representation of cost functions.* In the centralized case, all cost functions are known and manipulated by a single agent, the one in charge of the problem solving. This agent keeps a single copy of each cost function, where all modifications (extensions, projections) are performed. In the distributed case, a cost function C_{ij} between agents i and j is known by both agents, and each one keeps a local copy of C_{ij}. Initially, they share

the same copy, but operations to maintain soft arc consistency modify C_{ij}. Since each agent performs these operations independently, after a while agents could end up with a different representation of C_{ij}. Therefore soft arc consistency operation must be done in such a way that both agents maintain a *legal representation* of C_{ij}, otherwise the same cost can be counted twice when projecting unary costs to C_ϕ. This is depicted in Figure 4.2, where an illegal increment in C_ϕ is performed, causing $C_i(a) + C_\phi$ to become an invalid lower bound for value a. To maintain a legal representation of cost functions, agent i has to simulate the action of agent j on its C_{ij} copy, and vice versa. In some cases, i has also to send a message to j.

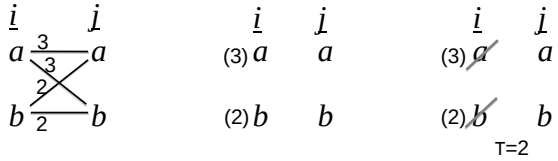


Figure 4.1: Left: Simple example with two agents i and j and two values per variable. Lines represent binary costs and values in parenthesis unary costs. The optimum cost is 2 and there are two optimal solutions $(i,b)(j,a)$ and $(i,b)(j,b)$. Center: A projection is performed from binary C_{ij} to unary costs C_i. Right: if $\top = 2$, pruning v with $cost(v) = \top$ causes to lose value (i,b) which is part of the two optimal solutions. In fact, no value would remain for i.

In general, when information is distributed agents are required to exchange messages in order to achieve the desired consistency level. In a naive approach, each time an agent needs some information of other agent it would generate two messages (request and response) which could cause a serious degradation in performance. In our approach, we try to keep the number of exchanged messages as low as possible. Using messages to coordinate, agents perform operations (projections, extensions) that modify cost functions. We perform these operations in such a way that bounds are always computed in a legal way (no costs are counted twice and no invalid lower bounds are produced).

Also, privacy issues must be taken into account. In many cases it is not desirable that agents share their preferences (unary costs) with other agents. We tried to maintain this assumption and only partial information about unary cost functions is revealed to other agents. In the distributed case, it is usually assumed that each agent knows about (i) its variable and (ii) the cost functions it has with other agents. Assumption (ii) implies that it also knows about the

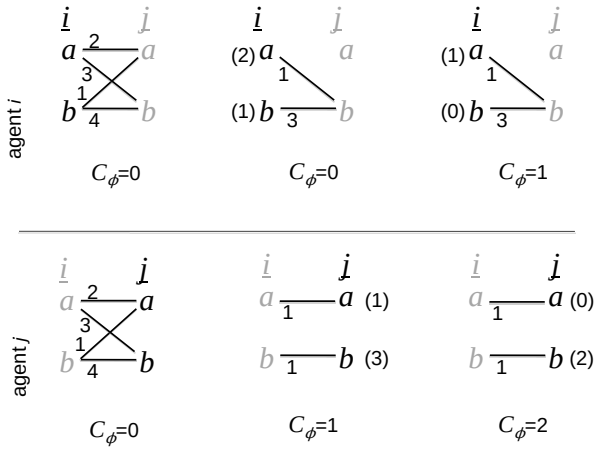


Figure 4.2: Simple example with two agents i and j and two values per variable. Lines represent binary costs and values in parenthesis unary costs. First row (UP) contains agent i representation of the problem. Second row (DOWN) contains agent j representation of the problem.
(UP and DOWN) Left: Initially, both representations are the same.
(UP) Center: i projects from C_{ij} to its unary costs. Right: i projects its unary costs to C_ϕ.
(DOWN) Center: j projects from C_{ij} to its unary costs, without considering previous projections of i (this is incorrect). Right: j projects its unary costs to C_ϕ, causing an incorrect increment.

domain of neighbors. To enforce any soft arc consistency, we explicitly require that if agent i is connected with agent j by C_{ij}, i has to represent locally D_j. For privacy reasons, we assume that the unary costs of the values of an agent are held by itself. So an agent neither can know nor update unary costs of other agents.

One of the main goals of soft arc consistency is to construct strong lower bounds: C_ϕ is a lower bound of the optimal solution; unary cost $C_i(v)+C_\phi$ is a lower bound of domain value v. Lower bounds are useful to identify sub-optimal values when $C_i(v)+C_\phi > \top$. In addition, they can provide a heuristic for value selection which may improve search efficiency. In (Ali et al., 2005) authors propose a preprocessing technique for the ADOPT algorithm, and show that by calculating lower bounds for every domain value they are able to speed up ADOPT search, since these lower bounds provide a good heuristic for value selection during search. In (Matsui et al., 2009) authors transform the original problem into an equivalent one projecting costs in a preprocessing step, then ADOPT is executed in the equivalent problem with performance improvements. In these two approaches authors compute lower bounds for values and use them during ADOPT search, but no deletions are performed, so soft arc consistency is not fully

enforced, since values in the problem may remain not NC.

In the following we present the combination of the distributed search algorithm BnB-ADOPT$^+$ with several soft arc consistency levels. We assure NC by deleting inconsistent values. Deletions are a key point when enforcing soft arc consistency since they lead to refinements in the lower bounds and further deletions. If a value deletion is propagated to neighbors new inconsistent values may be discovered (which can also be propagated leading to further deletions, etc). In our approach, inconsistent values are removed during preprocessing and also during search.

4.2 Unconditional Deletions in BnB-ADOPT$^+$

As explained in Chapter 2, when enforcing soft arc consistency on a DCOP problem we can remove inconsistent values (if $C_i(v) + C_\phi > \top$). These values are removed *unconditionally*, this means that they are never needed to be restored again. Such unconditional value deletions can be done in a preprocessing step, before the search process begins.

During search, an agent may remove values *conditionally* to particular value assignments of ancestors, in the following way. When an agent i assigns a value, the domain of i is reduced to this single value. Then, if some value v is proved to be inconsistent in a neighbor j considering this context it can be removed. However when i change its value assignment (this produces a $context$ change) the deleted value v must be restored in j. Propagation of conditionally deleted values is expensive because it is not permanent and requires a high communication effort to delete and restore values. This strategy has not paid off in Distributed Constraint Satisfaction Problems (Brito and Meseguer, 2008). Because of this, for the distributed optimization case we focus on *unconditional* deletions, where values do not depend on any particular assignments of ancestors.

In addition to inconsistent values, during BnB-ADOPT$^+$ search an agent may remove values *unconditionally*, assuring that they will not be used in the optimal solution, as explained next. Let us consider a DCOP instance, where agents are arranged in a pseudo-tree and each agent executes BnB-ADOPT$^+$. The $root$ agent updates and propagates to other agents the $\top$ value (initially ∞) with the best solution found so far. Let us consider a generic agent *self* that takes value v. After sending VALUE messages, *self* receives COST messages from its children. A COST message contains the lower bound computed by children given the context contained in the COST message. We consider COST messages whose context is simply the

assignment $(self, v)$, the $self$ agent with current value assignment v. Observe that for the $root$ agent such COST messages are always received. If the sum of the lower bounds of these COST messages (sent with a context containing only the *self* agent) exceeds $\top$, then v can be deleted unconditionally. To see this, it is enough to realize that the lower bound is computed assuming a context: if this context is simply $(self, v)$, the actual cost of v does not depend on the value assignment of any ancestor, so if it exceeds $\top$ it can be deleted permanently. This reasoning is valid for any agent of the pseudo-tree.

Unconditional deletions do not depend on values assignments of other agents and can be propagated causing new deletions. Any deletion caused by propagation of unconditional deletions is also unconditional. To effectively propagate deletions to other agents some kind of soft arc consistency during search must be maintain, as explain next.

4.3 BnB-ADOPT$^+$ Combined with AC and FDAC

We present in this Section the connection of BnB-ADOPT$^+$ with AC and FDAC consistency levels. Some soft arc consistencies require agents to be ordered. We assume this order as the order of agents in each branch of the pseudo-tree used by BnB-ADOPT$^+$. Observe that, although it is not a total order, agents in separate branches do not share cost functions, so for enforcing soft arc consistency it is enough with the partial ordering of agents in the pseudo-tree.

For simplicity, we assume that cost functions are unary and binary only. We also assume, for the moment, that simultaneous deletions do not occur on neighboring agents. This specific case is detailed in Section 4.3.4.

Soft arc consistency is maintained during distributed search in a preprocess step and also during search every time soft arc consistency is broken. AC/FDAC levels are achieved implementing the projection and extension operators for the distributed case, and pruning node inconsistent values on every agent. These operations are performed asynchronously on every agent and flows of costs can be extended in parallel across all branches of the pseudo-tree.

The proposed combination of BnB-ADOPT$^+$ with AC and FDAC keeps the optimality and termination properties of BnB-ADOPT (Yeoh et al., 2010) as follows. While BnB-ADOPT$^+$ search is performed using the cost functions to calculate costs, projections and extensions to maintain AC/FDAC are performed on a copy of these cost functions. In this way, the search process is not altered with any modification in the original cost functions. Modifications comes from the fact that inconsistent values are removed from the domain of agents. If we only remove

node inconsistent values (suboptimal values) –which is assured by the AC/FDAC properties– then it is easy to see that the optimality and termination of the algorithms is guaranteed.

4.3.1 BnB-ADOPT$^+$-AC

BnB-ADOPT$^+$-AC algorithm combines distributed branch-and-bound search with AC consistency level. Its search process is based on BnB-ADOPT$^+$, maintaining the same data and communication structure. The main difference is that agents are able to detect and delete suboptimal values. The inclusion of AC in BnB-ADOPT$^+$ have caused a number of modifications in the original algorithm, both in the structure of the exchanged messages and in the computation done.

Regarding memory, the domain of neighbors is represented in each agent. Each agent has a local copy of C_ϕ and $\top$. Also, two copies of the same cost functions are kept on every agent: the first one to calculate costs during search (this copy remains unchanged) and the second one to perform the projections needed for AC enforcement.

Regarding messages, new information needed to maintain AC are included in messages, as shown in Figure 4.3:

BnB-ADOPT$^+$ messages:
VALUE($sender$, $destination$, $value$, $threshold$)
COST($sender$, $destination$, $context[]$, lb, ub)
STOP($sender$, $destination$)

BnB-ADOPT$^+$-AC messages:
VALUE($sender$, $destination$, $value$, $threshold$, $\top$, C_ϕ)
COST($sender$, $destination$, $context[]$, lb, ub, $projectionsToC_\phi$)
STOP($sender$, $destination$, $emptydomain$)
DEL($sender$, $destination$, $deletedValues$)

Figure 4.3: Messages of BnB-ADOPT$^+$ and BnB-ADOPT$^+$-AC.

- A new DEL message is introduced to inform deletions to neighbors. With the new message DEL($i; j; deletedValues$), i informs neighbor j that it has deleted the value(s) in the list $deletedValues$. When received, j updates its D_i copy and recheck the AC property on its values, which may lead to further deletions.

- VALUE messages include $\top$ and C_ϕ. Both values are calculated at the $root$ agent of the pseudo-tree and propagated from root to leafs. $\top$ is initially ∞ or a user specified value and is refined during search with the best solution found so far.
- COST messages include the variable $projectionsToC_\phi$ that aggregates the costs of unary projections to C_ϕ made by agents of the pseudo-tree. Each agent adds its own unary cost projections to C_ϕ with the projections of all its children. In this way, $projectionsToC_\phi$ aggregates the projections of all the agents of the subtree rooted at the $sender$ agent.

Regarding computation, domain values are tested for deletion and cost functions are projected (binary into unary, unary into C_ϕ). A value d is proved sub-optimal and can be deleted unconditionally from the domain of x_i if $C_i(d) + C_\phi > \top$. Only the agent owner of a variable can delete values in its domain.

When performing projections in two constrained agents i and j, changes on C_{ij} should be done carefully since i and j operate asynchronously and after a while they might end up with different copies of the cost functions. There is a pre-established ordering for projections. If $i < j$ in the pseudo-tree, AC is maintained in the following order: first projections are done from j to i and then from i to j, as shown in Figure 4.4. This is done in both agents to avoid counting twice the same cost, which may lead to delete optimal values. When an agent *self* performs a projection from a neighbor to *self*, it updates its binary and unary costs. When an agent *self* performs a projection from *self* to a neighbor agent, it updates its binary costs but not the unary costs of neighbors.

When a deletion occurs the AC property may be broken (supports may be lost on neighbors). Therefore when a deletion occurs on an agent i it sends DEL messages to neighbors and projects binary costs over neighbors; when these DEL messages are received, neighbors project binary costs over themselves.

During preprocess, projections are performed following the established order and if inconsistent values are found they are deleted. During search, $\top$ and C_ϕ values are aggregated and propagated and new deletions and projections might be performed. Pseudocode of the preprocess phase appears in Figure 4.5 and 4.6, a quick description follows:

- `Preprocess`. It initializes the problem variables and performs `AC`. Then, it processes DEL and STOP messages from the message queue until there are no more messages ($quiescence$ is $true$) or until an empty domain has been detected in some agent (end is $true$). An empty domain means that the problem has no acceptable solution.

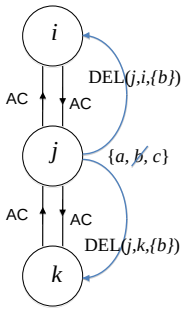


Figure 4.4: Three agents i, j, k in the same branch of the pseudo-tree. Maintaining AC: Cost functions are AC in both senses; deleting value b in D_j causes to send two DEL messages to i and k to propagate deletions.

- `AC`. Binary projections are performed between $self$ and each neighbor constrained with $self$, following the established ordering: costs are first projected to the higher agent and after to the lower agent. Notice that executing the procedure `BinaryProjection`$(i, self)$ does not change the unary costs of $self$ but modifies its copy of $C_{i,self}$.

- `BinaryProjection`(i, j). Binary projections are performed from i to j as follows: binary cost function C_{ij} is decremented with the minimum cost α (lines 19-22); unary cost function C_i is incremented with the minimum cost α only if the projection is done over the *self* agent (line 23).

- `UnaryProjectionToC`$_\phi$. Projects the minimum unary cost of $self$ to the variable $myProjectionC_\phi$, which accumulates all $self$ projections to C_ϕ.

- `CheckDomainForDeletions`. Checks every domain value for deletion. A value v is proved sub-optimal under certain conditions: when its unary cost plus C_ϕ exceeds $\top$ (line 30) or when the sum of its lower bounds exceeds $\top$ and this bounds were sent with a context that contains only the *self* agent (line 31, the bounds do not depend on any higher agent, as explained in Section 4.2). After identifying sub-optimal values they are deleted (line 33). Then, if the domain of *self* becomes empty, it means that no optimal solution can be found (the problem has no acceptable solution) and *self* sends $STOP$ messages to all neighbors with $emptydomain = true$. If *self* domain is not empty, it sends a DEL message to every neighbor j informing about its deleted value(s) and

```
01 procedure Preprocess()
02   initialize ⊤, C_φ, lb, ub, contexts, domains;
03   AC();
04   while ¬end ∧ ¬quiescence do
05     msg ← getMessageFromQueue();
06     switch(msg.type)
07       DEL: ProcessDelete(msg);    STOP: ProcessStop(msg);

08   procedure AC()
09     for each i ∈ neigbors(self) do
10       if i < self then
11         BinaryProjection(self, i);
12         BinaryProjection(i, self);
13       else
14         BinaryProjection(i, self);
15         BinaryProjection(self, i);
16     CheckDomainForDeletions();
17     UnaryProjectionToC_φ();

18   procedure BinaryProjection(i, j)
19     for each a ∈ D_i do
20       α ← min_{b∈D_j}{C_ij(a, b)};
21       for each b ∈ D_j do
22         C_ij(a, b) ← C_ij(a, b) − α;
23       if i = self then C_i(a) ← C_i(a) + α;

24   procedure UnaryProjectionToC_φ()
25     α ← min_{a∈D_self}{C_self(a)};
26     myProjectionC_φ ← myProjectionC_φ + α;
27     for each a ∈ D_self do C_self(a) ← C_self(a) − α;

28   procedure CheckDomainForDeletions()
29     for each v ∈ D_self do
30       if C_self(v) + C_φ > ⊤ then deleteValues.add(v);
31       if Σ_{c∈children st.contexts(c,v)={self}} lb(c, v) > ⊤ then deleteValues.add(v);
32     if deleteValues.size > 0 then
33       for each v ∈ deleteValues do D_self ← D_self − {v};
34       if D_self = ∅ then
35         for each j ∈ neighbors(self) do sendMsg(STOP, self, j, true); end ← true;
36       else
37         for each j ∈ neighbors(self) do
38           sendMsg(DEL, self, j, deleteValues);
39           BinaryProjection(j, self);
```

Figure 4.5: Pseudocode of BnB-ADOPT$^+$-AC Preprocess. (1)

```
40  procedure ProcessDelete()
41    for each v ∈ msg.deletedValues do
42      domains(sender) ← domains(sender) − {v};
43      BinaryProjection(self, sender);

44  procedure ProcessStop(msg)
45    end ← true;
46    if (msg.emptyDomain) then
47      for each j ∈ neighbors(self), j ≠ sender do
48        sendMsg(STOP, self, j, true);
```

Figure 4.6: Pseudocode of BnB-ADOPT$^+$-AC Preprocess. (2)

perform a binary projection from *self* to j, since a support may be lost in j. When the DEL message arrives, a projection in the same direction is performed in neighbor j, as explain next.

- `ProcessDelete`(msg). *self* received a DEL message, which means that $sender$ has deleted some value(s) from its domain. *self* registers these deletions in its D_{sender} copy and performs a binary projection from $sender$ to *self*.

- `ProcessStop`. *self* received a STOP message. If caused by an empty domain (no acceptable solution exists), *self* resends STOP messages to all its neighbors, except $sender$. In any case, *self* records its reception in end.

During the search phase, BnB-ADOPT$^+$-AC includes the following actions to maintain AC:

- When *self* receives a VALUE message, the local copies of $\top$ and C_ϕ are updated if the values contained in the received message are better (lower for $\top$, higher for C_ϕ).

- When *self* receives a COST message from a child c, *self* records c subtree projections to C_ϕ. When a new COST message is sent, the subtree projections of all children are added to *self* own projections to C_ϕ and this is sent to the parent agent.

- In the `Backtrack` procedure, after processing completely the message queue, *self* calls the `CheckDomainForDeletions` and `UnaryProjectionToC`$_\phi$ procedures. During VALUE, COST and DEL message processing unary cost functions, $\top$ and C_ϕ may

have been updated, so *self* tries to find new sub-optimal values to delete and new projections to C_ϕ.

Notice that agents could check their domain looking for sub-optimal values every time $\top$ or C_ϕ are updated (during reception of VALUE or COST messages), since with every refinement in $\top$ or C_ϕ a new opportunity for value deletion might appear. However during search we check for deletions after the agent has completely processed the message queue. Such as it happens when agents decide to change value, they will first gather all information from incoming messages (all possible refinements to $\top$ and C_ϕ) before checking for sub-optimal values. With this, computational effort is saved.

In the same way, unary projections to C_ϕ could be done every time there is a possibility to increment the agents local contribution to C_ϕ. This can happen when a value is deleted in *self* or when binary projections are performed over *self* leading to an increment in its unary costs. However, we perform this operation after the agent has completely processed the message queue. We do this for the following reason. Every time there is a unary projection in *self* its unary costs are decremented. In the centralized case, this decrement is quickly compensated with an increment in the global C_ϕ, but in a distributed environment this compensation is not immediate, it takes some time. First *self* projection must travel in COST messages to the $root$ agent, where is aggregated with other projections. Afterwards the aggregated C_ϕ is informed in VALUE messages to lower neighbors. This process can take some cycles, so we delay the decrement of unary costs in agents until the next COST message is sent (in the `Backtrack` procedure).

We have taken the strategy of delaying the projections from unary costs to C_ϕ and checking for deletions until the agent reads completely the message queue. Experimentally, we have observed that this strategy reduces considerably the computational effort made by the algorithm, although in some cases it may cause a few extra messages since these operations are not done as early as they could be.

4.3.2 BnB-ADOPT$^+$-FDAC

BnB-ADOPT$^+$-FDAC algorithm performs distributed search and maintains FDAC level of soft arc consistency. Extensions and projections are performed following the following order: if i and j are two neighbor agents, $i < j$ in the pseudo-tree, DAC is maintained from j to i and AC

from i to j. Therefore costs are extended always to higher agents in the pseudo-tree, as shown in Figure 4.7.

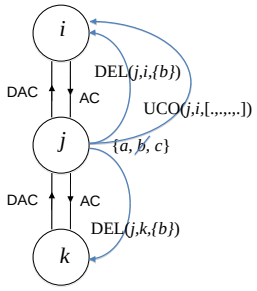


Figure 4.7: Three agents i, j, k in the same branch of the pseudo-tree. Maintaining FDAC: Cost functions are FDAC (DAC in one sense and AC in the other); deleting value b in D_j causes to send two DEL messages to i and k, plus one UCO message to the higher agent i if costs can be extended to i.

As in the BnB-ADOPT$^+$-AC algorithm, two copies of the cost functions are kept on every agent: one copy is modified with projections and extensions needed to achieve FDAC while the other copy remains unchanged and is used to calculate costs during search. In addition to the messages required for BnB-ADOPT$^+$-AC, a new UCO (unary costs) message is added to inform of cost extensions (Figure 4.8). When agent j extends costs to a higher agent i, j sends a UCO message to i with the unary costs extended (extended costs are the maximum unary costs in j that can be extended and after projected to i, following (Larrosa and Schiex, 2003)), as depicted in Figure 4.7. Upon reception, agent i performs the extension of these unary costs into its binary cost function C_{ij} and the projection of binary costs in C_{ij} to its unary cost function C_i.

During preprocess, projections and extensions are performed following the established order and if inconsistent values are found they are deleted. During search, $\top$ and C_ϕ values are aggregated and propagated and new deletions and projections/extensions might be performed. Pseudocode is depicted in Figure 4.9. The BnB-ADOPT$^+$-FDAC process is the same as BnB-ADOPT$^+$-AC code, plus the reception and process of the new UCO message to inform cost extensions. A summary description of this new procedures follows:

- `Preprocess`. Includes a call to the `FDAC` procedure and processes the new UCO message.

BnB-ADOPT$^+$-AC messages:
VALUE($sender$, $destination$, $value$, $threshold$, $\top$, C_ϕ)
COST($sender$, $destination$, $context[]$, lb, ub, $projectionsToC_\phi$)
STOP($sender$, $destination$, $emptydomain$)
DEL($sender$, $destination$, $deletedValues$)

BnB-ADOPT$^+$-FDAC messages: those of BnB-ADOPT$^+$-AC plus
UCO($sender$, $destination$, $vectorOfExtensions$)

Figure 4.8: Messages of BnB-ADOPT$^+$-AC and BnB-ADOPT$^+$-FDAC.

```
01 procedure Preprocess()
02   initialize ⊤, C_φ, lb, ub, contexts, domains;
03     FDAC();
04   while ¬end ∧ ¬quiescence do
05     msg ← getMessageFromQueue();
06     switch(msg.type)
07       DEL: ProcessDelete(msg);   UCO: ProcessUnaryCost(msg);   STOP: ProcessStop(msg);

08 procedure FDAC()
09   AC();
10   for each i ∈ neigbors(self) do
11     if i < self then ExtendCostsToNeighbor(i);

12 procedure ExtendCostsToNeighbor(i)
13   for each a ∈ D_i do P[a] ← min_{b∈D_self} {C_{i,self}(a, b) + C_self(b)};
14   for each b ∈ D_self do E[b] ← max_{a∈D_i} {P[a] − C_{i,self}(a, b)};
15   if E has some cost greater than 0 then
16     UnaryExtension(i, E);
17     BinaryProjection(i, self);
18     sendMsg(UCO, self, i, E);

19 procedure UnaryExtension(i, E)
20   for each b ∈ D_self do
21     for each a ∈ D_i do C_{i,self}(a, b) ← C_{i,self}(a, b) + E[b];
22     C_self(b) ← C_self(b) − E[b];

23 procedure ProcessUnaryCosts(msg)
24   for each b ∈ D_sender do
25     for each a ∈ D_self do C_{self,sender}(a, b) ← C_{self,sender}(a, b) + msg.vectorOfExtensions(b);
26   FromBinaryToUnary(self, sender);
```

Figure 4.9: Pseudocode of BnB-ADOPT$^+$-FDAC Preprocess. Missing procedures appear in Figure 4.5 and 4.6 (AC Preprocess)

- `FDAC`. After maintaining AC, costs are extended to every higher neighbor i calling the procedure `ExtendCostsToNeighbor`(i).

- `ExtendCostsToNeighbor`(i). ***self*** extends costs to agent i. The vector of extensions E is calculated, which contains the extensions for every domain value in D_{self}. If at least some of these extensions is greater than zero, these costs are extended from C_{self} to $C_{self,i}$ (procedure `UnaryExtension`(i, E)). After this, binary costs are projected from ***self*** to i and a UCO message is sent to agent i informing of the cost extensions. The vector E is sent in this COST message ($vectorOfExtensions$).

- `UnaryExtension`(i, E) Costs in the vector of extensions E are extended from C_{self} to $C_{self,i}$.

- `ProcessUnaryCost`. ***self*** has received a UCO message. Costs are extended from the vector of extensions included in the UCO message to $C_{self,sender}$. After this, costs are projected from binary costs $C_{self,sender}$ to unary costs C_{self}.

In addition to these new procedures, the following changes must be done in the algorithm:

- `CheckDomainForDeletions`. If ***self*** deletes a value, a flag $extendCosts$ is set to $true$.

- `BinaryProjection`(i, j). If unary costs are incremented in ***self***, a flag $extendCosts$ is set to $true$.

- `Backtrack`. After ***self*** checks its domain for deletions and projects unary costs to C_ϕ, if the flag $extendCosts$ is set to $true$, the agent tries to extend costs to all higher neighbors with the procedure `ExtendCostsToNeighbor`(i).

4.3.3 Example

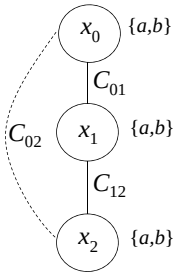
In this Section we present an example to illustrate savings obtained as result of maintaining soft arc consistency in distributed problems during search. Consider variables x_0, x_1 and x_2 ($x_0 < x_1 < x_2$ in the pseudo-tree) with domain $\{a, b\}$ and cost functions as represented in Figure 4.10 (Up). Enforcing AC on this problem (projecting binary costs into unary costs, projecting unary costs over C_ϕ and deleting inconsistent values) we obtain the equivalent problem of Figure 4.10 (Down, Right). In this problem C_ϕ is 20 and inconsistent values $x_1 = a$ and $x_2 = b$ has been

deleted. In the following we show how the AC consistency level is achieved in the distributed problem.

During BnB-ADOPT$^+$-AC execution, the unary projections to C_ϕ of all agents are aggregated in the root of the pseudo-tree (agent x_0) using COST messages and C_ϕ is propagated to agents x_1 and x_2 using VALUE messages. We present the execution trace of BnB-ADOPT$^+$ (Table 4.1, Left) and BnB-ADOPT$^+$-AC (Table 4.1, Right). As shown, BnB-ADOPT$^+$ needs 34 messages and 15 cycles to reach an optimal solution, while BnB-ADOPT$^+$-AC needs only 20 messages and 7 cycles. In the following we explain briefly this decrement in messages and cycles. For a clearer explanation, we omit some messages of BnB-ADOPT$^+$ execution that makes explanation more complex and are not relevant to show the benefits of BnB-ADOPT$^+$-AC.

In BnB-ADOPT$^+$ execution, all agents are initialized with value a and they send the correspondent VALUE messages (Figure 4.11(a)). After receiving the VALUE message from x_0, agent x_1 also receives the correspondent COST message from its child (Figure 4.11(b)). Then, x_1 tries value b, and receives the correspondent COST message from its child (Figure 4.11(c) and (d)). Now, all values of x_1 has been explored and x_1 chooses value b as the best value for the current context, and sends a COST message to its parent with UB=LB=20 (Figure 4.11(d)). When x_0 receives this COST message it changes to value b and sends VALUE messages to x_1 and x_2 (Figure 4.11(e)). When x_1 and x_2 receive the VALUE message, they reinitialize their information (because context has changed), and they start exploring their values under the current context (lines 20-29 of BnB-ADOPT$^+$ execution trace). Agents x_2 and x_1 exchange messages until all values are explored and x_1 send a COST message to its parent with UB=LB=25 (Figure 4.11(f)). When x_0 receives this COST, as $UB(b) > UB(a)$, and all values has been explored, x_0 changes its value to a and terminates (Figure 4.11(g)). With this new context change, x_1 and x_2 reinitialize values and exchange messages until they also terminate with the optimal solution (Figures 4.11(h) and (i)). Lines in the bottom of each subfigure in Figure 4.11 indicate the line that they represent in the execution trace.

In BnB-ADOPT$^+$-AC execution, a $\top$ of 30 is propagated in value messages. When $\top$ arrives to agent x_1 it is able to delete value a (line 6 of execution trace). Observe that at this point C_ϕ has not been yet aggregated and is equal to zero, however $C_1(a) = 36$ so $C_1(a) + C_\phi > \top$ ($36+0 > 30$). When this deletion is propagated to agent x_2 (line 8 of execution trace) x_2 reinforces AC, as result its unary cost $C_2(b) = 35$. After this, x_2 also receives the partially aggregated $C_\phi = 15$ and is able to delete value b since $C_2(b) + C_\phi > \top$ ($36 + 15 > 30$). In



C_{01} :

	$x_0 = a$	$x_0 = b$
$x_1 = a$	9	2
$x_1 = b$	4	1

C_{02} :

	$x_0 = a$	$x_0 = b$
$x_2 = a$	1	9
$x_2 = b$	4	5

C_{12} :

	$x_1 = a$	$x_1 = b$
$x_2 = a$	50	15
$x_2 = b$	50	50

$C_0(a) = C_1(a) = C_2(a) = 0$
$C_0(b) = C_1(b) = C_2(b) = 0$
$C_\phi = 0, \top = 30$

C_{01} :

	$x_0 = a$	$x_0 = b$
$x_1 = a$	4	0
$x_1 = b$	0	0

C_{02} :

	$x_0 = a$	$x_0 = b$
$x_2 = a$	0	4
$x_2 = b$	3	0

C_{12} :

	$x_1 = a$	$x_1 = b$
$x_2 = a$	0	0
$x_2 = b$	0	35

$C_0(a) = 0, C_0(b) = 1$
$C_1(a) = 36, C_1(b) = 0$
$C_2(a) = 0, C_2(b) = 0$
$C_\phi = 20, \top = 30$

C_{01} :

	$x_0 = a$	$x_0 = b$
$x_1 = b$	0	0

C_{02} :

	$x_0 = a$	$x_0 = b$
$x_2 = a$	0	0

C_{12} :

	$x_1 = b$
$x_2 = a$	0

$C_0(a) = 0, C_0(b) = 5$
$C_1(b) = 0$
$C_2(a) = 0$
$C_\phi = 20, \top = 30$

Figure 4.10: (Up) Simple example with three variables and its initial binary and unary cost functions, $C_\phi = 0$ and $\top = 30$. (Down, Left) Cost functions after projections from binary to unary costs functions and from unary cost functions to C_ϕ; (Down, Right): Cost functions after deletion of inconsistent values $x_1 = a$ and $x_2 = b$.

line 16 of execution trace, $C_\phi = 20$ has been aggregated completely in x_0. Also, x_0 receives a COST message with $LB(a) = UB(a) = 20$ so $\top$ is updated to 20 (the cost of the best solution found so far). Then, x_0 is able to delete value b since $C_0(b) + C_\phi > \top$ $(5 + 20 > 25)$. Since $x_0 = b$ has been deleted, x_0 will not explore this value. Messages sent on Figure 4.11

(e) and (f) are not sent on BnB-ADOPT$^+$-AC execution. Instead, x_0 sends a VALUE message with $x_0 = a$ and terminates (Figure 4.11 (g)). Lines 20-30 and 33-38 of the execution trace

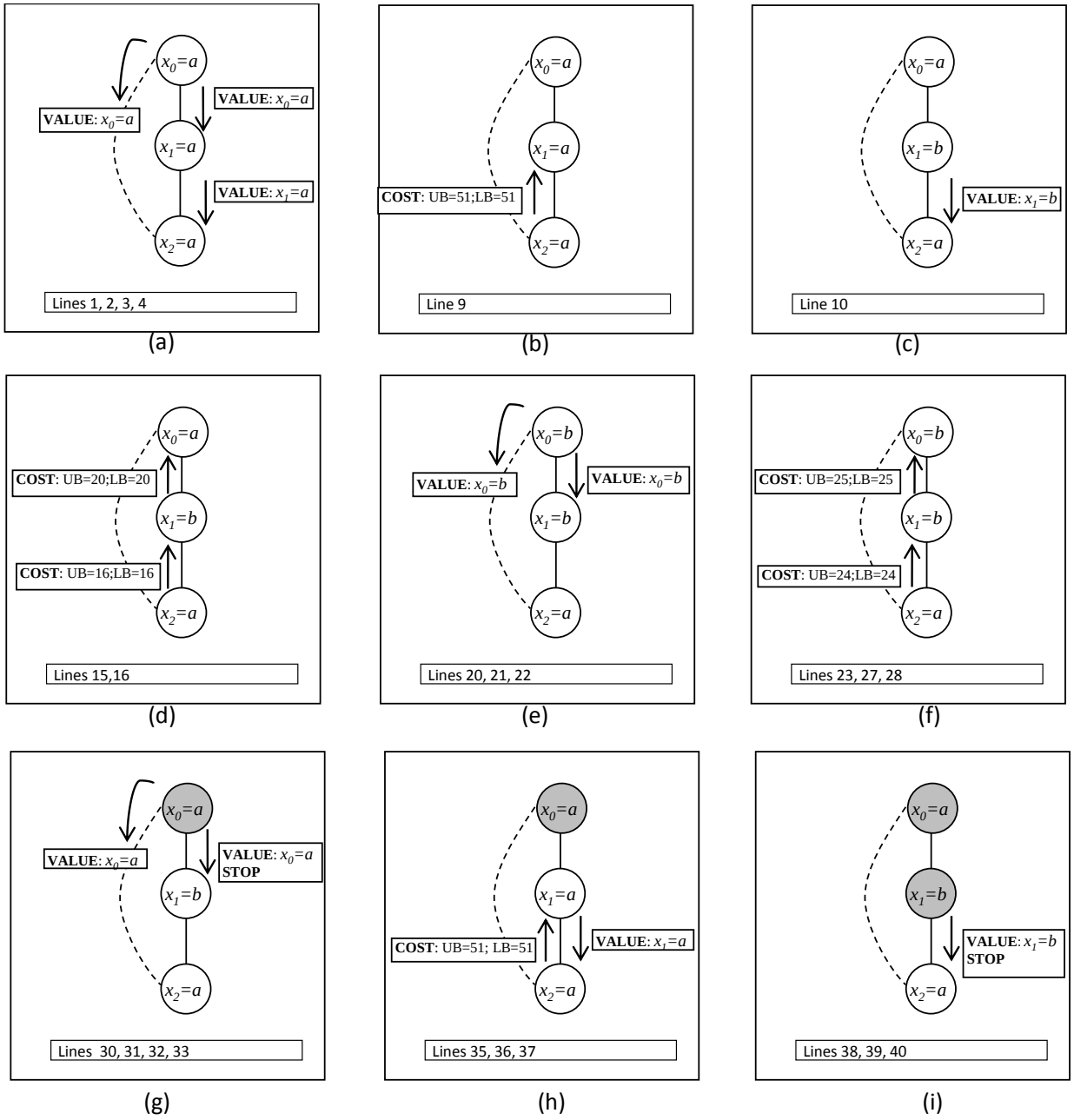


Figure 4.11: BnB-ADOPT$^+$ execution example. Lines in the bottom of each sub-figure correspond to a line in the execution trace (Figure 4.1). In BnB-ADOPT$^+$-AC, messages represented in cases (e), (f) and (h) are not sent.

	BnB-ADOPT$^+$	BnB-ADOPT$^+$-AC
(1)	**start:** $x_0 = a; x_1 = a, x_2 = a$	**start:** $x_0 = a; x_1 = a, x_2 = a$
(2)	x_1 **received VALUE:** $x_0 = a$	x_1 **received VALUE:** $x_0 = a$**, TOP=30,** $C_\phi = 0$
(3)	x_2 **received VALUE:** $x_0 = a$	x_2 **received VALUE:** $x_0 = a$**, TOP=30,** $C_\phi = 0$
(4)	x_2 **received VALUE:** $x_1 = a$	x_2 **received VALUE:** $x_1 = a$**, TOP=30,** $C_\phi = 0$
(5)	**(backtrack:** $x_1 = b$**)**	**(backtrack:** $x_1 = b$**)**
(6)		(x_1 deletes value a)
(7)		x_0 received DEL: $x_1 = a$
(8)		x_2 received DEL: $x_1 = a$
(9)	x_1 **received COST: sender =**x_2**, context=**$\{x_1 = a, x_2 = a\}$**, LB=51, UB=51**	x_1 **received COST: sender =**x_2**, context=**$\{x_1 = a, x_2 = a\}$**, LB=51, UB=51**
(10)	x_2 **received VALUE:** $x_1 = b$	x_2 **received VALUE:** $x_1 = b$**, TOP=30,** $C_\phi = 15$
(11)		(x_2 deletes value b)
(12)		x_0 received DEL: $x_2 = b$
(13)		x_1 received DEL: $x_2 = b$
(14)		x_1 received VALUE: $x_0 = b$, TOP=30, $C_\phi = 20$
(15)	x_1 **received COST: sender =** x_2**, context=**$\{x_0 = a, x_1 = b\}$**, LB =16, UB=16**	x_1 **received COST: sender =** x_2**, context=**$\{x_0 = a, x_1 = b\}$**, LB =16, UB=16**
(16)	x_0 **received COST: sender =** x_1**,context=**$\{x_0 = a\}$**, LB =20, UB=20**	x_1 **received COST: sender =** x_1**, context=**$\{x_0 = a\}$**, LB =20, UB=20**
(17)		(x_0 deletes value b)
(18)		x_1 received DEL: $x_0 = b$
(19)		x_2 received DEL: $x_0 = b$
(20)	(backtrack: $x_0 = b$)	
(21)	x_1 received VALUE: $x_0 = b$	
(22)	x_2 received VALUE: $x_0 = b$	
(23)	x_1 received COST: sender = x_2, context=$\{x_0 = b, x_1 = b\}$, LB=24, UB=24	
(24)	(backtrack: $x_1 = a$)	
(25)	x_2 received VALUE: $x_1 = a$	
(26)	x_1 received COST: sender = x_2, context=$\{x_0 = b, x_1 = a\}$,LB =55, UB=55	
(27)	(backtrack: $x_1 = b$)	
(28)	x_0 received COST: sender = x_1, context=$\{x_0 = b\}$,LB =25, UB=25	
(29)	x_2 received VALUE: $x_1 = b$	
(30)	(backtrack: $x_0 = a$)	
(31)	x_1 **received VALUE:** $x_0 = a$	x_1 **received VALUE:** $x_0 = a$**, TOP=20,** $C_\phi = 20$
(32)	x_1 **received STOP**	x_1 **received STOP**
(33)	x_2 received VALUE: $x_0 = a$	
(34)	x_1 received COST: sender = x_2, context=$\{x_0 = a, x_1 = b\}$, LB=16, UB=16	
(35)	(backtrack: $x_1 = a$)	
(36)	x_2 received VALUE: $x_1 = a$	
(37)	x_1 received COST: sender = x_2, context=$\{x_0 = a, x_1 = a\}$, LB=51, UB=51	
(38)	(backtrack: $x_1 = b$)	
(39)	x_2 **received VALUE:** $x_1 = b$	x_2 **received VALUE:** $x_1 = b$**, TOP=20,** $C_\phi = 20$
(40)	x_2 **received STOP**	x_2 **received STOP**
	No more messages...	No more messages...
	34 total messages	20 total messages
	17 VALUE msg , 15 COST msg	8 VALUE msg, 4 COST msg, 6 DEL msg
	15 cycles	7 cycles
	TOTAL cost: 20	TOTAL cost: 20
	OPT. SOLUTION: $x_0 = a; x_1 = b; x_2 = a$	OPT. SOLUTION: $x_0 = a; x_1 = b; x_2 = a$

Table 4.1: Trace of BnB-ADOPT$^+$ and BnB-ADOPT$^+$-AC on the example of Figure 4.10. Some messages have been omitted to simplify the explanation. Messages in bold are common to both algorithms. Notice that from lines 20 to 38 BnB-ADOPT$^+$ messages are saved in BnB-ADOPT$^+$-AC.

are not executed in BnB-ADOPT$^+$-AC as result of value deletions. As we can see, removing inconsistent values has been beneficial, the propagation of deletions and aggregation of the global lower bound C_ϕ has produced a positive impact.

4.3.4 Simultaneous Deletions

When a value is deleted in an agent *self* binary projections are perform from *self* to neighbors, since neighbors may have lost their support. If deletions are non-simultaneous in BnB-ADOPT$^+$-AC/FDAC (that is, if deletions never occur at the same time on neighboring agents), it is easy to see that projections are performed in the same order on both agent (*self* and neighbor), so their cost functions eventually remain equivalent. However, in the case that deletions occur at the same time on neighboring agents, something different happens.

Consider the example in Figure 4.12. First row correspond to actions taking place in agent i and second row actions taking place in agent j. Every column show simultaneous operations, occurring at the same time on i and j. Black domain values and costs are values and costs stored in the agent. Gray domain values and costs are what the agent believes of its neighbor (this information is not stored in the agent). Agents i and j only store the unary costs of their own domain. Lines represent binary costs C_{ij} with cost one and unary costs appear between parenthesis. Initially, $C_\phi = 0$, $C_i(b) = 1$, $C_j(b) = 1$ and the rest of unary costs are zero.

In the first column, two simultaneous deletions take place on agents i and j. In the second column, both agents make a projection from *self* to the neighbor and send a DEL message. When projecting over the neighbor, binary costs are reduced from C_{ij} and the agent assumes that an increment in the unary costs of the neighbor will eventually occur, when the DEL

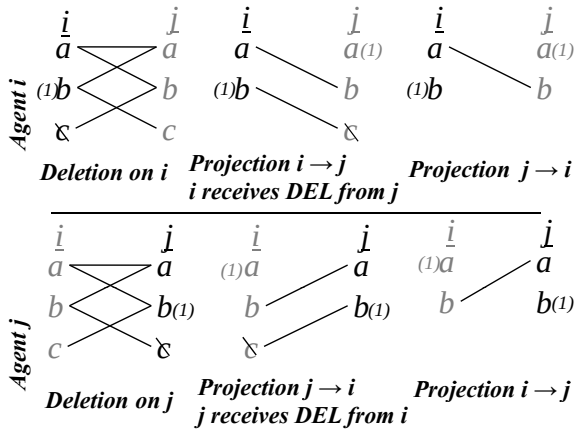


Figure 4.12: Agents i and j, and the process of two simultaneous deletions. Possible values for each agent are a, b, c, unary costs appear between parenthesis. In black, what an agent knows of itself. In gray, what an agent believes of the other agent. Lines indicate pairs of values with cost 1, no lines indicate cost 0.

message arrives to the neighbor. But notice that, because these operations occur at the same time, the order of resulting projections is opposite on agent i and j. This would not be the case if one deletion would have preceded the other. Then both agents would have kept the same ordering in projections (for example, a projection first from i to j and after from j to i) and they would have obtained $C_\phi = 1$. Notice that in the example C_ϕ remains zero.

Both agents projected at the same time a binary cost of 1 from *self* to the unary cost of their neighbor, but this operation was never performed in the neighbor (a cost of 1 is decremented in the binary costs of *self* but is never incremented in the unary costs of neighbor). So this cost is lost from the problem. Notice that costs are not counted twice and no illegal deletions are produced, but loosing costs from the problem diminish deletion opportunities. In addition, on the last column the resulting cost functions on i and j are not equivalent.

One may wonder then if the approach of Section 4.3.1 is correct and complete. The answer is yes because soft arc consistency operations are done in a copy of the cost functions, while search is performed using the original cost functions to calculate costs. As long as there is no illegal deletion of values (which is not the case here, since costs are not counted twice), search maintains its completeness.

However, we can avoid this behavior assuring synchronous deletions. It is impossible that two agents know if they are performing deletions at the same time, but it is possible that they communicate beforehand and agree on the order to follow. If one deletion always precedes the other, projections on neighboring agents maintain the same order. This assures that cost functions remain equivalent and no costs are lost from the problem.

Another option is to omit synchronization of deletions, knowing that there is a possibility that some costs may be lost. This could be practical if simultaneous deletions are considered rare in some problems. Observe that the lost of cost shown in Figure 4.12 happens because both agents i and j deleted values that were a support on the neighbor agent and these supports were deleted simultaneously, otherwise costs would not be lost even if performing the deletions at the same time.

To maintain synchronous deletions, two main changes must be done in BnB-ADOPT$^+$-AC:

- Two new messages are introduced to synchronize deletions: SYNC_1 and SYNC_2
- Agents have a *locked* property. While an agent is *locked* it is able to read and process messages, but it will not change its value or send messages to neighbors, except for synchronization messages SYNC_1 and SYNC_2. An agent is *locked* because it is waiting

to delete a value, or because a deletion is occurring in one or several neighbors. A *locked* agent changes to *unlocked* when it is no longer locked with any of its neighbors.

On Figure 4.13 and 4.14 a pseudocode of the synchronous deletion process for BnB-ADOPT$^+$-AC is shown. Modifications are described below:

- `CheckDomainForDeletions`. When agent i realizes that it can delete values from its domain, instead of immediately erasing them, it marks them as pending to delete and sends DEL messages to neighbors $k_1, k_2, ..k_i$. Afterwards, i is *locked* with neighbors $k_1, k_2, ..k_i$, so i can process incoming messages but it can not change its value or send VALUE, COST or DEL messages (lines 5-10).

- `ProcessDelete`. When neighbor k receives a DEL message from i it can be the case that k is already *locked* with i, this means that simultaneous deletions are taking place. In this case, the higher agent in the pseudo-tree is the one that processes the DEL message first, otherwise the message remains as pending to process (lines 12-13) and will be processed afterwards when the agent is *unlocked* (lines 29-31). To process the DEL message, k deletes the values of i from its copy of D_i, and performs a projection from i to *self*. After this, it sends a message SYNC$_1$ to i to inform that the deletion has been processed, and change its status to *locked* with i (lines 15-20).

- `ProcessSYNC`$_1$. Only after receiving SYNC$_1$ message from *all* its neighbors i is unlocked (line 23, lines 35-37). At this point, all neighbors k have done a projection from i to k. Then, i deletes the values from its domain, makes projections from *self* to k for every neighbor k, and send a SYNC$_2$ messages to neighbors (lines 24-28).

- `ProcessSYNC`$_2$. When neighbor k receives a SYNC$_2$ message from i, it unlocks from i (lines 33-34).

- `ProcessStop`. A special case should be consider on termination. When an agent terminates execution it informs its lower neighbors (lines 40-43). Once an agent has stopped, it will no longer be considered in the synchronizing process because it will not be able to respond, causing other agents to freeze forever. Therefore, before sending DEL messages to an agent or updating the *locked* status with an agent, it is first checked that the agent has not stopped execution (line 7, line 18).

```
01 procedure CheckDomainForDeletions()
02   for each v ∈ D_self do
03     if C_self(v) + C_φ > ⊤ then deleteValues.add(v);
04     if Σ_{c∈children st.contexts(c,v)={self}} lb(c, v) > ⊤ then deleteValues.add(v);
05 if valuesToDelete.size > 0 then
06   for each k ∈ neighbors(self) do
07     if ¬hasStopped(k) then
08       sendMsg:(DEL, self, k, valuesToDelete);
09       locked(k) = true;
10   UpdateLockStatus();

11 procedure ProcessDelete(msg)
12 if locked(msg.sender) and self < sender then
13   processPending(msg.sender) = msg;
14 else
15   D_sender ← D_sender − {msg.valuesToDelete};
16   BinaryProjection(self, sender);
17   sendMsg:(SYNC_1, self, msg.sender);
18   if ¬hasStopped(msg.sender) then
19     locked(msg.sender) = true;
20     UpdateLockStatus();

21 procedure ProcessSYNC_1(msg)
22 locked(msg.sender) = false;
23 UpdateLockStatus();
24 if ¬locked then
25   D_self ← D_self − valuesToDelete;
26   valuesToDelete ← ∅
27   for each k ∈ neighbors do
28     BinaryProjection(k, self); sendMsg:(SYNC_2, self, k);
29   for each msg ∈ processPending do
30     ProcessDelete(msg);
31     processPending.remove(msg);

32 procedure ProcessSYNC_2(msg)
33 locked(msg.sender) = false;
34 UpdateLockStatus();

35 procedure UpdateLockStatus()
36 locked = false;
37 for each k ∈ neigbors(self) do if locked(k) then locked = true;
```

Figure 4.13: Pseudocode for Synchronizing Deletions.

- `Backtrack`. If the agent is $locked$, it will not change its value or send messages to neighbors (except synchronization messages) (line 45, line 48).

```
39 procedure ProcessStop(msg)
40   if msg.sender == parent then receivedTerminate ← true;
41   locked(msg.sender) = false;
42   UpdateLockStatus();
43   hasStopped(msg.seder) = true;

44 procedure Backtrack()
45   if locked then return;
46   UpdateLBUB();
47   CheckDomainForDeletions();
48   if locked then return;
49   if LB(value) ≥ min(TH, UB) then
50     value ← argmin_{v∈D_self} {LB(v)};
51   UnaryProjectionToC_φ();
52   if value has changed then
53     SendValueToLowerNeighbors();
54   else
55     SendValueToChildrenToUpdateTH();
56   if (receivedTerminate or self == root) and LB == UB and
57   LB(value) == UB(value) then
58     SendStopMessageToLowerNeighbors();
59   SendCostToParent();
```

Figure 4.14: Pseudocode for Synchronizing Deletions.

4.3.5 Experimental Results

We evaluate the efficiency of BnB-ADOPT$^+$-AC/FDAC with synchronous deletions against original BnB-ADOPT$^+$. We present the number of messages exchanged (communication effort) and the number of non-concurrent constraint checks (computational effort). We also present the numbers of cycles of the simulator, as a complementary measure. Experimental evaluation is made on unstructured instances with binary random DCOPs, and on structured distributed meeting scheduling datasets.

We generated random DCOP instances: $\langle n = 10, d = 10, p_1 = 0.3, 0.4, 0.5, 0.6, 0.7, 0.8\rangle$. where costs are selected from an uniform cost distribution. Two types of binary cost functions are used, small and large. Small cost functions extract costs from the set $\{0, \ldots, 10\}$ while large ones extract costs from the set $\{0, \ldots, 1000\}$. The proportion of large cost functions is 1/4 of the total number of cost functions (this is done to introduce some variability among tuple costs; using a unique type of cost function causes that all tuples look pretty similar from an optimization view). Results appear in Table 4.2, averaged over 50 instances.

p_1	Algorithm	#Msgs	#VALUE	#COST	#DEL	#UCO	#Cycles	#NCCC	#Deletions
	BnB-ADOPT$^+$	28,873	15,007	13,857	**0**	**0**	5,930	318,309	0
	BnB-ADOPT$^+$-AC	3,076	1,473	1,267	98	**0**	714	43,724	64
0.3	BnB-ADOPT$^+$-FDAC	**2,062**	**917**	**767**	108	18	**436**	**34,226**	**78**
	BnB-ADOPT$^+$	97,816	47,762	50,045	**0**	**0**	21,574	1,392,129	0
	BnB-ADOPT$^+$-AC	31,325	14,759	16,092	145	**0**	7,950	491,241	63
0.4	BnB-ADOPT$^+$-FDAC	**15,765**	**7,427**	**7,781**	162	54	**3,923**	**262,279**	**80**
	BnB-ADOPT$^+$	243,220	114,828	128,382	**0**	**0**	62,780	5,233,455	0
	BnB-ADOPT$^+$-AC	78,572	37,323	40,630	192	**0**	20,329	1,704,705	64
0.5	BnB-ADOPT$^+$-FDAC	**38,393**	**17,657**	**19,999**	215	48	**11,267**	**953,622**	**80**
	BnB-ADOPT$^+$	613,794	301,717	312,067	**0**	**0**	108,937	10,914,925	0
	BnB-ADOPT$^+$-AC	286,963	140,688	145,594	211	**0**	59,102	6,003,448	57
0.6	BnB-ADOPT$^+$-FDAC	**174,660**	**85,652**	**88,141**	254	57	**37,037**	**3,863,055**	**76**
	BnB-ADOPT$^+$	2,106,960	1,047,228	1,059,723	**0**	**0**	390,674	53,654,134	0
	BnB-ADOPT$^+$ -AC	1,662,531	832,850	829,320	103	**0**	328,856	40,955,650	21
0.7	BnB-ADOPT$^+$-FDAC	**1,181,727**	**588,523**	**592,417**	223	64	**242,160**	**30,273,694**	**53**
	BnB-ADOPT$^+$	2,455,663	1,220,387	1,235,267	**0**	**0**	450,070	65,951,172	0
	BnB-ADOPT$^+$-AC	1,949,410	959,475	989,538	113	**0**	397,831	53,369,192	22
0.8	BnB-ADOPT$^+$-FDAC	**1,222,317**	**593,222**	**628,167**	267	70	**261,076**	**35,673,984**	**56**

Table 4.2: Experimental results in random binary DCOPs. BnB-ADOPT$^+$ (first row) compared to BnB-ADOPT$^+$-AC (second row) and BnB-ADOPT$^+$-FDAC (third row)

On the meeting scheduling formulation, we present 4 cases obtained from the DCOP repository (Yin, 2008) with different hierarchical scenarios and domain 10: case A (8 variables), case B (10 variables), case C (12 variables) and case D (12 variables). Results appear in Table 4.3, averaged over 30 instances.

DFS pseudo-trees are built for every instance following a most-connected heuristic. For each problem, we calculate an initial $\top$ to have higher pruning opportunities during AC and FDAC preprocess. This initial $\top$ is calculated in the following way. Each leaf agent in the pseudo-tree choose the best value (the one that minimizes LB(v)) with its local information, and informs its parent of the selected value and its cost. Parents receive this information from children and choose their own value (minimizing LB(v), observe that only lower agents assignments are taking into account) and also inform their parents accumulating the cost of the partial solution. When all agents have chosen their value, the $root$ agent can compute the cost of a complete solution (likely not the optimal one) which is an upper bound of the optimum problem cost. This cost is considered the initial $\top$ of the problem and is propagated to the rest of the agents in the pseudo-tree. Only two messages per each agent are required: one from child to parent informing the partial solution cost, and one from parent to children informing of the initial $\top$. These messages and computation are considered in BnB-ADOPT$^+$-AC/FDAC

	#Msgs	Algorithm	#VALUE	#COST	#DEL	#UCO	#Cycles	#NCCC	#Deletions
		BnB-ADOPT$^+$	17,714	7,358	10,349	**0**	**0**	7,901	661,043	0
		BnB-ADOPT$^+$-AC	4,273	1,745	2,192	101	**0**	1,603	131,831	46
A		BnB-ADOPT$^+$-FDAC	**3,756**	**1,513**	**1,892**	104	6	**1,383**	**116,462**	**49**
		BnB-ADOPT$^+$	15,187	7,595	7,583	**0**	**0**	2,628	181,443	0
		BnB-ADOPT$^+$-AC	5,379	2,489	2,556	99	**0**	1,177	**81,701**	46
B		BnB-ADOPT$^+$-FDAC	**5,073**	**2,311**	**2,391**	108	7	**1,155**	82,612	**53**
		BnB-ADOPT$^+$	11,163	6,188	4,963	**0**	**0**	2,311	118,969	0
		BnB-ADOPT$^+$-AC	3,125	1,603	1,194	96	**0**	646	39,961	75
C		BnB-ADOPT$^+$-FDAC	**2,912**	**1,466**	**1,102**	99	6	**628**	**39,772**	**82**
		BnB-ADOPT$^+$	13,237	7,086	6,140	**0**	**0**	1,530	77,080	0
		BnB-ADOPT$^+$-AC	2,472	1,222	926	94	**0**	305	21,182	79
D		BnB-ADOPT$^+$-FDAC	**2,189**	**1,056**	**796**	97	5	**275**	**20,048**	**83**

Table 4.3: Experimental results in Meeting Scheduling instances. BnB-ADOPT$^+$ (first row) compared to BnB-ADOPT$^+$-AC (second row) and BnB-ADOPT$^+$-FDAC (third row)

results.

On random DCOPs, BnB-ADOPT$^+$-AC/FDAC showed clear benefits on communication costs with respect to BnB-ADOPT$^+$. Maintaining AC level (BnB-ADOPT$^+$-AC) the number of exchanged messages is divided by a factor from 1.2 to 9. Notice that this reduction is obtained generating only very few DEL messages. In addition, including the FDAC level (BnB-ADOPT$^+$-FDAC) enhances this reduction, dividing the number of BnB-ADOPT$^+$ exchanged messages by a factor from 1.7 to 14. Notice that maintaining the higher FDAC level increases the number of deletions, so it increases the number of DEL messages in addition to the new UCO messages. However as result of these deletions, important savings are obtained compared to AC. In general, including few DEL and UCO messages and performing extra local computation to enforce soft arc consistency allows BnB-ADOPT$^+$-AC/FDAC to obtain large reductions in VALUE and COST messages. VALUE and COST messages are used to coordinate the search process, since sub-optimal values are removed agents need to test for fewer values and consequently they generate less VALUE and COST messages.

This reduction in messages is so important, that the number of NCCC also show clear improvements. Although agents need to perform more local computation to maintain local consistency, the number of NCCC is significantly reduced. This is the combination of two opposite trends: agents are doing more work enforcing soft arc consistency and processing new DEL and UCO messages, but less work processing less VALUE and COST messages. This combination turns out to be very beneficial, saving computational effort in all cases tested.

For the meeting scheduling instances, we observe the same trend: benefits with AC, en-

hanced by FDAC. For the stronger FDAC level (BnB-ADOPT$^+$-FDAC) messages are divided by a factor from 2.9 to 6 and there are significant savings in NCCCs. Few DEL and UCO messages are needed, and the extra computational effort required to maintain AC or FDAC is effectively balanced by a decrement on VALUE and COST messages.

4.4 GAC in N-ary Constraints

In this Section we explain how to maintain the GAC level (defined in Section 2.1.3.1 for the centralized case) in distributed problems with n-ary constraints.

Such as it happens with AC enforcement when cost functions are binary, agents must perform projections following an established ordering (we maintain the pseudo-tree ordering) from their n-ary cost functions to unary cost functions. Then projections are performed from unary cost functions to C_ϕ and domains are checked for inconsistent values that can be removed from the problem. Every time a value is deleted in one agent it is propagated to neighbors which must reinforce GAC. As result, new deletions may appear on neighbors.

Agents maintain the same modification as in the binary case:

- The domain of neighboring agents are represented in *self*.
- Extra information needed to discover inconsistent values are propagated among the agents, such as: $\top$, the global C_ϕ and projections to C_ϕ from agents. Specifically in BnB-ADOPT$^+$, this information travels in VALUE and COST messages (see Figure 4.3).
- A new DEL message is added to notify value deletions to neighbors.

The main difference with the binary case lies in the projection from n-ary cost functions to unary cost functions, which is performed in every agent in the following way.

The projection of costs from the n-ary cost function C_S to the unary cost function $C_i(a)$ of variable x_i, where S is the set of variables involved in the constraint, $x_i \in S$ and $a \in D_i$ is a flow of costs defined as follows:

Let α_a be the minimum cost in the set of tuples of C_S where $x_i = a$ (namely $\alpha_a = min_{t \in tuples\, s.t.\, x_i = a} C_S(t)$). The projection consists in adding α_a to $C_i(a)$ (namely, $C_i(a) = C_i(a) + \alpha_a, \forall a \in D_i$) and subtracting α_a from $C_S(t)$ (namely, $C_S(t) = C_S(t) - \alpha_a, \forall t \in tuples\, s.t.\, x_i = a, \forall a \in D_i$).

In every agent *self*, projections are performed from their copy of the n-ary C_S to the unary costs of every agent in S, following the established order. As in the binary case, n-ary cost functions in *self* are decremented and unary costs are only incremented if the projection is performed over *self*. Unary cost of neighbors are not updated, since this information is not known by *self*.

4.5 Higher Consistency Levels

Combining BnB-ADOPT$^+$ with soft AC have caused substantial efficiency improvements, drastically reducing the number of messages required to compute the optimal solution and also reducing the computational effort required. Because of that, we aim at combining BnB-ADOPT$^+$ with higher consistency levels. The next level of soft arc consistency, in the centralized case, is EDAC (de Givry et al., 2005). However, when moving to the distributed case we have not been able to achieve the EDAC level, because of the following reasons.

Maintaining AC and FDAC during distributed search requires each agent to know the binary cost functions in which it is involved and the unary costs of its values. This is in agreement with usual privacy requirements not permitting an agent to know the unary costs of values of other agents. However EDAC seems to be conflicting with this privacy requirement. EDAC maintenance requires that at each variable there is a value with unary cost 0 which is fully supported in both directions (cost functions linking ancestors with this variable, cost functions linking this variable with descendants).

Let us consider two agents i, j, where $i < j$ sharing a cost function C_{ij}. To assure that j has a value fully supported by i, i has to extend some of its unary costs into the binary ones, which will be projected on the unary costs of j values. However, i will only extend its unary costs if it is sure that from this operation C_ϕ will increase (otherwise termination is not guaranteed). The most direct way to assure this condition is that i knows the unary costs of j or vice versa. This can be seen in line 1 of function `FindExistentialSupport` in (de Givry et al., 2005). The expression of α involves $C_i(a)$ and $C_j(b)$, the unary costs of values of x_i and x_j. While this causes no difficulties in a centralized approach, it becomes an issue in a distributed setting, since it breaks usual privacy requirements.

Even ignoring privacy, reaching EDAC in a distributed context could be expensive since agents should inform their neighbors when their unary costs change, which requires more com-

munication and computation. For higher consistency levels, even more information needs to be shared and exchanged among agents.

As alternative, we have explored two new ways to perform propagation in DCOPs. These two new approaches are explained next.

4.6 FDAC in Multiple Representations

Looking for an inference technique that allows more pruning than FDAC and maintaining standard privacy assumptions, we observed the following fact. The first variable in a FDAC ordering satisfies the EDAC property: since the variable is FDAC each value has a support and there is at least one value with unary cost 0; since the variable considered is the first one in the ordering, these supports have to be full supports. This suggests us an alternative way for distributed problems: instead of having a single ordering of agents, we may have several orderings. On each ordering we enforce FDAC, and the first variable on every ordering satisfies EDAC. For every ordering, we will need a copy of the cost functions, on which projection and extensions are performed. The key point is that an inconsistent value discovered in one copy of the cost function can be erased from all copies in any ordering.

It is known that with different variable orderings FDAC maintenance may discover different inconsistent values. This fact also motivates the present approach. It is very expensive to determine the *best ordering*, in the sense of the ordering that prunes most. Instead of looking for that ordering, we consider as alternative to keep *multiple orderings* $O_1, ..., O_r$ at each agent, on which FDAC is separately enforced. Having different orders produces different flows of costs and as result, some values may be found inconsistent in some ordering. Maintaining FDAC in O_p may cause the deletion of value a in variable i: this deletion is propagated to all other orderings $O_1, ..., O_{p-1}, O_{p+1}, O_r$.

Propagating value deletions among different orderings is legal and do not compromise correctness and completeness. Let us assume that enforcing FDAC on the ordering O_1 causes to delete value (i, a), while enforcing FDAC on the ordering O_2 causes to delete value (j, b). Then, both values can be deleted without losing any optimal solution. If enforcing FDAC using ordering O_1 we delete value (i, a), this means that value a for variable i will not appear in any optimal solution of the problem. This fact derives directly from soft arc consistency, and it is independent of the ordering used. The same situation happens with ordering O_2 and value

(j, b). Therefore, it is legal to remove both values from their domains, independently of the ordering used.

Since cost functions evolve depending on the ordering used, we prefer to talk about different representations of cost functions instead of different orderings (clearly, each ordering defines a representation).

The idea of multiple representations can be included in BnB-ADOPT$^+$, producing the new BnB-ADOPT$^+$-FDAC-MR (BnB-ADOPT$^+$-FDAC with multiple representations) algorithm. Implementing r representations requires each agent holding a set of r cost functions $\{C_1, C_2...C_r\}$. On all r cost functions agents enforce FDAC. The direction of the arc consistency enforcement will be defined by the set of partial orders among agents $\{O_1, O_2, ..O_r\}$.

Orders are generated in the following way: r different agents are selected, each of them will be the first agent in each of the r orders. Each selected agent chooses randomly a neighbor and sends a message containing the order. When this message arrives, if the receiver is not already in the order it decides if it wants to be the next agent. After this, the receiver chooses randomly another neighbor and sends the order. When the order is complete (all agents are included), this process stops.

In BnB-ADOPT$^+$-FDAC-MR, agents need to store:

1. One copy of the binary and unary cost functions for every representation r.
2. The order O_r for projections/extensions in every representation r.
3. One C_ϕ value for every representation r. Since different projections and extensions are performed on every representation, different C_ϕ values are obtained.
4. The $projectionToC_\phi$ of every child for every representation r. Since different projections and extensions are performed on every representation, agents will contribute to the C_ϕ differently on every one of them.

The following changes in messages are needed to maintain the previous structures:

- VALUE: A vector C_ϕ[] is sent containing the C_ϕ values for every representation.
- COST: A vector $projectionsToC\phi$[] is sent containing the subtree projections to the C_ϕ for every representation.

- UCO: A vector $vectorOfExtensions$[][] is sent containing the extensions for every representation.

Every time there is a deletion, the agent will need to reinforce FDAC over the r representations.

4.6.1 Experimental Results

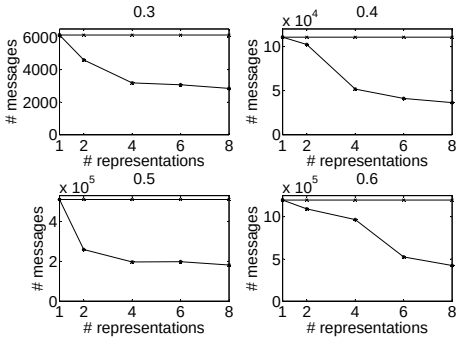
We evaluate the efficiency of BnB-ADOPT$^+$-FDAC-MR (multiple representations) with respect to BnB-ADOPT$^+$-FDAC (single representation) on unstructured instances with binary random DCOPs, and on structured distributed meeting scheduling.

We have generated random DCOP instances: $\langle n = 10, d = 10, p_1 = 0.3, 0.4, 0.5, 0.6\rangle$. Two types of binary cost functions are used, small and large. Small cost functions extract costs from the set $\{0, \ldots, 10\}$ while large ones extract costs from the set $\{0, \ldots, 1000\}$. The proportion of large cost functions is 1/4 of the total cost functions. On the meeting scheduling formulation, we present 4 cases obtained from the DCOP repository (Yin, 2008) with different hierarchical scenarios and domain 10: case A (8 variables), case B (10 variables), case C (12 variables) and case D (12 variables).

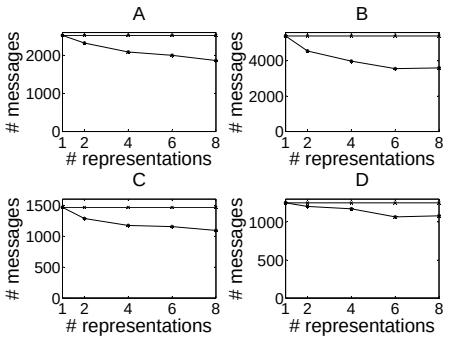
Figure 4.15 (a) and (b) shows experimental results for meeting scheduling and random problems averaged over 30 and 50 instances respectively, with a number of representations from 2 to 8. For an easy comparison, BnB-ADOPT$^+$-FDAC results are drawn as an horizontal line. On random DCOPs, BnB-ADOPT$^+$-FDAC-MR showed clear benefits on communication costs with respect to BnB-ADOPT$^+$-FDAC. Maintaining from 4 to 6 representations, the number of exchanged messages is divided by a factor of at least 2. For meeting scheduling instances we also observe a decrement in the number messages exchanged, although to a smaller extent.

In Figure 4.15 we observe that benefits in communication are unevenly distributed: the #saved messages/#representations ratio is higher in the left-half of the plots. This suggests that $n/2$ could be a good number of representations, although more work is needed to sustain this conjecture.

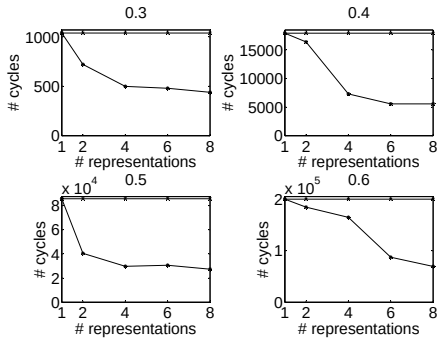
Table 4.4 shows results in detail of the experiments maintaining 6 representations. Notice that maintaining FDAC-MR produces only a few extra DEL and UCO messages. In the case of DEL messages, this slight increment is because more deletions have been produced. In the case of UCO messages, the increment is because UCO messages are only sent if their $costOfExtension$ vector is different from zero. As several representations are maintained



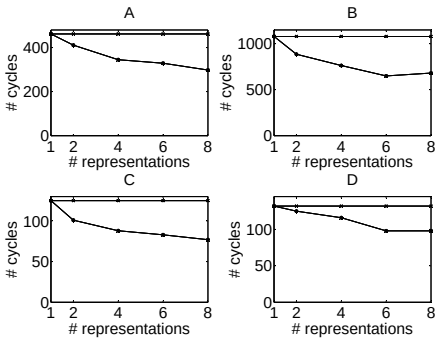
(a) Random Instances



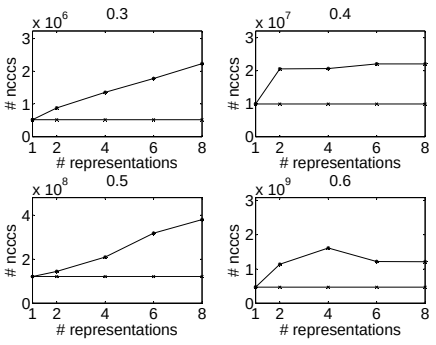
(b) Meeting Scheduling



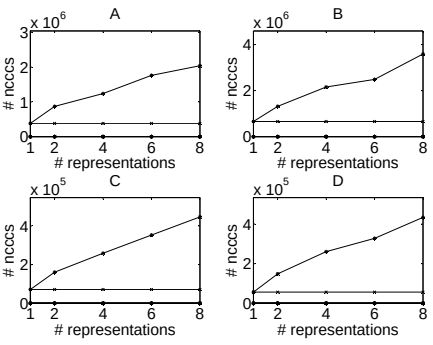
(c) Random Instances



(d) Meeting Scheduling



(e) Random Instances



(f) Meeting Scheduling

Figure 4.15: Experimental results of BnB-ADOPT$^+$-FDAC-MR. Number of messages, cycles and NCCCs (y axis) when solving random instances (a,c,e) and meeting scheduling (b,d,f) with an increasing number of representations (x axis).

(a) Random DCOPs

p_1	Algorithm	#Msgs	#VALUE	#COST	#DEL	#UCO	#Cycles	#NCCC	#Deletions
	BnB-ADOPT$^+$-FDAC	6,128	2,795	3,047	**230**	**28**	1,039	**519,112**	80
0.3	BnB-ADOPT$^+$-FDAC-MR	**3,068**	**1,335**	**1,391**	245	69	**480**	1,778,525	**86**
	BnB-ADOPT$^+$-FDAC	110,696	48,281	62046	**288**	**53**	17,937	**9,910,897**	78
0.4	BnB-ADOPT$^+$-FDAC-MR	**41,147**	**19,309**	**21,357**	311	142	**5,561**	22,063,034	**85**
	BnB-ADOPT$^+$-FDAC	510,411	225,155	284,781	**366**	**82**	85,710	**121,453,697**	78
0.5	BnB-ADOPT$^+$-FDAC-MR	**198,474**	**91,506**	**106,299**	397	244	**30,659**	318,565,730	**85**
	BnB-ADOPT$^+$-FDAC	1,196,935	475,416	720,975	**408**	**108**	199,971	**470,462,443**	74
0.6	BnB-ADOPT$^+$-FDAC-MR	**524,406**	**209,150**	**314,454**	459	314	**87,357**	1,217,511,858	**84**

(b) Distributed Meeting Scheduling

	#Msgs	Algorithm	#VALUE	#COST	#DEL	#UCO	#Cycles	#NCCC	#Deletions
	BnB-ADOPT$^+$-FDAC	2,524	1,056	1,240	**200**	**5**	462	**382,676**	49
A	BnB-ADOPT$^+$-FDAC-MR	**2,001**	**820**	**921**	216	9	**329**	1,762,278	**53**
	BnB-ADOPT$^+$-FDAC	5,405	2,323	2,863	**184**	**7**	1,080	**659,314**	53
B	BnB-ADOPT$^+$-FDAC-MR	**3,556**	**1,513**	**1,7821**	210	23	**650**	2,487,359	**61**
	BnB-ADOPT$^+$-FDAC	1,467	697	505	**225**	**6**	125	**71,439**	80
C	BnB-ADOPT$^+$-FDAC-MR	**1,156**	**509**	**353**	238	21	**83**	352,795	**85**
	BnB-ADOPT$^+$-FDAC	1,251	526	448	**234**	**8**	132	**56,447**	83
D	BnB-ADOPT$^+$-FDAC-MR	**1,067**	**423**	**345**	241	24	**98**	327,749	**85**

Table 4.4: Experimental results of BnB-ADOPT$^+$-FDAC (first row) compared to BnB-ADOPT$^+$-FDAC-MR (second row) maintaining 6 representations.

with different orders, is more probable that the extensions will be different from zero in any of the 6 representations.

The number of NCCCs increases since more projection and extensions are needed to maintain FDAC-MR. This increment however is not linear with respect to the number of representations maintained, it is smoothed by the fact that less messages are generated and more deletions are performed. So there are messages on the BnB-ADOPT$^+$-FDAC algorithm that are not needed to process with multiple representations, and also there are values that will not be needed to assign or to check for node consistency.

In general, BnB-ADOPT$^+$-FDAC-MR is able to obtain important savings in communication, although suffers from an increment in NCCCs since more work is needed to enforce FDAC in multiple representations. Inconsistent value deletions are increased in BnB-ADOPT$^+$-FDAC-MR. Flows of costs from one agent to another, implemented by UCO messages, allow an agent to pass some of their unary costs to higher agents following different orders, causing more pruning opportunities. In general, it is expected that the combination of multiple enforcements will be able to extend the inference benefits. If we assume the usual case where communication

time is higher than computation time, then the total elapsed time is dominated by communication time and savings in communication can be an important indicator of improvement.

4.7 DAC by Token Passing

In this Section, we propose an alternative way to perform inference and propagate deletions during BnB-ADOPT$^+$. Although this proposal is not the translation into the distributed context of the EDAC approach, we believe it has the same motivation: calculate strong lower bounds that allow to discover sub-optimal values.

When BnB-ADOPT$^+$ is connected with FDAC, agents maintains DAC pointing up to the pseudo-tree, and AC in the opposite direction. This means that unary costs are extended following the pseudo-tree structure, from leaves to root, but never in any other direction (Figure4.16, left). Observe that following this ordering, in an intermediate agent in the pseudo-tree there are always lower neighbors from which extended costs are received and higher neighbors to which costs are extended. However, it would be good if given an agent *self*, all its neighbors could extend unary costs to *self*, expecting to achieve a higher pruning than the one obtained by extending unary costs only from the agents below *self* in the pseudo-tree.

Our proposal considers that an agent *self* becomes privileged and asks its neighbors to extend unary costs to it. Neighbors extend their unary costs on the binary constraints they share with *self* and inform *self* with a message. In this way, *self* receives cost contributions from all neighbors. We call this approach *DAC by token passing*, because agents become privileged after receiving a token (Figure 4.16, center and right).

This approach does not maintain any local consistency property during search, such as AC, FDAC or EDAC. Therefore theoretically we can not assure that it is stronger than these consistency levels. However, empirically we have observed that combining this technique with BnB-ADOPT$^+$ search allows to find an optimal solution with less computational and communication effort than using BnB-ADOPT$^+$-FDAC.

This approach involves two parts that we will explain in detail: preprocess and search.

Preprocess. After building the pseudo-tree, there are three phases:

1. We try to increase unary costs and C_ϕ. In this phase only one token exists at any time. When agent i has the token, it asks its neighbors to extend costs towards it using the new message: ASK(i, j) – i asks j to extend unary costs–. After receiving an ASK message, j performs the extension over i and answers with an UCO message containing

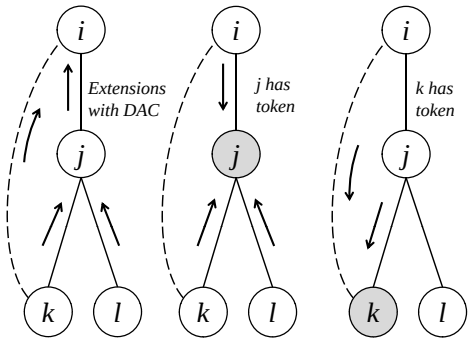


Figure 4.16: Left: Cost extensions with DAC, following the order of the pseudo-tree. Center: Cost extensions when j has the token. Right: Cost extensions when k has the token

the extended costs. When this message is received, unary costs of i may increase, and perhaps C_ϕ increases. After receiving UCO messages from *all* neighbors, i passes the token to the next agent k traversing the pseudo-tree in a depth-first order using the new message: TOKEN(i, k) –i passes the token to k–. Starting at the root, which initially has the token, this process ends when the token comes back to the root. This phase can be iterated, since further iterations might cause extra increments in C_ϕ.

2. BnB-ADOPT$^+$ is executed during a few cycles, to allow VALUE and COST messages to disseminate $\top$ and C_ϕ between all agents.

3. Since agents have $\top$ and C_ϕ they can perform value deletions in their domains. For every $a \in D_i$, the deletion condition is $C_i(a) + C_\phi > \top$. If i deletes some values, it automatically has the token and, after deletion, i sends the token to all its neighbors. If k receives the token, it sends ASK messages to all its neighbors. If an UCO message arrives from i with ∞ as the unary cost of value b, it means i has deleted b from its domain. In this way, value deletions are notified between neighbors (so no DEL messages are required). After k receives UCO messages from all its neighbors, if it is able to increment C_ϕ or delete values, k sends the token to all its neighbors. In this phase, more than one token can exist. However, this does not cause conflicts, since extensions are synchronized with ASK and UCO messages, and when two neighboring agents send ASK messages at the same time, priority is given to the one higher in the pseudo-tree. This phase ends when the network becomes silent.

Search. BnB-ADOPT$^+$ execution starts after the preprocess.

BnB-ADOPT$^+$ execution coexists with the token passing strategy. During search, some values may be found suboptimal in i and deleted unconditionally, as explained in Section 4.2. When this occurs, i automatically has the token and sends it to its neighbors as in phase 3 of preprocess. As result of receiving UCO messages, C_ϕ may increment in i. When this occurs, i also sends the token to their neighbors as in phase 3 of preprocess. Agent i having the token may extinguish it if i does not perform any deletion or C_ϕ increment. The process ends when BnB-ADOPT$^+$ terminates.

It is easy to see that the BnB-ADOPT$^+$ remains optimal since this process only removes values that are proved sub-optimal because of propagation or search. Value removal does not alter the normal execution of BnB-ADOPT$^+$, it simply shrinks the search space causing efficiency benefits. Also, observe that this mechanism can not lead to an infinite loop of operations (sending the token, asking for extensions, etc) compromising termination. This is because token passing can occur only as result of deletions or C_ϕ increments, otherwise the token is not passed to any neighbor. Since deletions and C_ϕ increments are finite, the token passing mechanism is also finite.

4.7.1 Experimental Results

We evaluate the performance of BnB-ADOPT$^+$ combined with DAC by token passing against BnB-ADOPT$^+$-FDAC (with synchronous deletions) on binary random DCOPs and meeting scheduling instances. Binary random DCOPs have 10 variables with domain size 10 and connectivity $p_1 = \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$. Costs are selected from an uniform cost distribution on small (costs from $[0...10]$) and large (cost from $[0...1000]$) cost functions. Large cost functions are 1/4 of the total number of cost functions. Results appear in Table 4.5, averaged over 50 instances. Meeting scheduling instances are obtained from the public DCOP repository (Yin, 2008). We present cases A (8 variables), case B (10 variables), case C (12 variables) and case D (12 variables). Results appear in Table 4.6, averaged over 30 instances. DFS pseudo-trees are built for every instance following a most-connected heuristic.

We present the total number of exchanged messages (communication effort), the number of non-concurrent constraint checks (computational effort) and the number of cycles as a complementary measure.

The proposed DAC by token passing approach reduces the total number of messages exchanged in all cases. Although some extra messages are needed for token passing (asking and

p_1	Algorithm	#Msgs	#Cycles	NCCCs
0.3	BnB-ADOPT$^+$-FDAC	2,062	436	**34,226**
	BnB-ADOPT$^+$ DAC-token	**1,919**	**163**	49,424
0.4	BnB-ADOPT$^+$-FDAC	15,765	3,923	262,279
	BnB-ADOPT$^+$ DAC-token	**12,567**	**1,544**	**260,964**
0.5	BnB-ADOPT$^+$-FDAC	38,393	11,267	953,622
	BnB-ADOPT$^+$ DAC-token	**20,971**	**3,063**	**508,892**
0.6	BnB-ADOPT$^+$-FDAC	174,660	37,037	3,863,055
	BnB-ADOPT$^+$ DAC-token	**125,575**	**16,453**	**3,862,508**
0.7	BnB-ADOPT$^+$-FDAC	1,181,727	242,160	30,273,694
	BnB-ADOPT$^+$ DAC-token	**608,274**	**74,613**	**20,691,307**
0.8	BnB-ADOPT$^+$-FDAC	1,222,317	261,076	35,673,984
	BnB-ADOPT$^+$ DAC-token	**827,615**	**106,307**	**32,854,873**

Table 4.5: Experimental results of binary random DCOPs. BnB-ADOPT$^+$-FDAC (first row) compared to BnB-ADOPT$^+$ combined with DAC by token passing (second row)

	Algorithm	#Msgs	#Cycles	NCCCs
A	BnB-ADOPT$^+$-FDAC	3,756	1,383	116,462
	BnB-ADOPT$^+$ DAC-token	**2,559**	**383**	**75,419**
B	BnB-ADOPT$^+$-FDAC	5,073	1,155	82,612
	BnB-ADOPT$^+$ DAC-token	**3,286**	**350**	**58,583**
C	BnB-ADOPT$^+$-FDAC	2,912	628	39,772
	BnB-ADOPT$^+$ DAC-token	**1,946**	**159**	**37,874**
D	BnB-ADOPT$^+$-FDAC	2,189	275	**20,048**
	BnB-ADOPT$^+$ DAC-token	**1,441**	**74**	30,320

Table 4.6: Experimental Results of Meeting Scheduling instances. BnB-ADOPT$^+$-FDAC (first row) compared to BnB-ADOPT$^+$ combined with DAC by token passing (second row)

informing extensions, etc), this is effectively balanced with less messages needed for search. Synchronous cycles decrement in correspondence with the amount of messages saved and non-concurrent constraints checks (NCCCs) show moderate reductions in most instances.

In BnB-ADOPT$^+$-FDAC (first row), when agents receive DEL or UCO messages they check their domain for deletions and try to increment C_ϕ. However, DAC by token passing (second row) does these operations only after UCO messages are received from all neighbors. This has two benefits: first, agents perform less work since they wait for all unary cost contribution to be aggregated before performing further operations, and second, costs are not extended to other agents until all possible cost contributions from neighbors are obtained and used for deletions or C_ϕ increment.

In summary, the token passing approach does not have the eagerness of trying to delete values as soon as some condition has changed. It also waits for UCO messages from all neighbors

before performing any further extension. This causes benefits in communication and computation. Instances are solved using less messages and generally using less computational effort than with FDAC and preserving privacy requirements.

4.8 Conclusions

In this Chapter we have connected BnB-ADOPT$^+$ with some forms of soft arc consistency aiming at detecting and pruning values which are not part of the optimal solution, with the final goal of improving search efficiency. These deletions are unconditional and do not rely on any previous variable assignment. The transformations introduced (projections, extensions and removal of inconsistent values) assure optimality and termination in the resulting algorithms.

According to experimental results, combining AC and FDAC levels of soft arc consistency with the distributed search algorithm BnB-ADOPT$^+$ (BnB-ADOPT$^+$-AC/FDAC) provides substantial benefits in distributed search for the benchmarks tested. New messages DEL and UCO have been introduced to inform deletions and cost extensions respectively. However, the increment in the number of messages due to the generation of new DEL and UCO messages has been largely compensated by a decrement in the number of COST and VALUE messages used to solve the problem. The propagation of deletions contribute to diminish the search effort, decreasing the total number of messages exchanged. Also, the flows of costs in the constraint network, implemented by UCO messages, allows an agent to pass some of their unary costs to other agents, increasing pruning opportunities. In general, maintaining AC and FDAC reduces substantially the number of messages required to reach the optimal solution, reducing also the number of cycles and the computational effort at each agent.

We have not been able to achieve the next soft arc consistency level EDAC because, in its direct form, it enters in conflict with usual privacy requirements. As alternative, we propose to maintain FDAC in multiple representations and DAC by token passing.

In FDAC with multiple representations, agents maintain several orderings among variables and enforce FDAC on every ordering. By doing this, inconsistent values can be detected in any of the ordering causing more pruning opportunities. Experimental results show significant savings in communication and a higher effort in computation, since more work must be done to maintain FDAC in every ordering.

In DAC by token passing, agents receiving a token ask neighbors for cost extensions. After receiving cost extensions from all neighbors, if a deletion occurs or a C_ϕ increment occurs in

an agent, the agent passes the token to its neighbors. With this strategy, an agent receives cost extensions from all its neighbors –while in FDAC agents only receive extensions from lower neighbors in the pseudo-tree. DAC by token passing, which does not maintain any soft local consistency property, turned out to be competitive when compared with FDAC (combined with the solving algorithm BnB-ADOPT$^+$) in the benchmarks tested, considering communication and computational effort.

5

Distributed Soft Global Constraints

In centralized problems, global constraints have been essential for the advancement of constraint reasoning. In this Chapter we propose to include soft global constraints in distributed constraint optimization problems. For this, we study possible decompositions of global constraints and their inclusion in DCOPs. We extend the distributed search algorithm BnB-ADOPT$^+$ to support these representations of global constraints. In addition, we explore the relation of global constraints with soft arc consistency in DCOPs, in particular for the generalized soft arc consistency (GAC) level, including specific propagators for some well-known soft global constraints.

5.1 Soft Global Constraints in Distributed Constraint Optimization

In this Chapter we consider the inclusion of soft global constraints in DCOPs. To the best of our knowledge, no relation between DCOPs and soft global constraints have been established so far. For the distributed context, we proposed an initial work for including global constraints in Distributed Constraint Satisfaction. This work can be consulted in Appendix B. However in this thesis we focus in the optimization case.

In centralized, soft global constraints have been defined as a class of soft constraints with non-fixed arity. For instance, *soft-alldifferent*(x_1, x_2, x_3) and *soft-alldifferent*(x_1, x_4, x_5, x_6) are two instances of the *soft-alldifferent* global constraint class. The cost of a value assignment in a soft global constraint is evaluated using the violation measure μ and this violation measure may vary for each soft global constraint.

Interesting classes of soft global constraints are those which are contractible, binary decomposable or decomposable with extra variables (for details, see Chapter 2). A first motivation to include soft global constraints in the distributed context is as follows. In DCOPs it is a common assumption that cost functions are binary, that is, defined over two variables. Most soft global constraints are not binary decomposable, so working with their original soft global formulations is really needed if one wants to achieve full constraint expressivity.

Also, some soft global constraints can be semantically decomposed in a finite number of fixed-arity constraints (binary decomposition in some cases, higher arity decomposition using extra variables in others). Although these decompositions preserve the same set of solutions, they might not preserve the same level of soft arc consistency. It is known that when applying soft arc consistency to some global constraints, the quality of the bounds obtained is better than when working with an equivalent decomposition. This is the case of the *soft-alldifferent* global constraint (for example see Figure 2.4 in Chapter 2). Also, local consistencies can be enforced much more efficiently using specific propagators that exploit the semantic of a particular global constraint than using generic propagators. Thus, we also explore in this Chapter the relation of soft global constraints with soft arc consistency in DCOPs.

In particular, we work with the solving algorithm BnB-ADOPT$^+$ and with the generalized soft arc consistency (GAC) level. Regarding soft global constraints, we consider *soft-alldifferent* and *soft-at-most[k,v]* global constraints. As explain in Chapter 2, they hold the following properties:

- *soft-all-different*(T) with μ_{dec}. It is contractible and binary decomposable
- *soft-all-different*(T) with μ_{var} It is contractible but not binary decomposable.
- *soft-at-most[k,v]*(T) with μ_{var}. It is contractible but not binary decomposable. It allows decomposition with extra variables in a polynomial number of constraints of arity 3.

In the following we analyze different ways to introduce soft global constraints in DCOP. This depends heavily on the characteristic of the considered soft global constraint. Specifically, we consider whether the soft global constraint is binary decomposable, contractible or decomposable with extra variables. For each case we propose a way to model this constraints in DCOP. When the soft global constraint contains more than one property (for example if it is contractible and is also binary decomposable) a user can decide which model to choose. In Section 5.2.3, we provide empirical results comparing the different models on different benchmarks.

5.1.1 Binary Decomposable Soft Global Constraints

As previously mentioned, some global constraints are decomposable in a set of binary constraints on the variables of the global constraint. For example, the *soft-alldifferent(T)* global constraint with the violation measure μ_{dec} is semantically equivalent to a clique connecting all variables in T where nodes are the variables and edges are binary cost functions. These binary cost functions assign a cost of 0 if the involved variables take different values and a cost of 1 if they take the same value.

Including the binary decomposition of a soft global constraint in a distributed problem does not cause extra difficulties in most DCOP solving algorithms (some soft binary constraints are simply added to the problem and are treated as any other soft constraint). Figure 5.1 shows the decomposition of *soft-alldifferent* into a clique of soft binary constraints.

5.1.2 Decomposition with Extra Variables

In the satisfaction case, there are global constraints that are not binary decomposable but they can be decomposed in a polynomial number of smaller, fixed arity constraints (Bessiere and Hentenryck, 2003; Bessiere et al., 2008), if we allow a polynomial number of *extra variables*. For example, the hard $atmost[k, v](y_1, ..., y_p)$ global constraint establishes that value v cannot

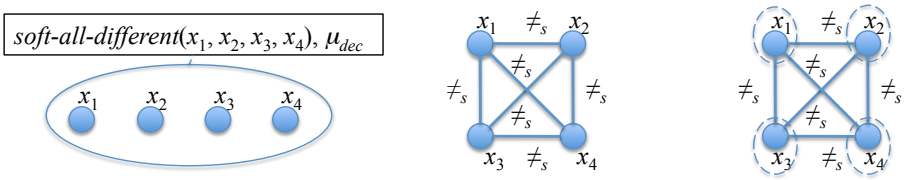


Figure 5.1: *Left*: The *soft-alldifferent*(x_1, x_2, x_3, x_4) global constraint with the μ_{dec} violation measure. *Center*: Its binary decomposition, $\neq_s$ stands for soft binary constraints. *Right*: Binary decomposition in DCOP; agents are represented with discontinuous lines.

appear more than k times in $\{y_1, ..., y_p\}$. Allowing $p + 1$ extra variables $\{z_0, z_1, ..., z_p\}$ with domains $D_{z_j} = \{0, 1, ..., j\}$, p new ternary constraints:

$$\text{if } y_i = v \text{ then } z_i = z_{i-1} + 1 \text{ else } z_i = z_{i-1} \qquad i : 1, ..., p$$

and one unary constraint:

$$z_p \leq k$$

It is easy to see that the original constraint is semantically equivalent to this set of new constraints. Variables $\{z_0, z_1, ..., z_p\}$ are acting as counters: z_i contains the number of times value v appears in variables $y_1, ..., y_i$. Variables $\{z_0, z_1, ..., z_p\}$ are called extra variables because they are not present in the original problem definition. However, they are treated as any other problem variable.

Passing to the soft case, the *soft-atmost*$[k, v](y_1, ..., y_p)$ has the following meaning: if value v appears less than or k times in the set $\{y_1, ..., y_p\}$ that assignment costs 0, otherwise it costs the number of times v appears minus k. This soft constraint can be decomposed with extra variables as follows. We keep the same extra variables as in the hard decomposition $\{z_0, z_1, ..., z_p\}$ with the same domains $D_{z_j} = \{0, 1, ..., j\}$. Previous p ternary constraints are defined as follows: tuples satisfying the condition have cost 0 and the remaining tuples have cost ∞. In addition, the unary constraint becomes:

$$\text{if } z_p \leq k \text{ then cost} = 0 \text{ else cost} = z_p - k$$

In tabular form with μ as cost, $\forall c \in D_z$ and $\forall d \in D_y$, each ternary constraint generates a table similar to the following table on the left, while the unary constraint generates a table similar to the following table on the right:

z_{i-1}	y_i	z_i	μ
c	$d = v$	$c+1$	0
c	$d \neq v$	c	0
	otherwise		∞

z_p	μ
$\leq k$	0
$> k$	$z_p - k$

The proposed decomposition appears in Figure 5.2.

Allowing extra variables in DCOP, a question naturally follows: which agent owns these extra variables, which have no real existence? To solve this issue we propose to add a number of *virtual agents* that own these extra variables. While this approach allows to keep the assumption that each agent owns a single variable, a new issue appears on the existence and activity of virtual agents with respect to real agents. Previous approaches have used the idea of virtual agents to accommodate modifications or extensions that deviate from original problem structure (Modi et al., 2005). In this case, all variables are treated in the same way, one variable per agent, so no preference is given to a particular subset of variables in front of others. DCOP solving algorithms do not need to make a distinction between virtual and not virtual agents.

Virtual agents could also be simulated by real agents. If some real agents have substantial computational/communication resources, they can host some virtual agents. The precise allocation of virtual agents would depend on the nature of the particular application to solve.

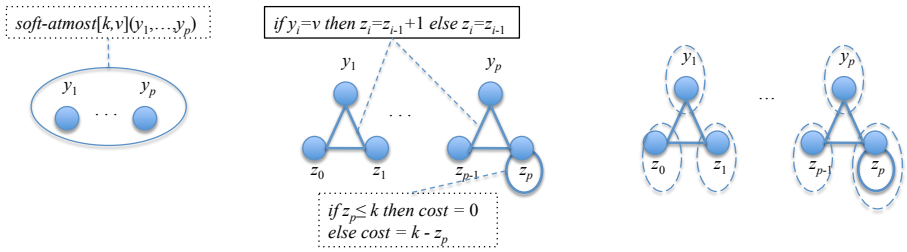


Figure 5.2: *Left*: The *soft-atmost*$[k, v](y_1, ..., y_p)$ soft global constraint. *Center*: Its decomposition in p ternary and one unary constraint. *Right*: This decomposition in the distributed context; agents are represented with discontinuous lines.

5.1.3 Contractible Soft Global Constraints

If a soft global constraint C is contractible, then C allows a *nested representation*. The nested representation of $C(T)$ with $T = \{x_{i_1}, \ldots, x_{i_p}\}$ is the set of constraints $\{C(x_{i_1}, \ldots, x_{i_j})$ with $j \in 2 \ldots p\}$. For instance, the nested representation of *soft-alldifferent*(x_1, x_2, x_3, x_4) is the set $S = \{$*soft-alldifferent*(x_1, x_2), *soft-alldifferent*(x_1, x_2, x_3), *soft-alldifferent* $(x_1, x_2, x_3, x_4)\}$.

The nested representation has the following benefit. Since x_2, x_3 and x_4 are respectively the last agent of a constraint in S, any of them is able to evaluate that particular constraint. When assignments are made following the order x_1, x_2, x_3, x_4, every intermediate agent is able to aggregate costs and calculate a lower bound of the current partial solution. Since C is contractible, this bound increases monotonically on every agent. By this, it is possible to calculate updated lower bounds during search and backtrack earlier if the current solution has unacceptable cost.

5.1.4 Including Soft Global Constraints in Distributed Problems

We consider several ways to model the inclusion of a soft global constraint in DCOPs, looking for the one that offers the best performance. The user chooses one of the following representations and the solving is done on that representation. We assume that agents are ordered. The evaluation of a global constraint $C(T)$ by every agent depends on the selected model. We analyze the three following representations:

- Direct representation. C is treated as a generic constraint of arity $|T|$. Only one agent involved in the constraint evaluates it: the one that appears last in the ordering.
- Nested representation. If $C(T)$ is contractible, each agent *self* involved in T includes the constraint in which *self* is the last evaluator. For example, for the *soft-alldifferent*(x_1, x_2, x_3), agent x_2 evaluates *soft-alldifferent*(x_1, x_2) and agent x_3 evaluates *soft-alldifferent*(x_1, x_2, x_3). Aggregation of costs from these evaluations must be done in such a way that costs are not counted twice, since more than one agent might evaluate the constraint.
- Bounded arity representation. If C is binary decomposable without extra variables, each agent $self$ includes all constraints of the binary decomposition of C that involve x_{self} in their scope. Otherwise, if C is decomposable with extra variables, new virtual agents are added to the problem (one per every extra variable required). Each agent $self$ includes all constraints of the decomposition of C that involve x_{self} in their scope. Unlike previous representations (direct and nested), in this case the constraints included by *self* are non-global.

Since the nested representation allows to calculate updated bounds and performs efficient backtracking, it is expected to be more efficient than the direct representation. However not all global constraints are contractible, so the direct representation has to be analyzed.

In a binary decomposable representation every agent is evaluator, as in the nested one. Binary decompositions may require more messages than the nested representation to accumulate costs in an agent since an agent *self* only knows some constraints of the binary decomposition (the ones in which *self* is involved), so calculating costs on one agent may require more than one step.

5.2 Including Soft Global Constraints in BnB-ADOPT$^+$

In this Section we modify the solving algorithm BnB-ADOPT$^+$ to include soft global constraints. In this way, we illustrate how to include the three different representations of global constraints (direct, nested and binary representation) in DCOPs. As a proof of concept, we implemented the inclusion of two soft global constraints –*soft-alldifferent* and *soft-atmost*– in connection with BnB-ADOPT$^+$ including also GAC propagation as explained in 5.2.2.

5.2.1 Searching with BnB-ADOPT$^+$

As detailed in Chapter 3, BnB-ADOPT$^+$ can handle constraints of any arity. We assume that our version of BnB-ADOPT$^+$ includes this generalization and that it is able to handle fixed arity constraints (from now on, we also call it non-global constraints).

In order to extend BnB-ADOPT$^+$ to support global constraints the following modifications are needed:

1. Every agent *self* keeps a set of global constraints, separated from the set of non-global constraints it is involved. Agent *self* knows about a constraint $C(T)$ iff *self* is in T. *self* also knows about the other agents involved in T (neighbors of *self*). For some global constraints, additional information can be stored. For example, for the *soft-atmost*$[k, v]$ constraint, parameters k (number of repetitions) and v (value) are stored.

2. During the search process, every time *self* needs to evaluate the cost of a given value v, all local costs are aggregated. Non-global constraints are evaluated as usual, and global constraints are evaluated according to their violation measure.

3. VALUE messages are sent to agents, depending on the constraint type:

   - For a non-global constraint (this includes binary decompositions and decompositions with extra variables), VALUE messages are sent to all the children and the

last pseudo-child in the ordering (the deepest agent in the DFS tree involved in the constraint evaluates it; VALUE messages to children are needed because they include a threshold required in BnB-ADOPT; observe that for binary constraints this is the original BnB-ADOPT behavior).

- For a global constraint, there are two options:
  - For the direct representation, VALUE messages are sent to all the children and the last pseudo-child in the ordering (the deepest agent in the DFS tree involved the constraint evaluates it; VALUE messages to children are needed because they include a threshold required in BnB-ADOPT). [1]
  - For the nested representation, VALUE messages are sent to all children and all pseudo-children (any child or pseudo-child is able to evaluate a constraint of the nested representation).

4. COST messages include a list of all the agents that have evaluated a global constraint. This is done to prevent duplication of costs when using the nested representation, as explained next. For example, consider a soft global constraint *soft-alldifferent*(x_1, x_2, x_3) with violation measure μ_{var} where all agents assign value a and send VALUE messages, as in Figure 5.3, left. When VALUE messages arrive, in the nested representation both agents x_2 and x_3 evaluate the constraint (x_2 evaluates *soft-alldifferent*(x_1, x_2) while x_3 evaluates *soft-alldifferent*(x_1, x_2, x_3)) and send COST messages as in Figure 5.3, center. When the COST messages arrive to x_1 and x_2 lower bounds are updated. Observe that, after receiving the COST message, if x_2 simply evaluates again the global constraint duplication of costs occur and x_2 sends a COST message with LB = 3 (Figure 5.3, right, this is incorrect) instead of LB =2, which is the correct evaluation. This is because x_3 evaluation already contained the evaluation of x_2. To prevent incorrect duplication of costs, before evaluating a soft global constraint agents must first check if some descendant has already evaluated it.

Figure 5.4 shows the pseudocode for cost aggregation in BnB-ADOPT$^+$. Costs coming from non-global constraints are calculated as usual and costs coming from global constraints are calculated according to its violation measure. A short description follows:

[1] In distributed search, a global constraint in the direct representation has the same treatment as a non-global one. However, when GAC is enforced, global and non-global constraints are treated differently (Section 5.2.2).

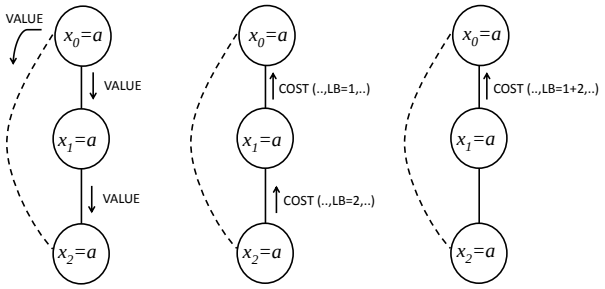


Figure 5.3: *Left*: The *soft-aldifferent*(x_1, x_2, x_3) soft global constraint with violation measure μ_{var} and all variables assigning value a. *Center*: Agent x_2 and x_3 evaluate the soft global constraint. *Right*: Agent x_2 evaluates incorrectly the soft global constraint and duplication of costs occur.

```
01 function CalculateCost(value)
02   cost = cost + NonGlobalCostWithValue(value);
03   cost = cost + GlobalCostWithValue(value);
04   return cost;

05 function CalculateNonGlobalCost(value)
06   cost = 0;
07   for each nonGlobal ∈ nonGlobalConstraintSet do
08     assignments = new list(); assignments.add(self, value);
09     for each (x_i, d_i) ∈ context do
10       if x_i ∈ nonGlobal.vars then assignmetns.add(x_i, d_i);
11     if assignments.size == nonGlobal.vars.size then        //self is the last evaluator
12       cost = cost + nonGlobal.Evaluate(assignments);
13   return cost;

14   function CalculateGlobalCost(value)
15     cost = 0;
16     for each global ∈ globalConstraintSet do
17       assignments = new list(); assignments.add(self, value);
18       for each (x_i, d_i) ∈ context do
19         if x_i ∈ global.vars then asignments.add(x_i, d_i);
20       if assignments.size == global.vars.size then          //self is the last evaluator
21         cost = cost + global.μ.Evaluate(assignments);
22       else                                  //self is an intermediate agent in the restriction
23         if NESTED representation then
24           for each x_i ∈ global.vars do
25             if lowerGlobalEvaluators.contain(x_i) then cost = cost + 0;
26             else cost = cost + global.μ.Evaluate(assignments);
27     return cost;
```

Figure 5.4: Pseudocode: Aggregating costs of binary and global cost functions.

- `CalculateCost`($value$). Calculates the cost of *self* assigning $value$. It aggregates non-global (line 2) and global (line 3) cost functions. Although there is no need to separate non-global from global cost aggregation, we have presented them in separate procedures for a better understanding of the new modifications.

- `CalculateNonGlobalCost`($value$). Calculates the cost of *self* assigning $value$ aggregating only non-global cost functions. If *self* is the evaluator (lowest agent in the pseudo-tree, line 11) it calculates the cost of assigning $value$ given the current context $context$. Otherwise the returned cost is zero.

- `CalculateGlobalCost`($value$). Calculates the cost of *self* assigning $value$ aggregating only global cost functions. For every global constraint $C(T)$ in which *self* is involved it creates a tuple with the assignments of the agents in T in the current context (lines 17-19). If *self* is the deepest agent in the DFS tree (taking into account the variables involved in the global constraint) then *self* evaluates the constraint (lines 20-21). If *self* is an intermediate agent, it does the following. If representation is direct, *self* cannot evaluate the global constraint: it does nothing and cost remains unchanged. If representation is nested, it requires some care (lines 23-26). A nested global constraint could be evaluated more than once by intermediate agents and if these costs were simply aggregated duplication of costs may occur. To prevent this, COST messages include the set of agents that have evaluated global constraints (*lowerGlobalEvaluators*). When a COST message arrives, $self$ knows which agents have evaluated their global constraints and contributed to the lower bound. If some of them appear in the scope of C, then $self$ does not evaluate C (lines 25-26). By doing this, the deepest agent in the DFS tree evaluating the global constraint precludes any other agent in the same branch to evaluate the constraint, avoiding cost duplication. Preference is given to the deepest agent because it is the one that receives more value assignments and can perform a more informed evaluation. When bounds coming from a branch of the DFS are reinitialized (this happens under certain conditions in BnB-ADOPT, for details see (Yeoh et al., 2010)), the agents in the set *lowerGlobalEvaluators* lying on that branch are removed.

5.2.2 Propagation with BnB-ADOPT$^+$

We maintain unconditional GAC (from now on, we call it simply GAC) combined with BnB-ADOPT$^+$ projecting costs from non-global/global cost functions to unary cost functions and

projecting unary costs to C_ϕ. After projections are performed, agents check their domains searching for inconsistent values. If values are deleted in an agent they are propagated and GAC is reinforced on neighbors.

Following the same technique proposed in Section 4.4, we maintain GAC during search performing only unconditional deletions. For this, some modifications are needed:

- The domain of neighboring agents are represented in *self*.
- Extra information needed to discover inconsistent values are propagated among the agents, such as: $\top$, the global C_ϕ and projections to C_ϕ from agents. Specifically in BnB-ADOPT$^+$, this information travels in VALUE and COST messages (see Figure 4.3).
- A new DEL message is added to notify value deletions to neighbors.

To reach the GAC level agents need to project costs not only from fixed arity cost functions, but from global cost functions as well. Projections are expensive operations. They require to find the minimum value among all combinations of domain values in a cost function and afterwards to modify that cost function. The cost of performing such operations grows exponentially when the number of variables involved in the constraint increase.

Specific propagators exploiting the semantics of global constraints have been proposed in the centralized case (Lee and Leung, 2009). These propagators allow to achieve GAC in polynomial time whereas a generic generic propagator is exponential in the number of variables in the scope of the constraint. In the following, we describe how to project costs specifically from the *soft-alldifferent* and *soft-atmost* global constraints using specific propagators.

Projecting costs with bounded arity constraints. The projection of costs from cost function $C(T)$ to the unary cost function $C_{x_i}(a)$, where T is a fixed set of variables, $x_i \in T$ and $a \in D_{x_i}$ is a flow of costs defined as follows.

Let α_v be the minimum cost in the set of tuples of $C(T)$ where $x_i = a$ (namely $\alpha_a = min_{t \in tuples\, s.t.\, x_i = a} C_T(t)$). The projection consists in adding α_a to $C_{x_i}(a)$ (namely, $C_{x_i}(a) = C_{x_i}(a) + \alpha_a, \forall a \in D_{x_i}$) and subtracting α_a from $C_T(t)$ (namely, $C_T(t) = C_T(t) - \alpha_a, \forall t \in tuples\, s.t.\, x_i = a, \forall a \in D_{x_i}$).

Projections are performed in every agent in T following a fixed order. Agents are ordered according to their position in the pseudo-tree and projections are done in that order over every agent in T. As in the binary case, cost functions of the *self* agent are updated, but unary cost

```
01  procedure ProjectFromAllDiffToUnary(global, v)
02  graph = graphsSet.get(global);                    //the graph associated with global is fetched
03  minCost = minCost + getMinCostFlow(graph);
04  for each x_i ∈ global.vars do
05    minCost = minCost + CostWithFlow(graph, x_i, v);
06    if minCost > 0 then
07      graph.getArc(x_i, v).cost = cost − minCost;
08      if x_i = self then C_self(v) = C_self(v) + minCost;

09  function getMinCostFlow(graph)
10    graph.SuccesiveShortestPath();
11    minCostFlow = 0;
12    for each arc ∈ graph.arcs do
13      minCostFlow = minCostFlow + (arc.flow ∗ arc.cost);
14    return minCostFlow;

15  function CostWithFlow(graph, x_i, v)
16     if graph.arc(x_i, v).flow = 0 mythen return 0;
17     path = graph.residualGraph.FindShortestPath(x_i, v);//shortest path from v to x_i in the residual graph
18    flow = ∞; cost = 0;                                //calculate flow as the minimum capacity in this path
19    for each arc ∈ path do
20      if arc.capacity < flow then flow = arc.capacity;
21    for each arc ∈ path do
22      cost = flow ∗ arc.cost;
23    return cost;
```

Figure 5.5: Pseudocode: Projection with *soft-alldifferent* global constraint.

functions of neighbors are not updated.

Projecting costs with *soft-alldifferent*. We follow the approach described in (Lee and Leung, 2009) for the centralized case, where GAC is enforced on the *soft-alldifferent* constraint in polynomial time, whereas it is exponential when a generic algorithm is used.

A graph for every *soft-alldifferent* constraint is constructed following (van Hoeve et al., 2006). This graph is stored by the agent and updated during execution. Every time a projection operation is required, instead of exhaustively looking at all tuples of the global constraint, the minimum cost that can be projected is computed as the flow of minimum cost of the graph associated with the constraint (Lee and Leung, 2009). Minimum flow cost computation is based on the successive shortest path algorithm, which searches shortest paths in the graph until no more flows can be added to the graph. Pseudocode appears in Figure 5.5.

Evaluation of these propagators in the distributed context is an extra issue because it is not based on table look-ups. In the centralized case, they are usually evaluated by their CPU time.

```
1   procedure ProjectFromAtMostToUnary(global, v)
2   if global.v = v and D_self.contains(v) then
3     singletonCounter = 0; cost = 0;
4     for each x_i ∈ global.vars do
5       if D_{x_i}.contains(v) and D_{x_i}.size() = 1 then
6         singletonCounter = singletonCounter + 1;
7     if singletonCounter > global.k then
8       cost = singletonCounter − global.k;
9       if cost > global.projectedCost then
10        temp = cost;
11        cost = cost − global.projectedCost;
12        global.projectedCost = temp;
13        if global.vars[0] = self then C_self(v) = C_self(v) + cost;
```

Figure 5.6: Pseudocode: Projection with *soft-atmost*$[k, v]$ global constraint.

An evaluation proposal appears in Section 5.2.3.

Projecting costs with *soft-atmost*. For the *soft-atmost* global constraint we propose the following technique to project costs from the global constraint *soft-atmost*$[k, v](T)$ to the unary cost functions $C_{x_i}(v)$. Agent x_i counts how many agents in T have a singleton domain $\{v\}$. If the number of singleton domains $\{v\}$ is greater than k, a minimum cost equal to the number of singleton domains $\{v\}$ minus k can be added to the unary cost $C_{x_i}(v)$ in one of the agents of the global constraint. We always project on the first agent of the constraint (we choose the first agent because in case of value deletion the search space reduction is larger). To maintain equivalence, the *soft-atmost* constraint stores this cost, that will be decremented from any future projection performed. Pseudocode appears in Figure 5.6.

5.2.3 Experimental Results

To evaluate the impact of including soft global constraints, we tested on several random DCOPs sets including *soft-alldifferent* and *soft-atmost* global constraints. Experiments considers binary random DCOPs with 10 variables and domain size of 5. The number of binary cost functions is $n(n-1)/2 * p_1$, where n is the number of variables and p_1 varies in the range $[0.2, 0.9]$ in steps of 0.1. Binary costs are selected from an uniform cost distribution. Two types of binary cost functions are used, cheap and expensive. Cheap cost functions extract costs from the set $\{0, ..., 10\}$ while expensive ones extract costs from the set $\{0, ..., 1000\}$. The proportion of expensive cost functions is 1/4 of the total number of binary cost functions. In addition

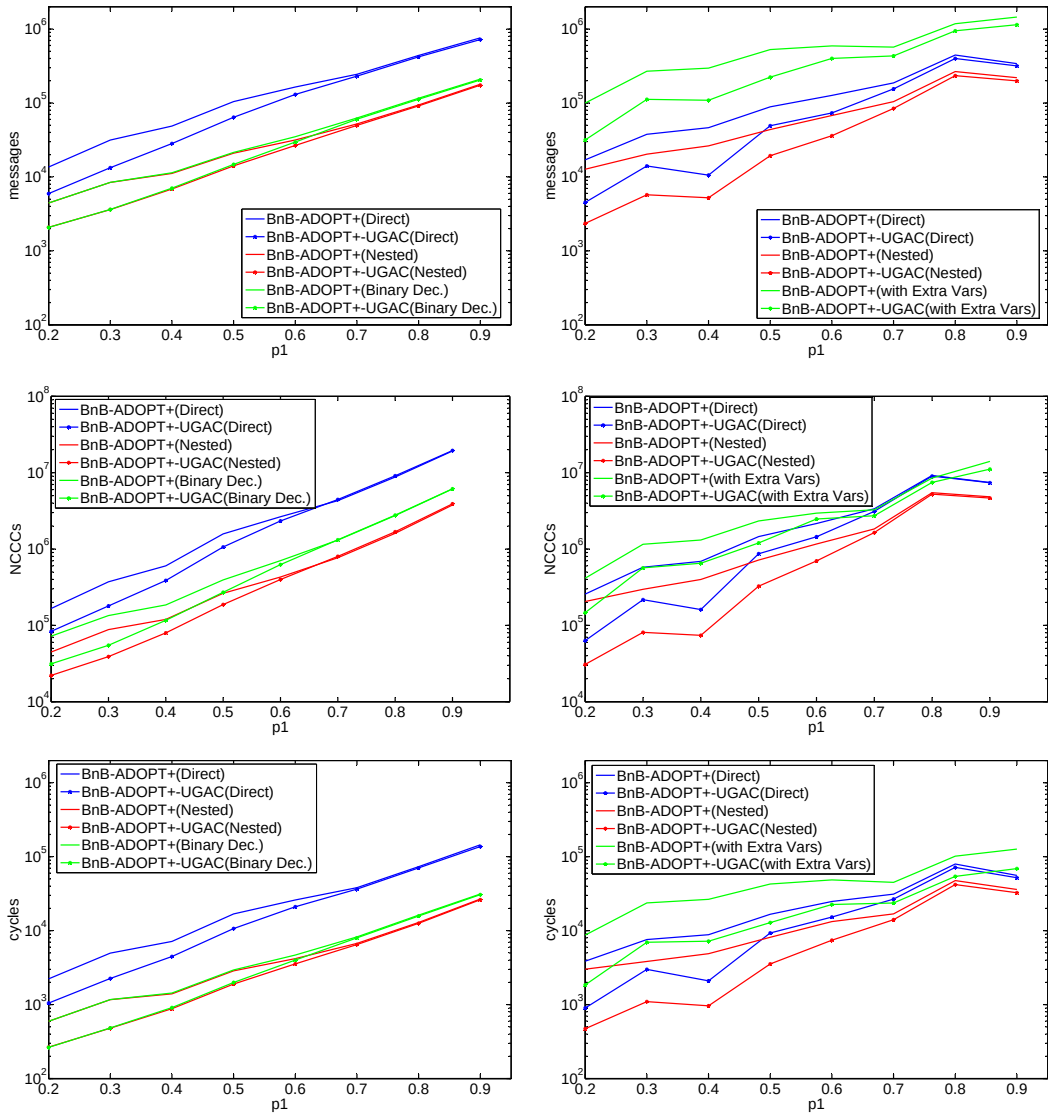


Figure 5.7: Experimental results of random DCOPs including (*left*) *soft-alldifferent* global constraints with the violation measure μ_{dec}; (*right*) *soft-atmost* global constraints with the violation measure μ_{var}.

to binary constraints, global constraints are included. The first set of experiments includes 2 *soft-alldifferent*(T) global constraints in every instance, where T is a set of 5 randomly chosen variables. The violation measure is μ_{dec}. The second set of experiments includes 2 *soft-atmost[k,v]*(T) global constraints in every instance, where T is a set of 5 randomly chosen variables, k (number of repetitions) is randomly chose from the set $\{0, ..., 3\}$ and v (value) is

randomly selected from the variable domain. The violation measure used is μ_{var}. To balance binary and global costs, the cost of the *soft-alldifferent* and *soft-atmost* constraints is calculated as the amount of the violation measure multiplied by 1000.

We tested the extended versions of BnB-ADOPT$^+$ and BnB-ADOPT$^+$-GAC able to handle global constraints on these benchmarks, results appear on Figure 5.7. Computational effort is evaluated in terms of non-concurrent constraint checks (NCCCs) and communication effort is evaluated in terms of the number of messages exchanged. We also present as a reference measure the number of synchronous cycles. For *soft-alldifferent* with μ_{dec} we present results of its direct representation, its nested representations and its binary decomposition. For *soft-atmost* with μ_{var} we present results of its direct representation, its nested representation and its decomposition with extra variables.

Specifically for GAC enforcement, computational effort is measured as follows. For the sets including *soft-alldifferent* global constraints, we use the special propagator proposed in (Lee and Leung, 2009). Every time a projection operation is required, instead of exhaustively looking at all tuples of the global constraint (which would increment the NCCC counter for every tuple), we compute the minimum flow of this graph. Minimum flow cost computation is based on the successive shortest path algorithm (we can think of every shortest path computation as a variable assignment of the global constraint). We assess the computational effort of computing the shortest path algorithm as the number of nodes of the graph where this algorithm is executed (looks reasonable for small graphs, which is the case here). Each time the successive shortest path algorithm is executed, we add this number to the NCCC counter of the agent.

For the sets including the *soft-atmost* global constraints, every time the cost of the violation measure is computed as the number of singleton domains $\{v\}$ minus k, the NCCC counter is incremented.

From results in Figure 5.7, we observe the following facts. The nested representation is better than the direct one in terms of messages, NCCCs and cycles, for both BnB-ADOPT$^+$ and BnB-ADOPT$^+$-GAC. In the direct representation, VALUE messages are sent to all children and the last pseudo-child in the global constraint, whereas in the nested representation VALUE messages are sent to all children and all pseudo-children in the global constraint. However the early detection of dead-ends in the nested representation compensates by far these extra number of messages.

It is of special interest to observe the behavior of the bounded arity decompositions. We observe that the nested representation is consistently better (in terms of messages and NCCCs) than the binary decomposition (in *soft-alldifferent)* and than the decomposition with extra variables (in *soft-atmost*). If an agent changes value, it will send the same number of VALUE messages in the nested representation as in the clique of binary constraints of the *soft-alldifferent* decomposition. However, in the nested representation receivers will evaluate larger constraints (with arity greater than 2), so they are more effective and as global effect this representation requires less messages than the binary decomposition. When GAC is maintained, the number of value deletions using the nested representation is higher than using the binary decomposition (because pruning using global constraints is more powerful than when using the binary decomposition); as consequence, the search space is slightly smaller when using the nested representation, and due to this, less messages are required for its complete exploration. Considering the decomposition with extra variables in *soft-atmost*, this decomposition includes new extra variables in the problem, causing many extra messages. These messages lead to more computational effort (more NCCCs).

Considering GAC enforcement, we observe that it always pays off in terms of messages, for all representations. For low and medium connectivities the reduction in messages is so drastic that as result the number of NCCCs is importantly reduced. For higher connectivities, NCCCs show smaller reductions and a slight increment in the *soft-alldifferent* case. GAC requires to do more computational work at every agent, however this is balanced with a reduction in the computational effort required to process less messages during search. This important fact indicates the impact of this limited form of soft GAC maintenance in distributed constraint optimization.

From these results, we conclude that the nested decomposition generally offers the best performance in the benchmarks tested. Decomposition with extra variables using virtual agents and the direct representation are models to avoid when representing contractible global constraints, since they require a higher communication effort which impacts also in computation. Considering propagation, enforcing GAC in any of the proposed representation pays off, causing savings in communication and generally also in computation.

5.3 Conclusions

In this Chapter we have introduced the use of soft global constraints in distributed constraint optimization. We proposed several ways to represent soft global constraints in a distributed constraint network, depending on soft global constraint properties. We extended the distributed search algorithm BnB-ADOPT$^+$ to support the inclusion of global constraints and we evaluated its performance with and without the GAC consistency level (generalized arc consistency with unconditional deletions).

From this work, we can extract the following conclusions:

- The use of global constraints is necessary in distributed constraint optimization to extend DCOP expressivity,
- Considering two global constraints (*soft-at-most* and *soft-all-different*) as a proof of concept, we show:
  - If the added soft global constraint is contractible, the nested representation is the one that offers better performance both in terms of communication cost (number of messages) and computational effort (NCCCs),
  - GAC maintenance always pays off in terms of number of messages, causing also less NCCCs in a very substantial portion of the experiments.

6

Conclusions

From the work presented in this thesis, fruit of the research during 4 years on DCOPs, we extract a number of conclusions, as well as lines for further research. They are detailed in the following.

6.1 Conclusions

Considering distributed search, we have improved BnB-ADOPT, a state-of-the-art DCOP optimal solving algorithm, as follows:

- It is possible to detect redundant messages in the BnB-ADOPT algorithm and remove them from its execution without compromising optimality and termination. Removing redundant messages caused significant performance improvements in communication. We show that our improved version BnB-ADOPT$^+$ is competitive with respect to other DCOP solving algorithms.
- We observed some problems at termination and an inefficient threshold management in BnB-ADOPT when dealing with n-ary cost functions and propose some modifications to overcome them. For n-ary problems, our proposed version n-ary BnB-ADOPT$^+$ obtained important savings in communication and computational effort.
- Taking a closer look to BnB-ADOPT and ADOPT algorithms, we noticed that ADOPT provides a tie-breaking strategy that when introduced in BnB-ADOPT$^+$ produces a significant boost in performance, so we include this modification in our version of BnB-ADOPT$^+$.

- We present the new algorithm ADOPT(k), which combines the search strategies of ADOPT and BnB-ADOPT depending on the k parameter. Our experimental results show that ADOPT(k) can provide a good mechanism for balancing the trade-off between runtime and network load between ADOPT and BnB-ADOPT. Also, it was able to outperforms both algorithms on commonly used benchmarks for a certain k.

Considering soft arc consistency in DCOPs, the combination of distributed search with soft arc consistency has shown to be beneficial to improve search efficiency in several benchmarks. We have done this combination using the BnB-ADOPT$^+$ algorithm for distributed search, keeping its optimality and termination properties. From this work, we can extract the following conclusions:

- Maintaining soft arc consistency during distributed search requires a different approach than in the centralized case. While in the centralize case all problem elements are available in the solver, in the distributed case agents only know some part of the problem and must exchange information in order to achieve the desired consistency level. In this process, the operations that modify the problem structures should be done in such a way that partial information of the global problem remains coherent on every agent. Otherwise optimality in search can be compromised.

- In order to combine distributed search with soft arc consistency it would be ideal to work with an algorithm that provides lower and upper bounds of the problem solution. These bounds and their quality during execution are crucial to discover sub-optimal values. Distributed algorithms performing a branch-and-bound search strategy, such as BnB-ADOPT$^+$, are suitable to be combined with soft arc consistency since they calculate an UB and refine it with every best solution found.

- We have connected BnB-ADOPT$^+$ with some forms of soft arc consistency to detect and unconditionally delete inconsistent values, with the final goal of improving search efficiency. The transformations introduced (projections, extensions and removal of inconsistent values) assure optimality and termination in the resulting algorithms.

- According to experimental results, combining AC and FDAC levels of soft arc consistency with the distributed search algorithm BnB-ADOPT$^+$ provides important savings in distributed search in several benchmarks tested. New messages are included to exchange

new information needed to maintain soft arc consistency. However, this increment in the number of messages is largely compensated by a decrement in the number of messages used to solve the problem. Since the domain of agents is reduced, the messages needed to explore the search space also decrement.

- FDAC can be enforced on multiple representations (multiple cost functions with different orderings). As result, inconsistent values can be detected in any of the ordering causing more pruning opportunities. Experimental results show significant savings in communication and a higher effort in computation, since more work must be done to maintain FDAC in every ordering.

- Moving to higher consistency levels, agents need to have a wider knowledge about the global problem. Stronger consistency levels require agents to know more information about other agents in the problem. This may compromise privacy, which is an issue to solve.

- We propose DAC by token passing, a new way to propagate deletions during distributed search. This strategy does not maintain any soft local consistency property, so theoretically it can not be compared to AC or FDAC. However experimentally it turned out to be competitive when compared with FDAC considering communication and computational effort.

Considering soft global constraints in DCOPs, we sustain that they have to be incorporated in DCOP formulations. We propose several ways to include soft global constraints in distributed search. We report the results of a number of experiments. Specifically:

- The use of global constraints is necessary in Distributed Constraint Optimization to extend DCOP expressivity. We proposed several ways to represent soft global constraints in a distributed constraint network, depending on the soft global constraint properties.

- We extended the distributed search algorithm BnB-ADOPT$^+$ to support the inclusion of soft global constraints and we evaluated its performance with and without the unconditional GAC consistency level (generalized arc consistency with unconditional deletions).

- Considering two global constraints (*soft-at-most* and *soft-all-different*) as a proof of concept, we show: (1) If the added soft global constraint is contractible, the nested representation is the one that offers better performance in terms of communication and computational cost (2) GAC maintenance always pays off in terms of communication, causing also improvements in computation in a very substantial portion of the experiments.

6.2 Future Work

The work presented in this thesis can be extended in a number of dimensions. In general terms, we identify the following points for further research:

- Other distributed search algorithms. The connection between distributed search and soft arc consistency has been implemented using the BnB-ADOPT algorithm, but any other distributed search algorithm can be used for the same purpose. Connecting another algorithm with soft arc consistency would require to modify its messages in order to include the elements needed to achieve the desired soft arc consistency level. Very probably, the DEL and UCO messages should be implemented as described in this document.

- Propagating conditional deletions. Propagation has been limited to unconditional deletions. This decision is based on previous work on distributed constraint satisfaction, where propagating conditional deletions was often found counterproductive: the extra overhead –specially in communication terms– often overcome the benefits caused by full propagation. This conclusion was extracted from experiments on random distributed problem instances. However, it remains to be analyzed whether this holds for any DCOP instance, or on the contrary, there are DCOP applications for which propagating conditional deletions pays off.

- Higher soft arc consistency levels. We have studied the connection of distributed search with initial levels of soft arc consistency, but the connection with higher levels remains to be analyzed. It is clear that this has an impact in the degree of privacy in the DCOP resolution. In this sense, we have observed the following trend: the higher the target soft arc consistency level, the more information –in some cases private– is required to share among agents, so less private would be the solving process. Although pursuing higher soft arc consistency levels may compromise privacy, this might not be a real issue for some DCOP applications.

- More soft global constraints. From our point of view, soft global constraints have to be included in the DCOP context, and the benefits of that inclusion are clear. However, we present experimental results on two soft global constraints only. These results are valid as a proof of concept, but more work on this line is needed, considering other soft global constraints (specially given the current size of the global constraint catalog).

Considering the specific work presented in this thesis, we foresee the following possible extensions:

- Automatic adjustment of the k parameter in the ADOPT(k) algorithm. After the ADOPT(k) algorithm, a straightforward question is to be able to adjust automatically the k value (or a sequence of good k values). With this goal, several approaches come to mind. One is to better understand the behavior of the algorithm, in order to select the k value most suited for a particular DCOP instance. Another is to use machine learning techniques to adjust automatically the k value.

- Connecting BnB-ADOPT$^+$ with higher soft arc consistency levels. The connection of BnB-ADOPT$^+$ with EDAC, VAC and OSAC remains to be done. As discussed above, this has consequences on the privacy of the solving process. Regarding efficiency, these connections will increase the pruning opportunities, but at the extra cost of more communication and computation. Therefore, it has to be assessed whether connecting BnB-ADOPT$^+$ with higher consistency levels really pays off.

- Alternative heuristics for the token passing approach. The efficiency of the token passing approach strongly depends on the heuristic criterion used to pass the token among agents. This is applicable to both the preprocess –which could be repeated one or several times– and the solving process. Alternative heuristics may be devised, and their efficiency has to be assessed.

- Soft global constraint decomposition with extra variables without virtual agents. When decomposing soft global constraints with extra variables, we took the modeling decision of including these extra variables in virtual agents, backed on a number of reasons. Alternatively, these extra variables could be included in the DCOP formulation without using virtual agents. This alternative approach implies that the assumption "one variable per agent" does no longer holds, but it is possible that this would increase the efficiency

of the solving process (specially in communication terms). More experimental work is needed to properly answer this question.

We believe that the work presented in this thesis is a step forward towards finding better ways for optimally solving DCOP instances. A final test for this work would be its adaptation for solving real-word applications in the context of distributed constraint optimization.

Appendix A

Saving Messages in the ADOPT Algorithm

ADOPT is an algorithm for distributed constraint optimization solving. It exchange a large number of messages, which is a major drawback for its practical application. Aiming at increasing its efficiency, in this Appendix we present results showing that some of ADOPT messages are redundant so they can be removed without compromising its optimality and termination properties. Removing most of those redundant messages we obtain ADOPT$^+$, which in practice, causes substantial reductions in communication costs with respect to the original algorithm.

A.1 Reengineering ADOPT

In this Section we introduce our ADOPT version. It has some differences with the original ADOPT (Modi et al., 2005). We have included these changes for efficiency purposes, but they do not compromise the optimality and termination of ADOPT. Specifically, our version differs from the original ADOPT in the following points:

1. ADOPT sends THRESHOLD messages to children and VALUE messages to children and pseudo-children. Since every time ADOPT sends a THRESHOLD message it also sends a VALUE message, we include all the information in a single VALUE message. The treatment of THRESHOLD and VALUE information is exactly the same, only that it is not split in two separate messages. With this simple modification the number of messages is reduced significantly. Obviously, these changes has no effect on optimality

and termination of ADOPT. From now on, we consider VALUE and COST messages only.

2. In ADOPT, each agent reads one message of the input queue, processes it and performs backtrack. In the backtrack procedure the agent decides if it must change its value. In any case, it sends the corresponding VALUE messages to its children and pseudo-children and a COST message to its parent. This is done for each incoming message until the message queue is empty. We modify the algorithm in the following way: on each iteration, the agent reads and processes all messages from the input queue without performing backtrack, and when the queue is empty, it performs backtrack sending the corresponding VALUE and COST messages.

   This modification does not affect optimality and termination. Since all messages are processed, an agent *self* will update its data structures (context, lb, ub, th) in the same way and order as before, but it will not assign its value or generate messages until the queue is empty. So, there are messages that would have been sent by *self* in the original ADOPT that will not be send with this modification. Let us assume an omitted message msg_1 and a final message msg_2 of the same type as msg_1 sent by *self*. It may happen:

   (a) If the omitted msg_1 is a VALUE, not sending it will cause no harm, because if this VALUE message is processed, the receiver would updates in its context only *self* assignment, which will be overwritten anyway when msg_2 arrives.

   (b) If the omitted msg_1 is a COST message, the same thing happens: the last message msg_2 will overwrite the lb and ub tables that might have been updated by msg_1. Also, if the omitted msg_1 would have cause that a new variable would be added to the receiver context, we can assure that this variable will also be added with msg_2, since both messages have the same context variables.

   However, when an agent receives VALUE or COST messages, information may be reinitialized if contexts are not compatible. It could be the case that the omitted msg_1 would have caused this effect on the original execution. If this is not caused also by msg_2, then the reinitialization was useless, since the considered obsolete information is now required and will be recalculated. So we can avoid this useless reinitialization. Therefore, these changes has no effect on optimality and termination of ADOPT. This way

of processing the input queue reduces a great deal the number of exchanged messages, which is very beneficial for executing this algorithm on difficult instances.

3. Finally, we include a timestamp for every assignment that travels in VALUE messages and in COST messages contexts (one timestamp per value). This timestamp allows to determine which of two assignments is more recent. We allow the receiver context to be updated also by COST messages if they contain more recent assignments. As consequence, an agent *self* is able to process more updated COST messages instead of discarding them. On the original ADOPT, COST messages with more recent information are discarded and *self* would need to wait for delayed VALUE messages until its context is updated with the most recent information. Now, the accepted COST message contains the same information as the ones being discarded, so we could have considered them before. Observe also that if a COST message updates *self* context, then a VALUE message will eventually arrive to *self* with that same context change, including an updated threshold for this new context. Before receiving this VALUE message, the threshold is maintained between LB and UB with the `MaintainThresholdInvariant` method (Modi et al., 2005). Therefore, these changes has no effect on optimality and termination of ADOPT.

In the following, we assume an ADOPT version that includes these changes with respect to the original algorithm (Modi et al., 2005).

A.2 Communication Structure

In the following, we summarize the communication structure of our version of ADOPT. We assume that the reader has some familiarity with ADOPT code (for a more complete description, see (Modi et al., 2005)). ADOPT arranges the agents in a DFS pseudo-tree. An agent *self* knows about its parent, its pseudo-parents, its children and pseudo-children. Also, *self* holds a context, which is the set of assignments involving *self* ancestors that will be updated with message exchange.

Our ADOPT version uses the following messages:

- VALUE$(i, j, val, th, context)$: i informs child or pseudo-child j that it has taken value val with threshold th in $context$,

- COST$(k, j, context, lb, ub)$: k informs parent j that with $context$ its bound are lb and ub,
- TERMINATE(i, j): i informs child j that terminates.

An agent of our ADOPT version executes the following loop: it reads and processes all incoming messages, and changes value if the lower bound of the current value surpasses the threshold. This strategy allows the agent to change its value whenever it detects a better local assignment. Then, it sends the following messages: a VALUE message per child, a VALUE message per pseudo-child and a COST message to its parent. Every time a VALUE or COST message is sent the receiver context is updated with the more recent assignments. Every time a COST message is sent the receiver lower bound and upper bound are updated if the COST message context and the receiver context are compatible, otherwise this information is considered obsolete. Agents maintain a bounded interval consisting in a lower and upper bound that will be refined during execution. When this interval shrinks to zero (lower bound equals the upper bound) the cost of the optimal solution has been determined. ADOPT manages threshold values calculated as estimated lower bounds for every agent subtree, that allow agents to efficiently reconstruct partial solutions.

As explained earlier, our ADOPT version associates with each assignment (either traveling in VALUE or COST messages) a timestamp. This permits COST messages to update the context of receiver, if some value is more recent than the value in the receiver context. On this respect, our ADOPT version and the BnB-ADOPT algorithm act in the same way (in BnB-ADOPT timestamps are called counters, referred as ID in (Yeoh et al., 2010)).

A.3 Redundant Messages

In this Section we present the results on redundant messages in ADOPT.

The lemmas and proofs for redundant messages in ADOPT are the same as the one presented for BnB-ADOPT in Chapter 3, Section 3.1.1. Only one distinction has to be done in the use of thresholds, since BnB-ADOPT and ADOPT manage thresholds in different ways. In the case of thresholds, sent in VALUE messages, we consider the following case.

Two consecutive VALUE messages V_1 and V_2 are sent from agent i to agent j with the same val and th. Let us consider the case where V_1 is received in j and V_2 is not sent. Con-

sidering th, since th was already copied in j at V_1 reception, copying V_2 threshold is not crucial because it contains the same information. It might be the case that, between V_1 and V_2, j changes its threshold as result of some context change. Notice that, because threshold is always maintained between LB and UB, this does not compromises optimality and termination. Observe that any higher neighbor connected with j is also an ancestor of i in the pseudo-tree (because i and j are on the same branch), so i will eventually receive a message as result of this context change and then it will send a new VALUE message to j with an updated th and val. Since threshold is always maintained between LB and UB (using the `MaintainThresoldInvariant` method, Modi et al. (2005)), the termination condition ($threshold = UB$) is assured when $LB = UB$.

A.4 New Version

Temporary, we define ADOPT$^+$ as ADOPT with the following changes:

- Agents will send the THRESHOLD and VALUE information on a single VALUE message.
- On every iteration, agents will read and process all messages from the input queue without performing backtrack, and when the input queue is empty, they perform backtrack sending the corresponding VALUE and COST messages.
- Timestamps are included on every assignment, and both VALUE and COST messages may update the receiver context if the message contains a more recent assignment (with higher timestamp).
- The second of two consecutive VALUE messages with the same i, j and val is not sent.
- The second of two consecutive COST messages with the same k, j, $context$, lb and ub when k detects no context change is not sent.

Theorem 6 *ADOPT$^+$ terminates with the cost of a cost-minimal solution.*

Proof. By similar results to the ones contained in Theorems 1 and 2 for BnB-ADOPT, messages not sent by ADOPT$^+$ are redundant so they can be eliminated. ADOPT terminates with the cost of a cost-minimal solution (Modi et al., 2005), so ADOPT$^+$ also terminates with the cost of a cost-minimal solution. □

But the new algorithm is not efficient because we have not considered thresholds reinitializations. Looking for an adequate threshold management, we define ADOPT$^+$ as our ADOPT version algorithm with the following changes:

1. Agent i remembers for each neighbor j the last message sent.

2. A COST message from j to i includes a boolean $ThReq$, set to true when j threshold could not be copied –upon a VALUE message reception– because context are incompatible, or when threshold is initialized.

3. If j has to send i a COST message equal to (ignoring timestamps) the last COST message sent, the new COST message is sent if and only if j has detected a context change between them.

4. If i has to send j a VALUE message equal to (ignoring timestamps) the last VALUE message sent, the new VALUE message is sent if and only if the last COST message that i received from j had $ThReq = true$; upon reception, this VALUE message will update j threshold.

A.5 Experimental Results

We test ADOPT$^+$ algorithm on binary random DCOPs. Performance is evaluated in terms of communication cost (messages exchanged) and computation effort (non-concurrent constraint checks (Meisels et al., 2002)). We consider also the cycles as the number of iteration the simulator must perform until the solution is found.

Binary random DCOP are characterized by $\langle n, d, p_1 \rangle$, where n is the number of variables, d is the domain size and p_1 is the network connectivity. We have generated random DCOP instances: $\langle n = 10, d = 10, p_1 = 0.2, ..., 0.8 \rangle$. Costs are selected randomly from the set $\{0,..., 100\}$. Results appear in Table A.1, averaged over 50 instances.

On random DCOPs, ADOPT$^+$ showed clear benefits on communication costs with respect to our ADOPT version. It divided the number of exchanged messages by a factor from 1.1 to almost 3, maintaining the number of cycles practically constant and obtaining also moderate reductions in NCCCs.

p_1	Algorithm	#Messages	#NCCC	#Cycles
0.2	ADOPT	524	2,877	**29**
	ADOPT$^+$	**468**	**2,827**	**29**
0.3	ADOPT	1,898,519	39,870,169	**82,565**
	ADOPT$^+$	**1,003,300**	**36,330,792**	82,604
0.4	ADOPT	39,456,612	1,032,565,939	**1,461,551**
	ADOPT$^+$	**16,810,366**	**891,171,565**	1,461,910
0.5	ADOPT	410,414,194	12,187,384,497	12,826,157
	ADOPT$^+$	**145,182,982**	**10,259,021,210**	**12,811,734**

Table A.1: Results of our ADOPT version (first row) compared to ADOPT$^+$ (second row) on random DCOPs

A.6 Conclusions

We have presented two contributions to increase the performance of the ADOPT algorithm. First, we describe our version of ADOPT, which saves some messages with respect to the original algorithm. Secondly, we present theoretical results to detect redundant messages in our ADOPT version. Using these results we generate ADOPT$^+$, which caused substantial savings in communication and moderate savings in computation.

A. SAVING MESSAGES IN THE ADOPT ALGORITHM

Appendix B

Global Constraints in Distributed Constraint Satisfaction

Global constraints have been crucial in the development of efficient constraint solvers (van Hoeve and Katriel, 2006). They allow to capture global properties on an unbounded set of variables. In many cases, the exploitation of the semantic associated with a particular global constraint allows to codify propagators able to reach local consistency levels (typically generalized arc consistency, GAC) with polynomial complexity. This is a great advantage with respect to GAC propagators for generic non-binary constraints, which have complexity exponential in the constraint arity.

Often, it is implicitly assumed that distributed constraint reasoning precludes the use of global constraints. With the usual assumption that each agent contains a single variable (so the terms agents and variables can be used interchangeably), an agent knows the constraint with each one of its neighbors, and nothing else (Yokoo et al., 1998). These constraints are obviously binary. But this interpretation is too restrictive because there are distributed applications for which it is natural to use global constraints.

When adding global constraints in distributed reasoning we obtain several benefits. First, the expressivity of distributed constraint reasoning is enhanced since there are relations among several variables that cannot be expressed as a conjunction of binary relations (most global constraints are not binary decomposable).

Second, the solving process can be done more efficiently. Local consistency can be more efficiently achieved when global constraints are involved (van Hoeve and Katriel, 2006). Assuming a solving strategy maintaining some kind of local consistency, using global constraints

improves its efficiency.

Accepting the interest of global constraints in distributed constraint reasoning, another question naturally follows: since some global constraints can be decomposed in simpler constraints, is it more efficient to leave the global constraint as it was initially posted or to decompose it? If several decompositions are possible, which offers the best performance? We provide some answers to these questions, exploring two decompositions (binary (Bessiere and Hentenryck, 2003) and nested for contractible constraints (Maher, 2009b)) against the global constraint without decomposition, in two contexts: complete distributed search with and without unconditional GAC maintenance (Brito and Meseguer, 2008).

We assume that readers are familiar with constraint reasoning, specially with distributed constraint satisfaction problems (DisCSP) and the ABT algorithm (Yokoo et al., 1998).

B.1 Adding Global Constraints

A global constraint C is a class of constraints defined by a Boolean function f_C whose arity is not fixed. Constraints with different arity can be defined by the same Boolean function. For instance, *alldifferent*(x_1, x_2, x_3) and *alldifferent* (x_1, x_4, x_5, x_6) are two instances of the *alldifferent* global constraint, where $f_{\textit{alldifferent}}(T)$ returns true iff $x_i \neq x_j, \forall x_i, x_j \in T$. In the following, we write C for a global constraint, while $C(T)$ means a particular instance of that global constraint on the set of variables T.

A global constraint C is *contractible* iff for any tuple t on $x_{i_1}, \ldots, x_{i_{p+1}}$, if t satisfies $C(x_{i_1}, \ldots, x_{i_{p+1}})$ then the projection $t[x_{i_1}, \ldots, x_{i_p}]$ of t on the first p variables satisfies $C(x_{i_1}, \ldots, x_{i_p})$ (Maher, 2009b). A global constraint C is *binary decomposable without extra variables* iff for any instance $C(T)$ of C, there exists a set S of binary constraints involving only variables in T such that the solutions of S are the solutions of $C(T)$ (Bessiere and Hentenryck, 2003). In this case, S is a *binary decomposition* of $C(T)$.

In the following, we consider three different representations for a global constraint instance: *direct*, *nested* and *binary*.

- In the *direct representation*, $C(T)$ is posted in the DisCSP as a single constraint that allows all tuples on T satisfying C. In this representation, each agent in T includes $C(T)$ in its constraint set. $C(T)$ will be treated as any other constraint. That is, if agent *self* is the last agent of T in the order, it evaluates $C(T)$. If not, it puts the other agents of T in its set of neighbors.

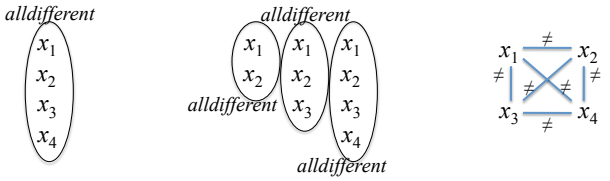


Figure B.1: Representations for *alldifferent*(x_1, x_2, x_3, x_4): (left) direct, (center) nested, (right) binary.

- The *nested representation* is applicable to all contractible global constraints. The nested representation of a global constraint $C(T)$ with $T = (x_{i_1}, \ldots, x_{i_p})$ is the set of constraints $\{C(x_{i_1}, \ldots, x_{i_j}) \mid j \in 2 \ldots p\}$. For instance, the nested representation of *all-different* (x_1, x_2, x_3, x_4) is the set $S = \{$*alldifferent*(x_1, x_2), *all-different* (x_1, x_2, x_3), *alldifferent*$(x_1, x_2, x_3, x_4)\}$. Since *alldifferent* is contractible, the set of solutions of S is exactly the same as the set of solutions of the original constraint. The idea behind this representation is to use some knowledge about the semantics of the global constraint $C(T)$ to provide a model where the handling of the constraint can be more distributed. In this representation, each agent in T includes all constraints of the nested representation of $C(T)$ that involve its variable in its constraint set. Observe that any agent in T is the last of an instance of C and it will be able to check it.

- The *binary representation* is applicable to all global constraints that are binary decomposable. The binary representation of $C(T)$ is the set of constraints of its binary decomposition. For instance, the binary representation of *alldifferent*(x_1, x_2, x_3, x_4) is the set $S = \{x_1 \neq x_2, x_1 \neq x_3, x_1 \neq x_4, x_2 \neq x_3, x_2 \neq x_4, x_3 \neq x_4\}$. Since *alldifferent* is binary decomposable, the set of solutions of S is exactly the same as the set of solutions of the original constraint. In this representation, each agent in T includes all constraints of the binary decomposition of $C(T)$ that involve its variable in its constraint set. In this case, the problem is treated as a usual binay constraint satisfaction problem.

The three representations for the *alldifferent*(x_1, x_2, x_3, x_4) global constraint appear in Figure B.1.

B.2 Searching with Global Constraints

In this Section, we use ABT as the basic search algorithm for DisCSP solving. It is worth noting that ABT –originally proposed for binary constraints– can be easily generalized to handle constraints of any arity (Brito and Meseguer, 2006). For example, let $C(x_i, x_j, x_k)$ be a ternary constraint, where the order of agents is i, j, k. Then the last agent of the constraint –agent k– is in charge of evaluating $C(x_i, x_j, x_k)$ when it is totally instantiated, while the others –agents i and j– have to send their values to k. In the following, we assume that our ABT version contains such generalization.

In the direct representation, $C(T)$ is posted in the DisCSP as a single constraint. Each agent in T includes $C(T)$ in its constraint set. The lowest priority agent of T in the ABT order is in charge of evaluating it. Other agents in T put a link between themselves and that agent. If *self* is the agent of T with lowest priority in the ABT order, it will be the one in charge of evaluating $C(T)$. If not, it puts a link between itself and each of the other agents of T.

In the nested representation, $C(T)$, $T = (x_{i_1}, \ldots, x_{i_p})$, is represented by the set of constraints $\{C(x_{i_1}, \ldots, x_{i_j}) \mid j \in 2 \ldots p\}$. Each agent in T includes all constraints of S that involve its variable in its constraint set. Thanks to the extra constraints that are posted, the checking of $C(T)$ is not postponed to the last agent in T. Any agent in T that is the last of an instance of C will be able to check it.

In the binary representation, the global constraint $C(T)$ is represented by the set of constraints of its binary decomposition. Thus, each agent in T includes all constraints of the binary decomposition of C that involve its variable in its constraint set.

These three representations of a global constraint instance are equivalent from the semantic point of view (they produce the same solutions). But they cause different ABT executions, so they can be seen as different models with dissimilar efficiency.

B.3 Propagating Global Constraints

Independently of the way a global constraint is included into ABT, this algorithm can be enhanced maintaining some form of local consistency during search. This was already investigated in (Brito and Meseguer, 2008), where limited/full forms of arc consistency (AC) were maintained during ABT execution for binary DisCSPs. While in (Brito and Meseguer, 2008) a limited form of AC causing unconditional deletions and full AC causing conditional deletions

were considered, here we only maintain the limited form of GAC that causes unconditional deletions (we enforce GAC because constraints may have arity higher than 2). Clearly this limited GAC, that from now on we call UGAC (unconditional GAC), is less powerful than full GAC. Maintaining full GAC in the distributed context would cause a substantial load of extra messages which could overcome the benefits of domain pruning. We enforce UGAC on each considered global constraint by adapting the methods developed in the centralized case to this distributed setting, making them work inside each agent.

Before search, a suitable preprocess makes the problem GAC (before search any value deletion is unconditional, so GAC is equivalent to UGAC). During search, UGAC is enforced as follows: in ABT execution, if agent *self* receives a nogood message justifying the removal of its value v where the nogood has an empty left-hand side (see (Brito and Meseguer, 2008; Yokoo et al., 1998) for details), v can be unconditionally deleted from its domain. A deletion in the domain of x_{self} is propagated maintaining UGAC on the constraints connecting x_{self} with other variables, which may cause further deletions. Since the initial deletion is unconditional, deletions caused by the propagation are also unconditional.

To maintain UGAC during ABT search, some modifications are needed over the ABT algorithm:

- The domain of variables constrained with *self* has to be represented in *self*.
- Only the agent owner of a variable can modify its domain; if agent i deduces that a value could be deleted from the domain of x_j, it does nothing because that deduction will be done by agent j at some point.
- There is a new message DEL to notify of value deletions: DEL$(self, k, v)$ –informing that *self* removes v from the domain of x_{self}– is sent from *self* to every agent k constrained with it.
- A suitable preprocess makes all constraints GAC before ABT starts. These changes do not modify ABT correctness and completeness.

B.4 Experimental Results

To evaluate the impact of the addition of global constraints, we compare ABT with and without UGAC on random DisCSP instances created as follows. We first generate random binary

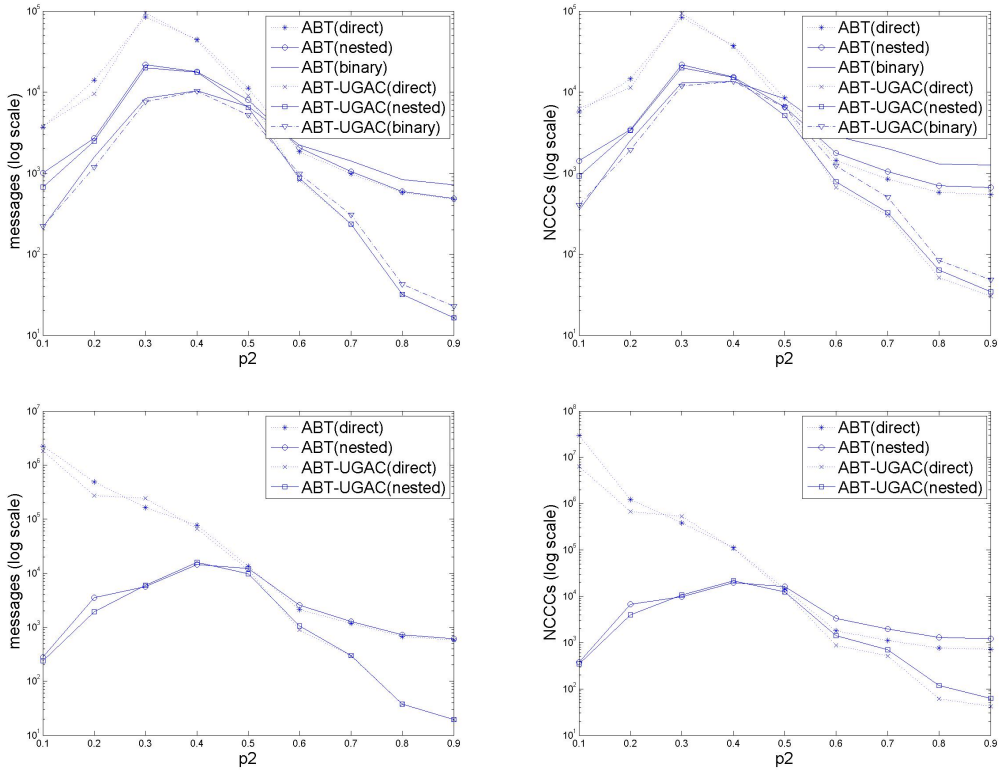


Figure B.2: Results in #messages and NCCCs for the *alldifferent* benchmark (top) and the *atmost* benchmark (bottom) described in the text. In both cases $p_1 = 0.2$.

instances and then add some global constraints. A random binary CSP class is characterized by $\langle n, d, p_1, p_2 \rangle$, where n is the number of variables, d is the domain size of each variable, p_1 is the problem connectivity defined as the ratio of existing constraints and p_2 is the constraint tightness expressed by the ratio of forbidden value pairs. Every instance contains $p_1 n(n-1)/2$ binary constraints and each of them has $p_2 d^2$ forbidden value pairs. From a binary instance, we can generate two types of benchmarks: the *alldifferent* benchmark and the *atmost* benchmark. In the *alldifferent* benchmark, each binary instance includes 2 *alldifferent* constraints, each involving 5 variables randomly chosen (we also performed this experiment with 10 –instead of 2– *alldifferent* constraints per instance, obtaining similar results). Direct, nested and binary representations are used with these *alldifferent*. In the *atmost* benchmark, each binary instance includes 10 *atmost* constraints, each involving from 3 to 10 variables randomly chosen, the value also randomly chosen and the number of repetitions between 1 and 2. Only direct and

nested representations are used on these *atmost* constraints because *atmost* is not binary decomposable.

For the *alldifferent* benchmark, we have done two sets of experiments: $\langle 20, 5, 0.2, p_2 \rangle$ and $\langle 20, 5, 0.7, p_2 \rangle$, where p_2 varies between 0.1 and 0.9 in steps of 0.1. For each, we evaluate performance as the number of messages exchanged and the number of non-concurrent constraint checks (NCCC) (Meisels et al., 2002), considering the three representations. UGAC enforcing uses generic table lookups when testing binary constraints and it executes a special propagator when testing the global *all-different* constraints (as described in (Regin, 1994)). This special propagator increments the NCCC counter each time a maximum matching is computed.

Results in number of exchanged messages for sparse problems ($p_1 = 0.2$) appear in Figure B.2 (top left), averaged on 100 instances per each p_2. For $p_2 < 0.5$ we observe that the curve of plain ABT with a particular global constraint representation follows closely the curve of ABT-UGAC with the same representation. This shows that maintaining UGAC when constraints are loose does not pay off and this is the type of representation that makes the difference in efficiency. The most efficient representation is binary, followed by nested and finally direct representation. The direct representation causes inefficient chronological backtracking (which causes many useless messages), while nested representation implies sending several OK? messages to the agents of the global constraint. In this setting where not much pruning occurs, the binary decomposition appears as the most efficient, because although it sends many OK? messages, it performs backtracking directly to the culprit.

For $p_2 > 0.5$ the situation changes and ABT curves are grouped according to UGAC enforcement: maintaining UGAC is now the most discriminant element. The analysis of this fact is simple: for medium to high tightnesses, UGAC maintenance really decreases the size of the search space, so algorithms including UGAC terminate faster and thus require less messages (much less, observe the logarithmic scale) to explore that space than plain ABT. On the relative performance of the three representations for global constraints, the less efficient representation is the binary, clearly dominated by the direct and nested ones, which practically use the same number of messages. We explain this as the combined effect of two facts: a high number of constraints (a single *alldifferent* becomes a quadratic number of constraints in binary; a linear number of constraints in nested; one constraint in direct) and the fact that the problem has no solution. In the absence of solutions, ABT necessarily generates nogoods to prove inconsistency and agents the in binary representation will tend to send a higher number of NGD

messages because they belong to more constraints than in the other representations. Some of these NGDs are useless and become obsolete.

Regarding NCCC, they appear in Figure B.2 (top right), showing a similar pattern to the #messages. Differences with/without UGAC are smaller than in #messages since enforcing UGAC causes more NCCCs. Nevertheless, for $p_2 > 0.5$ this increment is clearly compensated by the reduction of the search space, resulting in a clear decrement in the computation requirements for each agent. Differences among representations are due to the different number of constraints existing in each representation.

Results on dense problems, $p_1 = 0.7$, follow a similar pattern to the ones presented here for sparse problem results.

Considering the *atmost* benchmark, we experimented with it because *atmost* is not binary decomposable, so only the direct and nested representations are available. Evaluation was similar to the one done on the *alldifferent* benchmark: two sets of experiments: $\langle 20, 5, 0.2, p_2 \rangle$ and $\langle 20, 5, 0.7, p_2 \rangle$, where p_2 varies between 0.1 and 0.9 in steps of 0.1 counting the number of messages exchanged and the number of non-concurrent constraint checks (NCCC) on the available representations. We report results on sparse problems $p_1 = 0.2$ averaged on 100 instances per p_2.

The number of exchanged messages appears in Figure B.2 (bottom left), showing similar pattern to the one observed in the *alldifferent* benchmark. For $p_2 < 0.5$ we observe that UGAC has no much effect and the dominant factor is the representation of the global constraint: as in the *alldifferent* benchmark, the direct representation is much worse than the nested one. For $p_2 > 0.5$ the situation changes. As in the case of the *alldifferent* benchmark, maintaining UGAC becomes the main reason for efficiency, more important than the type of representation of the global constraints. Thus, ABT curves are closely related, while ABT-UGAC curves are also closely joined. The explanation of this fact is the same as in the *alldifferent* benchmark: For medium to high tightness, UGAC maintenance really decrements the size of the search space, so algorithms including UGAC terminate faster and require much less messages to explore that space than plain ABT.

Regarding NCCC, they appear in Figure B.2 (bottom right), showing a pattern similar to the #messages. Differences with/without UGAC are smaller than in #messages since enforcing UGAC causes more NCCCs. Again, for $p_2 > 0.5$ this increment is clearly compensated by the reduction of the search space. Differences among representations are due to the different number of constraints existing in each representation.

B.5 Conclusions

We have introduced the use of global constraints in distributed constraint reasoning. We have proposed three different ways to represent global constraints in a distributed constraint network, depending on whether the constraint is contractible and/or binary decomposable. We have evaluated the performance of ABT on DisCSPs both with or without unconditional GAC using different representations for global constraints.

According to experimental results, maintaining some form of local consistency is never harmful in terms of messages. It may require more effort in NCCCs, but for those instances where it significantly reduces the search space (i.e., those where the number of messages significantly decreases), the extra effort of local consistency maintenance pays off. Regarding the different representations of global constraints, the direct representation is often the less efficient one. For loose instances the binary representation wins, but for very tight instances (instances without solution) it degrades quickly, generating too many NGD messages. The nested representation seems to offer a good compromise: it is never worse than direct, and in some cases it is better than binary. This is good news because there are many more constraints that are contractible (the condition for nested representation) than constraints that are binary decomposable.

Bibliography

S. Ali, S. Koenig, and M. Tambe. Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 1041–1048, 2005. 68, 76

C. Bessiere and P. Van Hentenryck. To be or not to be ... a global constraint. In *International Conference on Principles and Practice of Constraint Programming (CP 2003)*, pages 789–794, 2003. 115, 146

C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. SLIDE: A useful special case of the CARDPATH constraint. In *European Conference on Artificial Intelligence (ECAI 2008)*, pages 475–479, 2008. 115

I. Brito and P. Meseguer. Asynchronous backtracking for non-binary DisCSP. In *Distributed Constraint Reasoning (DCR) workshop in ECAI 2006*, 2006. 148

I. Brito and P. Meseguer. Connecting ABT with arc consistency. In *International Conference on Principles and Practice of Constraint Programming (CP 2008)*, pages 387–401, 2008. 77, 146, 148, 149

A. Chechetka and K. P. Sycara. No-commitment branch and bound search for distributed constraint optimization. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006)*, pages 1427–1429, 2006. 2

M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154: 199–227, 2004. 12

M. Cooper, S. de Givry, and T. Schiex. Optimal soft arc consistency. In *International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 68–73, 2007. 12

M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual arc consistency for weighted CSP. In *AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 253–258, 2008. 3, 12, 15, 73

M. Cooper, S. de Givry, M. Sanchez, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174:449–478, 2010. 12, 15, 20

S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 84–89, 2005. 3, 12, 15, 20, 73, 100

R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999. 12, 26

R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003. 9, 10, 12

E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992. 12

A. Gershman, A. Meisels, and R. Zivan. Asynchronous forward bounding for distributed COPs. *Journal of Artificial Intelligence Research*, 34:61–88, 2009. 2, 28, 53

K. Hirayama and M. Yokoo. Distributed partial constraint satisfaction problem. In *International Conference on Principles and Practice of Constraint Programming (CP 1997)*, pages 222–236, 1997. 2, 27, 53, 57

M. Jain, M. Taylor, M. Tambe, and M. Yokoo. DCOPs meet the realworld: Exploring unknown reward matrices with applications to mobile sensor networks. In *International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 181–186, 2009. 2

R. Junges and A. L. C. Bazzan. Evaluating the performance of DCOP algorithms in a real world dynamic problem. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, pages 599–606, 2008. 2

J. Larrosa. Node and arc consistency in weighted CSP. In *AAAI Conference on Artificial Intelligence (AAAI 2002)*, pages 239–244, 2002. 15

J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted CSP. *International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 239–244, 2003. 3, 12, 15, 18, 73, 85

J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159:1–26, 2004. 12

J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107:149–163, 1999. 12

J. H. M. Lee and K. L. Leung. Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. In *International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 559–565, 2009. 17, 23, 123, 124, 127

A. K. Mackworth. Constraint satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. J. Wiley and Sons, 1991. 14

M. J. Maher. Soggy constraints: Soft open global constraints. In *International Conference on Principles and Practice of Constraint Programming (CP 2009)*, pages 584–591, 2009a. 21

M.J. Maher. Open contractible global constraints. In *International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 578–583, 2009b. 146

R. Maheswaran, M. Tambe, E. Bowring, J. Pearce, and P. Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, pages 310–317, 2004. 2, 35, 58, 70

R. Mailler and V. Lesser. Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 25: 529–576, 2006. 2

T. Matsui, M. Silaghi, K. Hirayama, M. Yokoo, and H. Matsuo. Directed soft arc consistency in pseudo trees. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 1065–1072, 2009. 76

A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Distributed Constraint Reasoning (DCR) workshop in AAMAS 2002*, pages 86–93, 2002. 34, 142, 151

P. Meseguer, F. Rossi, and T. Shiex. *Soft Constraints*, chapter 9 of Handbook of Constraint Programming, pages 281–328. Elsevier, 2006. 10, 11

S. Miller, S. D. Ramchurn, and A. Rogers. Optimal decentralised dispatch of embedded generation in the smart grid. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, pages 281–288, 2012. 2

P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005. 1, 2, 29, 31, 38, 48, 60, 65, 117, 137, 139, 141

A. Petcu. *A class of algorithms for Distributed Constraint Optimization*. PhD thesis, 2007. 24, 26

A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 266–271, 2005. 2, 26, 27

A. Petcu and B. Faltings. Distributed constraint optimization applications in power networks. *International Journal of Innovations in Energy Systems and Power*, 3, 2008. 2

T. Petit, J. C. Regin, and C. Bessiere. Specific filtering algorithms for over-constrained problems. In *International Conference on Principles and Practice of Constraint Programming (CP 2001)*, pages 451–463, 2001. 21

J. C. Regin. A filtering algorithm for constraints of difference in CSPs. In *AAAI Conference on Artificial Intelligence (AAAI 1994)*, pages 362–367, 1994. 151

F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier, 2006. 9

T. Schiex. Arc consistency for soft constraints. In *International Conference on Principles and Practice of Constraint Programming (CP 2000)*, pages 411–424, 2000. 12

N. Schurr, S. Okamoto, R. Maheswaran, P. Scerri, and M. Tambe. Evolution of a teamwork model. In R. Sun, editor, *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, pages 307–327. Cambridge University Press, 2005. 2

R. Stranders, A. Farinelli, A. Rogers, and N. R. Jennings. Decentralised coordination of mobile sensors using the Max-Sum algorithm. In *International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 299–304, 2009. 2

W. J. van Hoeve and I. Katriel. *Global Constraints*, chapter 6 of Handbook of Constraint Programming, pages 169–208. Elsevier, 2006. 145

W. J. van Hoeve, G. Pesant, and L. M. Rousseau. On global warming: flow-based soft global constraints. *Journal of Heuristics*, 12:347–373, 2006. 124

G. Verfaillie, M. Lemaitre, and T. Schiex. Russian doll search. In *AAAI Conference on Artificial Intelligence (AAAI 1996)*, pages 181–187, 1996. 12

W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010. 2, 31, 34, 38, 39, 40, 41, 46, 47, 48, 50, 51, 62, 65, 68, 78, 122, 140

Z. Yin. USC DCOP repository. 2008. 35, 53, 60, 69, 97, 103, 108

M. Yokoo, E. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998. 24, 145, 146, 149