SPEEDING UP DISTRIBUTED CONSTRAINT OPTIMIZATION SEARCH ALGORITHMS

by

William Yeoh

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

December 2010

Acknowledgements

First and foremost, I would like to thank my advisor, Sven Koenig, for his guidance and support throughout this journey, as well as the other members of my committee, Maged Dessouky, Gaurav Sukhatme, Milind Tambe and Makoto Yokoo, for their helpful comments and suggestions. I would also like to thank my colleagues and collaborators, Po-An Chen, Kenny Daniel, Ariel Felner, Alex Nash, Pradeep Varakantham, Xiaoming Zheng and Roie Zivan, for all the stimulating discussions we have had. I would like to separately acknowledge Xiaoxun Sun for the many fruitful collaborations that directly influenced my dissertation. Last but not least, I would like to thank my family and my fiancee for being so understanding and patient with me.

Table of Contents

Acknowledgements ii

List Of Tables vi

List Of Figures vii

Abstract xi

Chapter 1 Introduction 1
- 1.1 DCOP Problems . . . 2
- 1.2 Hypotheses . . . 4
- 1.3 Contributions . . . 7
- 1.4 Dissertation Structure . . . 11

Chapter 2 Background 12
- 2.1 Overview of DCOP Problems . . . 12
  - 2.1.1 Constraint Satisfaction Problems . . . 12
  - 2.1.2 DCOP Problems . . . 14
    - 2.1.2.1 Definition of DCOP Problems . . . 14
    - 2.1.2.2 Search Trees . . . 16
    - 2.1.2.3 Heuristic Values . . . 18
  - 2.1.3 DCOP Algorithms . . . 19
    - 2.1.3.1 Incomplete DCOP Algorithms . . . 19
    - 2.1.3.2 Complete DCOP Algorithms . . . 20
  - 2.1.4 DCOP Applications . . . 22
  - 2.1.5 DCOP Problem Types Used in the Experiments in this Dissertation 23
- 2.2 Overview of ADOPT . . . 24
  - 2.2.1 Properties of ADOPT . . . 25
  - 2.2.2 Search Strategy of ADOPT . . . 27
  - 2.2.3 Pre-processing Techniques for ADOPT . . . 30
- 2.3 Overview of Approaches Used to Speed Up Centralized Search Algorithms 31
  - 2.3.1 Search Strategies . . . 31
  - 2.3.2 Approximation Algorithms . . . 34
  - 2.3.3 Caching Algorithms . . . 36
  - 2.3.4 Incremental Search Algorithms . . . 37

Chapter 3 Speeding Up via Appropriate Search Strategies **40**
3.1 Motivation . . . 41
3.2 BnB-ADOPT . . . 42
3.2.1 Notations and Key Terms . . . 42
3.2.2 Updating the Bounds . . . 45
3.2.3 Adhering to the Memory Limitations . . . 50
3.2.4 Performing Depth-First Search . . . 50
3.2.5 Performing Branch-and-Bound Search . . . 59
3.2.6 Further Enhancements . . . 61
3.2.7 Pseudocode . . . 64
3.2.8 Execution Trace . . . 69
3.3 Correctness, Completeness and Complexity . . . 75
3.3.1 Correctness and Completeness . . . 75
3.3.2 Complexity . . . 89
3.4 Experimental Evaluation . . . 90
3.4.1 Metrics . . . 90
3.4.2 Problem Types . . . 92
3.4.3 Experimental Results . . . 93
3.5 Summary . . . 100

Chapter 4 Speeding Up via Approximation Mechanisms **102**
4.1 Motivation . . . 103
4.2 Approximation Mechanisms . . . 104
4.2.1 Absolute Error Mechanism . . . 105
4.2.2 Relative Error Mechanism . . . 107
4.2.3 Weighted Heuristics Mechanism . . . 108
4.3 Correctness, Completeness and Complexity . . . 110
4.3.1 Correctness and Completeness . . . 110
4.3.2 Complexity . . . 112
4.4 Experimental Evaluation . . . 113
4.4.1 Metrics . . . 113
4.4.2 Problem Types . . . 114
4.4.3 Experimental Results . . . 115
4.5 Summary . . . 121

Chapter 5 Speeding Up via Caching Schemes **123**
5.1 Motivation . . . 124
5.2 Caching . . . 125
5.2.1 Cache Design . . . 126
5.2.2 Caching Problem . . . 127
5.2.2.1 Likelihood of Future Use: P(I) . . . 128
5.2.2.2 Invested Search Effort: E(I) . . . 129
5.2.3 Caching Schemes . . . 129
5.2.3.1 Benchmark Schemes . . . 130
5.2.3.2 MaxPriority Scheme . . . 130

5.2.3.3 MaxEffort Scheme . . . 133
5.2.3.4 MaxUtility Scheme . . . 135
5.3 Correctness, Completeness and Complexity . . . 135
5.3.1 Correctness and Completeness . . . 136
5.3.2 Complexity . . . 140
5.4 Experimental Evaluation . . . 141
5.4.1 Metrics . . . 141
5.4.2 Problem Types . . . 142
5.4.3 Experimental Results . . . 143
5.4.3.1 Caching Schemes . . . 143
5.4.3.2 Caching Schemes with Approximation Mechanisms . . . 148
5.5 Summary . . . 151

Chapter 6 Speeding Up via Incremental Search 153
6.1 Motivation . . . 154
6.2 Incremental Approaches . . . 155
6.2.1 Dynamic DCOP Problems . . . 155
6.2.2 Incremental Procedure . . . 157
6.2.3 Incremental Pseudo-tree Reconstruction Algorithms . . . 164
6.2.3.1 Existing Pseudo-tree Reconstruction Algorithms . . . 165
6.2.3.2 HARP Pseudo-tree Reconstruction Algorithm . . . 166
6.3 Correctness, Completeness and Complexity . . . 175
6.3.1 Correctness and Completeness . . . 175
6.3.1.1 ReuseBounds Procedure . . . 175
6.3.1.2 HARP Pseudo-tree Reconstruction Algorithm . . . 189
6.3.2 Complexity . . . 198
6.4 Experimental Evaluation . . . 199
6.4.1 Metrics . . . 199
6.4.2 Problem Types . . . 200
6.4.3 Experimental Results . . . 202
6.5 Summary . . . 208

Chapter 7 Conclusions 210

Bibliography 215

List Of Tables

2.1 Properties of Complete DCOP Search Algorithms 25

3.1 Trace of the Updates of all Variables of BnB-ADOPT 68

List Of Figures

1.1 Example DCOP Problem 2

2.1 DCOP Problems as a Generalization and Extension of Constraint Satisfaction Problems 13

2.2 Example Pseudo-tree 15

2.3 AND/OR Search Tree 17

2.4 Taxonomy of DCOP Algorithms 19

2.5 Example Sensor Network Problem 23

2.6 Example Meeting Scheduling Problem 23

2.7 Trace of Simplified Memory-Bounded Best-First Search (Centralized ADOPT) 28

2.8 Trace of Depth-First Branch-and-Bound Search 32

3.1 Simplified Trace of the Updates of the (Lower and Upper) Bounds 47

3.2 Trace of the Updates of the Lower Bounds 54

3.3 Trace of the Updates of the Upper Bounds 55

3.4 Pseudocode of BnB-ADOPT 65

3.5 Trace of the Updates of the Lower Bounds of BnB-ADOPT 67

3.6 Trace of the Updates of the Upper Bounds of BnB-ADOPT 69

3.7 Experimental Results Comparing ADOPT, BnB-ADOPT and NCBB on Graph Coloring Problems with Constraint Costs Ranging from 0 to 10,000 94

3.8 Experimental Results Comparing ADOPT, BnB-ADOPT and NCBB on Graph Coloring Problems with 10 Vertices 95

3.9 Experimental Results Comparing ADOPT, BnB-ADOPT and NCBB on Sensor Network Problems 96

3.10 Experimental Results Comparing ADOPT, BnB-ADOPT and NCBB on Meeting Scheduling Problems 96

3.11 Experimental Results Comparing ADOPT, BnB-ADOPT and NCBB on Combinatorial Auction Problems 97

3.12 Experimental Results on the Reason for the Speedup of BnB-ADOPT over ADOPT 98

4.1 Trace of the Updates of the Lower Bounds of BnB-ADOPT$_{AEM}$ with Absolute Error Bound $b = 24$ 105

4.2 Trace of the Updates of the Lower Bounds of BnB-ADOPT$_{REM}$ with Relative Error Bound $p = 3$ 106

4.3 Trace of the Updates of the Lower Bounds of BnB-ADOPT$_{WHM}$ with Relative Error Bound $w = 3$ 108

4.4 Experimental Results Comparing ADOPT with the Approximation Mechanisms on Graph Coloring Problems 116

4.5 Experimental Results Comparing BnB-ADOPT with the Approximation Mechanisms on Graph Coloring Problems 117

4.6 Experimental Results Comparing ADOPT and BnB-ADOPT with the Approximation Mechanisms on Sensor Network, Meeting Scheduling and Combinatorial Auction Problems 118

4.7 Experimental Results Comparing ADOPT$_{WHM}$, BnB-ADOPT$_{WHM}$ and MGM-k 120

5.1 Trace of Simplified Best-First Search (Centralized ADOPT without Memory Limitations) 126

5.2 Correlation of $\hat{P}(I)$ and $P(I)$ 132

5.3 Correlation of $\hat{E}(I)$ and $E(I)$ 134

5.4 Experimental Results Comparing ADOPT and BnB-ADOPT with the Caching Schemes on Graph Coloring Problems 144

5.5 Experimental Results Comparing ADOPT and BnB-ADOPT with the Caching Schemes on Sensor Network, Meeting Scheduling and Combinatorial Auction Problems 145

5.6 Experimental Results of ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ with the Caching Schemes on Graph Coloring Problems 149

5.7 Experimental Results of ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ with the Caching Schemes on Sensor Network, Meeting Scheduling and Combinatorial Auction Problems 150

6.1 Delta and Gamma Costs of the Example DCOP Problem 159

6.2 Delta and Gamma Costs after the Removal of the Constraint between Agents a_1 and a_3 160

6.3 Pseudocode of ReuseBounds 161

6.4 DFS, Mobed and HARP Pseudo-trees after Removal of the Constraint between Agents a_4 and a_5 164

6.5 Pseudocode of HARP 167

6.6 HARP Pseudo-tree Reconstruction Steps 168

6.7 Experimental Results Comparing ADOPT and BnB-ADOPT with the ReuseBounds Procedure and the Pseudo-tree Reconstruction Algorithms on Graph Coloring Problems with Change 1 203

6.8 Experimental Results Comparing ADOPT and BnB-ADOPT with the ReuseBounds Procedure and the Pseudo-tree Reconstruction Algorithms on Sensor Network, Meeting Scheduling and Combinatorial Auction Problems with Change 1 (1) 204

6.9 Experimental Results Comparing ADOPT and BnB-ADOPT with the ReuseBounds Procedure and the Pseudo-tree Reconstruction Algorithms on Sensor Network, Meeting Scheduling and Combinatorial Auction Problems with Change 1 (2) 205

6.10 Experimental Results Comparing ADOPT and BnB-ADOPT with the ReuseBounds Procedure and the Pseudo-tree Reconstruction Algorithms on Graph Coloring Problems with Change 2 206

6.11 Experimental Results Comparing ADOPT and BnB-ADOPT with the ReuseBounds Procedure and the Pseudo-tree Reconstruction Algorithms on Graph Coloring Problems with Change 3 206

Abstract

Distributed constraint optimization (DCOP) is a model where several agents coordinate with each other to take on values so as to minimize the sum of the resulting constraint costs, which are dependent on the values of the agents. This model is becoming popular for formulating and solving agent-coordination problems. As a result, researchers have developed a class of DCOP algorithms that use search techniques. For example, Asynchronous Distributed Constraint Optimization (ADOPT) is one of the pioneering DCOP search algorithms that has been widely extended. Since solving DCOP problems optimally is NP-hard, solving large problems efficiently becomes an issue.

DCOP search algorithms can be viewed as distributed versions of centralized search algorithms. Therefore, I hypothesize that one can speed up DCOP search algorithms by applying insights gained from centralized search algorithms, specifically (1) by using an appropriate search strategy, (2) by sacrificing solution optimality, (3) by using more memory, and (4) by reusing information gained from solving similar DCOP problems. However, DCOP search algorithms are sufficiently different from centralized search algorithms that these insights cannot be trivially applied.

To validate my hypotheses: (1) I introduce Branch-and-Bound ADOPT (BnB-ADOPT), an extension of ADOPT that changes the search strategy of ADOPT from

memory-bounded best-first search to depth-first branch-and-bound search, resulting in one order of magnitude speedup. These results validate my hypothesis that DCOP search algorithms that employ depth-first branch-and-bound search can be faster than DCOP search algorithms that employ memory-bounded best-first search. (2) I introduce an approximation mechanism that uses weighted heuristic values to trade off solution costs for smaller runtimes. This approximation mechanism allows ADOPT and BnB-ADOPT to terminate faster with larger weights, validating my hypothesis that DCOP search algorithms that use weighted heuristic values can have runtimes that decrease as larger weights are used. Additionally, the new approximation mechanism provides relative error bounds and thus complements existing approximation mechanisms that only provide absolute error bounds. (3) I introduce the MaxPriority, MaxEffort and MaxUtility DCOP-specific caching schemes, which allow ADOPT and BnB-ADOPT to cache DCOP-specific information when they have more memory available and terminate faster with larger amounts of memory. Experimental results show that the MaxEffort and MaxUtility schemes speed up ADOPT more than the currently used generic caching schemes, and the MaxPriority scheme speeds up BnB-ADOPT at least as much as the currently used generic caching schemes. Therefore, these results validate my hypothesis that DCOP-specific caching schemes can reduce the runtime of DCOP search algorithms at least as much as the currently used generic caching schemes. (4) I introduce an incremental procedure and an incremental pseudo-tree reconstruction algorithm that allow ADOPT and BnB-ADOPT to reuse information gained from solving similar DCOP problems to solve the current problem faster, resulting in runtimes that decrease with larger amounts of information reuse. These results validate my hypothesis that DCOP search algorithms that reuse

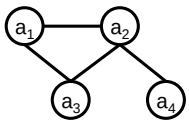
information from searches of similar DCOP problems to guide their search can have runtimes that decrease as they reuse more information.

Chapter 1

Introduction

Distributed constraint optimization (DCOP) (Modi, 2003; Mailler, 2004; Petcu, 2007; Pearce, 2007; Burke, 2008) is a model where several agents coordinate with each other to take on values so as to minimize the sum of the resulting constraint costs, which are dependent on the values of the agents. This model is becoming popular for formulating and solving agent-coordination problems (Lesser, Ortiz, & Tambe, 2003; Maheswaran, Pearce, & Tambe, 2004; Schurr, Okamoto, Maheswaran, Scerri, & Tambe, 2005; Junges & Bazzan, 2008; Ottens & Faltings, 2008; Kumar, Faltings, & Petcu, 2009). As a result, researchers have developed several DCOP algorithms that use search techniques. For example, Asynchronous Distributed Constraint Optimization (ADOPT) (Modi, Shen, Tambe, & Yokoo, 2005) is one of the pioneering DCOP search algorithms that has been widely extended. Since solving DCOP problems optimally is NP-hard (Modi et al., 2005), solving large problems efficiently becomes an issue.

DCOP search algorithms can be viewed as distributed versions of centralized search algorithms. Therefore, I hypothesize that one can speed up DCOP search algorithms by applying insights gained from centralized search algorithms, specifically (1) by using



(a)

a_1	a_2	Cost
0	0	5
0	1	8
1	0	20
1	1	3

a_1	a_3	Cost
0	0	5
0	1	10
1	0	20
1	1	3

a_2	a_3	Cost
0	0	5
0	1	4
1	0	3
1	1	3

a_2	a_4	Cost
0	0	3
0	1	8
1	0	10
1	1	3

(b)

Figure 1.1: Example DCOP Problem

an appropriate search strategy, (2) by sacrificing solution optimality, (3) by using more memory, and (4) by reusing information from searches of similar DCOP problems.

To validate my hypotheses: (1) I introduce Branch-and-Bound ADOPT (BnB-ADOPT), an extension of ADOPT that changes the search strategy of ADOPT from memory-bounded best-first search to depth-first branch-and-bound search, resulting in one order of magnitude speedup when solving sufficiently large DCOP problems; (2) I introduce an approximation mechanism that allows ADOPT and BnB-ADOPT to use weighted heuristic values to trade off solution costs for smaller runtimes; (3) I introduce DCOP-specific caching schemes that allow ADOPT and BnB-ADOPT to store more information when they have more memory available, which can be at least as fast as the currently used generic caching schemes; and (4) I introduce an incremental procedure and an incremental pseudo-tree reconstruction algorithm that allow ADOPT and BnB-ADOPT to reuse information from searches of similar DCOP problems. ADOPT and BnB-ADOPT terminate faster when they reuse more information.

1.1 DCOP Problems

A DCOP problem consists of a set of agents, each responsible for taking on (= assigning itself) a value from its finite domain. The agents coordinate their value assignments,

which are subjected to a set of constraints. Two agents are said to be constrained if they share a constraint. Each constraint has an associated cost, which depends on the values taken on by the constrained agents. An agent only knows the costs of constraints that it is involved in. A complete solution is an assignment of values to all agents, and a partial solution is an assignment of values to a subset of agents. The cost of a solution is the sum of the constraint costs of all constraints resulting from the given value assignments. Solving a DCOP problem optimally means to find a complete solution such that the sum of all constraint costs is minimized. Finding such a cost-minimal solution is NP-hard (Modi et al., 2005). It is common to visualize a DCOP problem as a constraint graph where the vertices are the agents and the edges are the constraints. Figure 1.1(a) shows the constraint graph of an example DCOP problem with four agents that can each take on the value zero or one, and Figure 1.1(b) shows the constraint costs. The cost-minimal solution of our example DCOP problem is where all agents take on the value one, incurring a total cost of 12 (3 from each constraint).

The DCOP model is becoming popular for formulating and solving agent-coordination problems such as the distributed scheduling of meetings (Maheswaran et al., 2004; Petcu & Faltings, 2005b; Greenstadt, Grosz, & Smith, 2007; Zivan, 2008; Yeoh, Felner, & Koenig, 2009, 2010), the distributed coordination of unmanned aerial vehicles (Schurr et al., 2005), the distributed coordination of sensors in a network (Lesser et al., 2003; Zhang, Xing, Wang, & Wittenburg, 2003; Yeoh, Sun, & Koenig, 2009a; Yeoh, Varakantham, & Koenig, 2009b; Zivan, Glinton, & Sycara, 2009; Lisỳ, Zivan, Sycara, & Péchoucek, 2010), the distributed allocation of resources in disaster evacuation scenarios (Carpenter, Dugan, Kopena, Lass, Naik, Nguyen, Sultanik, Modi, & Regli, 2007; Lass,

Kopena, Sultanik, Nguyen, Dugan, Modi, & Regli, 2008), the distributed synchronization of traffic lights (Junges & Bazzan, 2008), the distributed planning of truck routes (Ottens & Faltings, 2008), the distributed management of power distribution networks (Kumar et al., 2009) and the distributed generation of coalition structures (Ueda, Iwasaki, & Yokoo, 2010). As a result, researchers have developed several DCOP algorithms that use search techniques (= DCOP search algorithms). For example, ADOPT (Modi et al., 2005) is one of the pioneering DCOP search algorithms that has been widely extended (Modi & Ali, 2004; Ali, Koenig, & Tambe, 2005; Bowring, Tambe, & Yokoo, 2006; Davin & Modi, 2006; Pecora, Modi, & Scerri, 2006; Choxi & Modi, 2007; Matsui, Silaghi, Hirayama, Yokoo, & Matsuo, 2008; Silaghi & Yokoo, 2009; Matsui, Silaghi, Hirayama, Yokoo, & Matsuo, 2009; Gutierrez & Meseguer, 2010). ADOPT is a distributed best-first search algorithm that is complete and memory-bounded.

1.2 Hypotheses

DCOP search algorithms can be viewed as distributed versions of centralized search algorithms. Therefore, my hypothesis is as follows:

> *One can speed up DCOP search algorithms by applying insights gained from centralized search algorithms to DCOP search algorithms.*

Specifically, there are four common approaches that are used to speed up centralized search algorithms in the literature that can be applied to DCOP search algorithms:

1. One can speed up DCOP search algorithms by using an appropriate search strategy for the given problem type. A common approach is to use depth-first branch-and-bound search instead of memory-bounded best-first search for problems whose search trees are bounded. Researchers have shown that depth-first branch-and-bound search is faster than memory-bounded best-first search for these problems (Zhang & Korf, 1995). Therefore, I hypothesize that DCOP search algorithms that employ depth-first branch-and-bound search can be faster than DCOP search algorithms that employ memory-bounded best-first search since the search trees of DCOP problems are bounded.

2. One can speed up DCOP search algorithms by sacrificing solution optimality. A common approach is to use weighted heuristic values to focus the search. Algorithms that use this approach include Weighted A* (Pohl, 1970) and Weighted A* with dynamic weights (Pohl, 1973). These algorithms guarantee that the costs of the solutions found are at most a constant factor larger than the minimal costs, where the constant is the largest weight used. Typically, the runtime of these algorithms decreases as larger weights are used. Therefore, I hypothesize that DCOP search algorithms that use weighted heuristic values can have runtimes that decrease as larger weights are used.

3. One can speed up DCOP search algorithms by using more memory. A common approach is to cache information as long as memory is available, such that the cached information can be used when needed. Algorithms that use this approach

include MA* (Chakrabarti, Ghosh, Acharya, & DeSarkar, 1989) and SMA* (Russell, 1992). Typically, the runtime of these algorithms decreases as more memory is available. Motivated by these results, researchers have developed any-space versions of DCOP search algorithms, such as any-space ADOPT (Matsui, Matsuo, & Iwata, 2005) and any-space NCBB (Chechetka & Sycara, 2006a), and showed that the runtime of these algorithms indeed decreases as more memory is available. However, they use generic caching schemes, such as FIFO and LRU, that are similar to popular page replacement schemes used in operating systems. These generic schemes do not exploit the cached information in a DCOP-specific way. Therefore, I hypothesize that DCOP-specific caching schemes can reduce the runtime of DCOP search algorithms at least as much as the currently used generic caching schemes.

4. One can speed up DCOP search algorithms by reusing information from searches of similar DCOP problems. A common approach is to reuse information from searches of similar problems to guide the search. Algorithms that use this approach include incremental search algorithms (Koenig, Likhachev, Liu, & Furcy, 2004b) such as D* (Stentz, 1995), Adaptive A* (Koenig & Likhachev, 2005) and FRA* (Sun, Yeoh, & Koenig, 2009b). Typically, the runtime of these algorithms decreases as they reuse more information. Therefore, I hypothesize that DCOP search algorithms that reuse information from searches of similar DCOP problems to guide their search can have runtimes that decrease as they reuse more information.

1.3 Contributions

Although DCOP search algorithms can be viewed as distributed versions of centralized search algorithms, they are often independently developed. For example, ADOPT was developed independent of RBFS (Korf, 1993), but both algorithms share many properties. For example, both algorithms are memory-bounded, use the same search strategy and use the same principle to restore information already purged from memory. Therefore, one should be able to utilize the insights gained from centralized search algorithms to speed up DCOP search algorithms. However, DCOP search algorithms are sufficiently different from centralized search algorithms that these insights cannot be trivially applied. For example, unlike centralized search algorithms, DCOP search algorithms operate in a distributed fashion, have memory that is distributed among the agents and have agents that can only perform local searches, yet must follow a global search strategy.

This dissertation uses the four approaches described in Section 1.2 to speed up DCOP search algorithms. I make a design choice to use the framework of ADOPT, which is one of the pioneering DCOP search algorithms, as the starting platform for the work in this dissertation. The motivation for this decision is that ADOPT has been widely extended (Modi & Ali, 2004; Ali et al., 2005; Bowring et al., 2006; Davin & Modi, 2006; Pecora et al., 2006; Choxi & Modi, 2007; Matsui et al., 2008; Silaghi & Yokoo, 2009; Matsui et al., 2009; Gutierrez & Meseguer, 2010) in addition to it having good properties. For example, agents in ADOPT operate concurrently and asynchronously to solve subproblems in parallel, which results in smaller runtimes than if they are to operate sequentially and synchronously. This dissertation makes the following four contributions:

1. To assess the hypothesis that DCOP search algorithms that employ depth-first branch-and-bound search can be faster than DCOP search algorithms that employ memory-bounded best-first search, we introduce BnB-ADOPT. BnB-ADOPT is a DCOP search algorithm that uses the framework of ADOPT but changes the search strategy of ADOPT from memory-bounded best-first search to depth-first branch-and-bound search. Although there exist other DCOP search algorithms, such as SBB (Hirayama & Yokoo, 1997), NCBB (Chechetka & Sycara, 2006b) and AFB (Gershman, Meisels, & Zivan, 2009), that employ depth-first branch-and-bound search, it is difficult to determine if depth-first branch-and-bound search is faster than memory-bounded best-first search since these algorithms differ by more than their search strategies when compared to ADOPT. In fact, SBB has been shown to be slower than ADOPT (Modi et al., 2005) while NCBB and AFB have been shown to be faster than ADOPT (Chechetka & Sycara, 2006b; Gershman et al., 2009). Hence, we introduce BnB-ADOPT since these results make clear the need for two DCOP search algorithms that differ *only* in their search strategies. Experimental results show that BnB-ADOPT is up to one order of magnitude faster than ADOPT when solving sufficiently large DCOP problems. Therefore, these results validate the hypothesis that DCOP search algorithms that employ depth-first branch-and-bound search can be faster than DCOP search algorithms that employ memory-bounded best-first search.

   This work is non-trivial since ADOPT is a rather complicated distributed algorithm whose agents operate concurrently and asynchronously at all times. Agents can only perform local searches, yet must follow a global search strategy.

2. To assess the hypothesis that DCOP search algorithms that use weighted heuristic values can have runtimes that decrease as larger weights are used, we introduce the Weighted Heuristics mechanism. This approximation mechanism allows ADOPT and BnB-ADOPT to use weighted heuristic values to trade off solution costs for smaller runtimes. Additionally, the Weighted Heuristics mechanism also guarantees that the costs of the solutions found are at most a constant factor larger than the minimal costs, where the constant is the largest weight used. Experimental results show that ADOPT and BnB-ADOPT terminate faster with larger weights, validating the hypothesis that DCOP search algorithms that use weighted heuristic values can have runtimes that decrease as larger weights are used. Additionally, the Weighted Heuristics mechanism provides relative error bounds and thus complements the existing Absolute Error mechanism (Modi et al., 2005) that only provides absolute error bounds.

   This work is non-trivial since ADOPT and BnB-ADOPT use heuristic values differently than centralized search algorithms, such as A* (Hart, Nilsson, & Raphael, 1968). Furthermore, the proof of the quality guarantee for DCOP search algorithms cannot be trivially obtained from the proof for centralized search algorithms since DCOP search algorithms are distributed and agents only have local views of the problem or, in other words, they know only the agents that they share constraints with and the costs of those constraints.

3. To assess the hypothesis that DCOP search algorithms that use DCOP-specific caching schemes can have runtimes that are at least as small as using generic caching

schemes, we introduce the MaxPriority, MaxEffort and MaxUtility DCOP-specific caching schemes. They allow ADOPT and BnB-ADOPT to determine which information to purge from memory when their memory is full and new information needs to be stored in memory. Experimental results show that the MaxEffort and MaxUtility schemes speed up ADOPT more than the currently used generic caching schemes, and the MaxPriority scheme speeds up BnB-ADOPT as much as the currently used generic caching schemes. Therefore, these results validate the hypothesis that DCOP-specific caching schemes can reduce the runtime of DCOP search algorithms at least as much as the currently used generic caching schemes.

This work is non-trivial since agents in ADOPT and BnB-ADOPT need to predict future information access while having only local views of the problem or, in other words, knowing only the agents that they share constraints with and the costs of those constraints.

4. To assess the hypothesis that DCOP search algorithms that reuse information from searches of similar DCOP problems can have runtimes that decrease as they reuse more information, we model dynamic DCOP problems as sequences of static DCOP problems and introduce the ReuseBounds procedure and the Hybrid Algorithm for Reconstructing Pseudo-trees (HARP). The ReuseBounds procedure and HARP algorithm allow ADOPT and BnB-ADOPT to reuse information from searches of similar static DCOP problems to guide their search to potentially solve the current static DCOP problem faster. Experimental results show that ADOPT and

BnB-ADOPT with the ReuseBounds procedure and the HARP algorithm terminate faster when they reuse more information, validating the hypothesis that DCOP search algorithms that reuse information from searches of similar DCOP problems to guide their search can have runtimes that decrease as they reuse more information.

This work is non-trivial since agents in ADOPT and BnB-ADOPT need to identify information that they can reuse or, in other words, know whether changes to the dynamic DCOP problem can affect its local information while having only local views of the problem or, in other words, knowing only the agents that they share constraints with and the costs of those constraints.

1.4 Dissertation Structure

This dissertation is structured as follows: In the next chapter, we give an overview of centralized search algorithms, DCOP problems and the ADOPT algorithm. We then introduce the BnB-ADOPT algorithm in Chapter 3 and the approximation mechanisms that trade off solution costs for smaller runtimes in Chapter 4. Next, we describe the caching schemes in Chapter 5 and the incremental procedure and incremental pseudo-tree reconstruction algorithm in Chapter 6 before concluding in Chapter 7.

Chapter 2

Background

This chapter begins by providing an overview of DCOP problems in Section 2.1. Since ADOPT is the starting platform for the work in this dissertation, we provide an overview of ADOPT in Section 2.2. Finally, we give an overview of common approaches used to speed up centralized search algorithms in Section 2.3.

2.1 Overview of DCOP Problems

In this section, we describe how DCOP problems are related to constraint satisfaction problems before providing an overview of DCOP problems, algorithms, applications and common problem types used in experiments.

2.1.1 Constraint Satisfaction Problems

Centralized constraint satisfaction problems (CSPs) have been well studied in artificial intelligence (Dechter, 2003). A CSP is defined as a finite set of variables, a finite set of values for each variable and a finite set of constraints. A constraint is defined to be between multiple variables. Each constraint has an associated cost, which depends on

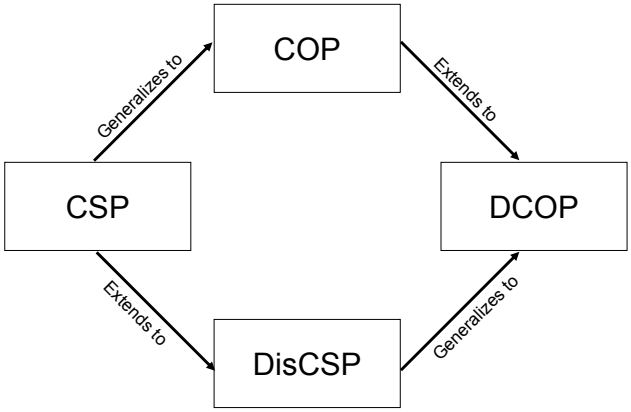


Figure 2.1: DCOP Problems as a Generalization and Extension of Constraint Satisfaction Problems

the values of the variables that it constrains. The constraint costs are Boolean (satisfied/unsatisfied). Solving a CSP means finding a solution such that all constraints are satisfied. An example application is the scheduling of jobs in a job-shop, where constraints express that some jobs can only be performed by certain machines and some jobs can only be performed after some other jobs. There could potentially be multiple solutions that satisfy all constraints. However, some solutions might be more desirable than others. For example, one might prefer the solution with the shortest completion time. Unfortunately, CSPs cannot capture these preferences. However, constraint optimization problems (COPs) are able to do so by using numeric constraint costs to represent the preferences. Therefore, one can view COPs as a generalization of CSPs.

Researchers have also extended CSPs to distributed constraint satisfaction problems (DisCSPs) (Yokoo, Durfee, Ishida, & Kuwabara, 1992, 1998). In a DisCSP, agents own

variables and are responsible for assigning values to the variables they own. Agents communicate with other agents to assign values to variables so as to satisfy all constraints. Similar to the generalization of CSPs to COPs, the DCOP model emerged as a generalization of the DisCSP model, where constraints no longer have Boolean costs but instead have numeric costs. As a result, several DCOP algorithms employ principles of DisCSP algorithms. For example, the NOGOOD messages used in ABT (Yokoo et al., 1998) (a DisCSP algorithm) are also used in ADOPT-ng (Silaghi & Yokoo, 2009) (a DCOP algorithm).

2.1.2 DCOP Problems

In this subsection, we formally define DCOP problems, describe the representation of solutions of DCOP problems and describe how heuristic values are used in DCOP problems.

2.1.2.1 Definition of DCOP Problems

Most DCOP algorithms operate on a pseudo-tree, which is a spanning tree of the (completely connected) constraint graph, previously defined in Section 1.1, with the property that edges in the constraint graph connect a vertex with one of its ancestor or descendant vertices in the pseudo-tree (Freuder & Quinn, 1985; Bayardo & Miranker, 1995). An edge of the constraint graph that is not part of the pseudo-tree is called a backedge. An agent c is called a pseudo-child agent of agent p if agent c is a descendant agent of agent p and they are both constrained via a backedge in the pseudo-tree. Similarly, agent p is called the pseudo-parent agent of agent c. Sibling subtrees in the pseudo-tree represent

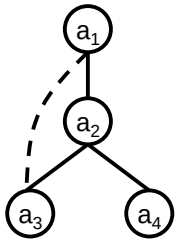


Figure 2.2: Example Pseudo-tree

independent DCOP subproblems (since no two agents in different sibling subtrees share a constraint). Figure 2.2 shows one possible pseudo-tree for the example DCOP problem in Figure 1.1, where the assignments of values to agents a_3 and a_4 are independent subproblems (the dotted line is a backedge).

A DCOP problem is defined by the following:

- a finite set of agents $A = \{a_1, a_2, ..., a_n\}$;
- a set of finite domains $D = \{Dom(a_1), Dom(a_2), ..., Dom(a_n)\}$, where $Dom(a_i)$ is the domain of possible floating point values of agent $a_i \in A$; and
- a set of binary constraints $F = \{f_1, f_2, ..., f_m\}$, where each constraint $f_i : Dom(a_{i,1}) \times Dom(a_{i,2}) \rightarrow \mathbb{R}^+ \cup \infty$, specifies its non-negative constraint cost as a function of the values of the distinct agents $a_{i,1}$ and $a_{i,2}$ that share the constraint.

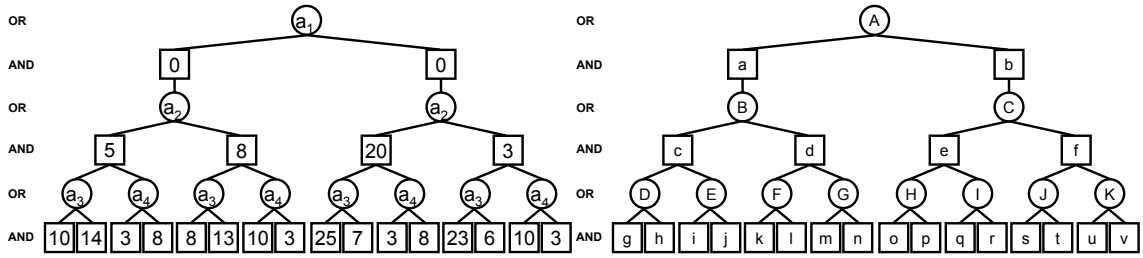
The above definition assumes that each agent takes on one value rather than multiple values, for example, a different value for each constraint that it is involved in. These DCOP problems are more commonly formulated as each agent being responsible for the assignments to multiple variables. However, there exist techniques that reduce such DCOP problems to our DCOP problems (Yokoo, 2001; Burke & Brown, 2006). Thus, we

use the terms agent and variable interchangeably. The above definition also assumes that constraints are binary (= between two agents) rather than n-ary (= between n agents). However, this restriction is justified since it has been shown that any problem with n-ary constraints can be represented as a problem with binary constraints (Bacchus, Chen, van Beek, & Walsh, 2002). Additionally, we assume that each DCOP problem is not made up of disjoint DCOP subproblems since each disjoint DCOP subproblem can be solved as an independent DCOP problem. We also assume that there are at least two agents in each DCOP problem and messages sent between agents can be delayed by a finite amount of time but are never lost and are received in the order they were sent.

2.1.2.2 Search Trees

The solution space of DCOP problems can be visualized with search trees. Traditional search trees or, synonymously, OR search trees (Marinescu & Dechter, 2009) assign values to agents sequentially. They do not utilize the fact that the values of agents that belong to independent DCOP subproblems do not have to be assigned sequentially. AND/OR search trees are based on pseudo-trees and remedy this issue (Marinescu & Dechter, 2009). Thus, we use AND/OR search trees and refer to them as search trees in this dissertation. Their depth is bounded by twice the number of agents.

Figure 2.3(a) shows the search tree that is based on the pseudo-tree in Figure 2.2. Figure 2.3(b) labels each node of the search tree with an identifier to allow us to refer to the nodes easily. Circular nodes are OR nodes (labeled with upper-case letters) and correspond to agents. For example, the agent of node C is agent a_2. Left branches of OR nodes correspond to the agents taking on value 0, and right branches correspond to the



(a) (b)

Figure 2.3: AND/OR Search Tree

agents taking on value 1. Square nodes are AND nodes (labeled with lower-case letters) and correspond to the partial solutions from the root node to those nodes. For example, the partial solution of node f is $\{(a_1, 1), (a_2, 1)\}$. The subtree rooted at an AND node represents the DCOP subproblem under the assumption that the (partial) solution to the complement DCOP subproblem is the partial solution of the AND node. For example, the subtree rooted at node f represents the DCOP subproblem of assigning values to agents a_3 and a_4 given the partial solution $\{(a_1, 1), (a_2, 1)\}$. The number of independent DCOP subproblems within this DCOP subproblem is indicated by the number of branches exiting the AND node. For example, there are two branches exiting node f, indicating that there are two independent DCOP subproblems, namely of assigning values to agents a_3 and a_4.

The numbers in the AND nodes in Figure 2.3(a) are the delta costs of the nodes. The delta cost of an AND node is defined to be the sum of the constraint costs of all constraints in its partial solution that involve the agent of its parent OR node. For example, the partial solution of node v is $\{(a_1, 1), (a_2, 1), (a_4, 1)\}$. There are two constraints in this partial solution, namely the constraint between agents a_1 and a_2, which has constraint cost 3, and the constraint between agents a_2 and a_4, which also has constraint cost 3.

Since the parent node of node v is node K with agent a_4, the delta cost of node v is 3, namely the constraint cost of the latter constraint. The former constraint is not included since it does not involve agent a_4. The solution cost of a partial solution of an AND node is the sum of the delta costs of all AND nodes along the branch from the root node to that node. For example, the solution cost of the partial solution of node v (= 6) is the sum of the delta costs of nodes b, f and v. In our example DCOP problem, a cost-minimal solution is the union of the partial solutions of nodes t and v (all agents take on value 1). Thus, the minimal solution cost (= 12) is the sum of the delta costs of nodes b, f, t and v.

2.1.2.3 Heuristic Values

Each AND node can have a heuristic value that estimates the smallest cost of assigning values to agents of all the OR nodes in the subtree rooted at that AND node, given the partial solution of that AND node. For example, the agents of the OR nodes in the subtree rooted at node f are agents a_3 and a_4, and the partial solution of node f is $\{(a_1, 1), (a_2, 1)\}$. Given this partial solution, the smallest cost of assigning values to agents a_3 and a_4 is the sum of the delta costs of nodes t and v (= 9). Heuristic values are typically used by search algorithms to speed up the search by making them more informed. The heuristic value of an AND node is admissible iff it does not overestimate the smallest cost of assigning values to the agents of all OR nodes in the subtree rooted at that AND node, given the partial solution of that AND node (Pearl, 1985). For example, if the heuristic value of node f in Figure 2.3 is no larger than 9, then it is an admissible heuristic value.

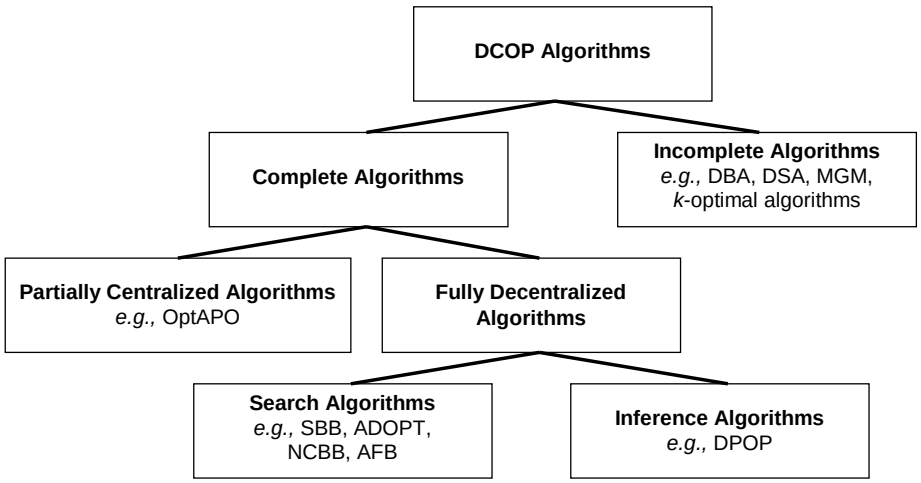


Figure 2.4: Taxonomy of DCOP Algorithms

2.1.3 DCOP Algorithms

We now illustrate the relationship of DCOP search algorithms to other DCOP algorithms by providing a taxonomy of DCOP algorithms. Figure 2.4 shows the taxonomy. DCOP algorithms are divided into two groups, namely complete and incomplete DCOP algorithms. Complete DCOP algorithms find cost-minimal solutions while incomplete DCOP algorithms are often faster but typically find suboptimal solutions.

2.1.3.1 Incomplete DCOP Algorithms

Incomplete DCOP algorithms typically use local search to find locally optimal solutions and can thus potentially get trapped in local minima. Nevertheless, since solving DCOP problems optimally is NP-hard, such DCOP algorithms are desirable for large DCOP problems where finding cost-minimal solutions might be slow. DBA (Hirayama & Yokoo, 2005), DSA (Fitzpatrick & Meertens, 2003), MGM (Maheswaran et al., 2004),

decentralized Max-Sum (Farinelli, Rogers, Petcu, & Jennings, 2008; Stranders, Farinelli, Rogers, & Jennings, 2009a, 2009b; Stranders, Delle Fave, Rogers, & Jennings, 2010), k-optimal DCOP algorithms (Pearce & Tambe, 2007; Bowring, Pearce, Portway, Jain, & Tambe, 2008; Bowring, Yin, Zinkov, & Tambe, 2009; Greenstadt, 2009), t-optimal DCOP algorithms (Kiekintveld, Yin, Kumar, & Tambe, 2010) and DCOP algorithms using the Anytime Local Search (Zivan, 2008) and Divide-and-Coordinate (Vinyals, Pujol, Rodriguez-Aguilarhas, & Cerquides, 2010) frameworks are examples of incomplete DCOP algorithms.

2.1.3.2 Complete DCOP Algorithms

Complete DCOP algorithms are generally divided into two groups, namely partially centralized and fully decentralized DCOP algorithms.

- **Partially centralized DCOP algorithms** allow some agents to transfer their constraint information (= information regarding the constraints that they are involved in) to a central agent for processing. OptAPO (Mailler & Lesser, 2004; Grinshpoun & Meisels, 2008) is an example of a partially centralized DCOP algorithm that uses cooperative mediation, where certain agents act as mediators to solve overlapping DCOP subproblems centrally.

- **Fully decentralized DCOP algorithms** do not have central agents that collect constraint information of other agents that are not constrained with them. Rather, every agent has access to only its own constraint information. Fully decentralized

DCOP algorithms are generally divided into two groups, namely DCOP inference and search algorithms.

- **DCOP inference algorithms** typically use dynamic programming to propagate aggregated constraint costs from one agent to another agent and thus reduce the DCOP problem size by one agent at each step. They repeat this procedure until the DCOP problem size is reduced to only one agent and the solution space thus cannot be reduced anymore. The sole remaining agent then has sufficient knowledge to find a cost-minimal solution. Action-GDL (Vinyals, Rodríguez-Aguilar, & Cerquides, 2009) and DPOP (Petcu & Faltings, 2005a, 2005b, 2005c, 2007; Petcu, Faltings, & Mailler, 2007; Atlas & Decker, 2007; Brito & Meseguer, 2010) are example DCOP inference algorithms. The number of messages sent between agents in Action-GDL and DPOP is only linear in the number of agents. However, their memory requirements are exponential in the induced width of the DCOP problem. The induced width depends on the number of backedges in the pseudo-tree. It can be as large as the number of agents minus one if the constraint graph is fully connected and every agent is thus constrained with every other agent.
- **DCOP search algorithms** use search strategies to search through the solution space to find a cost-minimal solution. ADOPT uses best-first search, and SBB (Hirayama & Yokoo, 1997), NCBB (Chechetka & Sycara, 2006b),

AFB (Gershman et al., 2009) and our new DCOP search algorithm, BnB-ADOPT, use depth-first branch-and-bound search. Their memory requirements are only polynomial in the number of agents. However, the number of messages sent between agents can be exponential in the number of agents.

2.1.4 DCOP Applications

The DCOP model is a popular way of formulating and solving agent-coordination problems. This model is well-suited for formulating multi-agent problems since they are distributed by nature. Moreover, distributed algorithms are able to work on subproblems in parallel and thus might be faster than centralized algorithms. As a result, the DCOP model has been used in formulating multi-agent problems such as the distributed scheduling of meetings (Maheswaran et al., 2004; Petcu & Faltings, 2005b; Greenstadt et al., 2007; Zivan, 2008; Yeoh et al., 2009, 2010), the distributed coordination of unmanned aerial vehicles (Schurr et al., 2005), the distributed coordination of sensors in a network (Lesser et al., 2003; Zhang et al., 2003; Yeoh et al., 2009a, 2009b; Zivan et al., 2009; Lisỳ et al., 2010), the distributed allocation of resources in disaster evacuation scenarios (Carpenter et al., 2007; Lass et al., 2008), the distributed synchronization of traffic lights (Junges & Bazzan, 2008), the distributed planning of truck routes (Ottens & Faltings, 2008), the distributed management of power distribution networks (Kumar et al., 2009) and the distributed generation of coalition structures (Ueda et al., 2010).

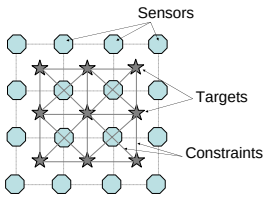


Figure 2.5: Example Sensor Network Problem

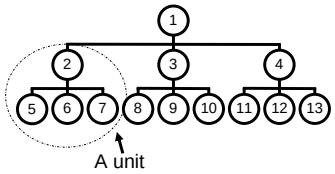


Figure 2.6: Example Meeting Scheduling Problem

2.1.5 DCOP Problem Types Used in the Experiments in this Dissertation

The three most popular DCOP problem types used to experimentally evaluate the performance of DCOP algorithms are graph coloring problems, sensor network problems and meeting scheduling problems (Junges & Bazzan, 2008). Therefore, we use these three problem types as well as an additional combinatorial auction problem type in the experiments of this dissertation.

- A **graph coloring problem** involves coloring the vertices of a graph. The agents are the vertices, their domains are the colors, and the constraints are between adjacent vertices.

- A **sensor network problem** involves assigning targets to sensors in a sensor network, taking restrictions in the availability of the sensors, restrictions in the number of sensors that need to track each target, and priorities of the targets into account. The agents are the targets, their domains are the time slots when they can be tracked, and the constraints are between adjacent targets (Maheswaran et al., 2004). Figure 2.5 shows a sensor network where the targets are located on a grid and each target is surrounded by 4 sensors, all of which are needed to track the target.

- A **meeting scheduling problem** involves scheduling meetings between the employees of a company, taking restrictions in their availability as well as their priorities into account. The agents are the meetings, their domains are the time slots when they can be held, and the constraints are between meetings that share participants (Maheswaran et al., 2004). Figure 2.6 shows a hierarchical organization with four units of a supervisor and their three subordinates, such as supervisor 2 with three subordinates 5, 6 and 7. In each unit, we assume five possible meetings: one of the entire unit (2, 5, 6, 7), two parent-child meetings (2, 5 and 2, 7) and two sibling-sibling meetings (5, 6 and 6, 7).

- A **combinatorial auction problem** involves determining the winner(s) in a combinatorial auction. The agents are the bidders with one bid each, their domains are the Boolean values indicating if they win or lose their bids, and the constraints are between bidders that bid on a common item (Petcu, Faltings, & Parkes, 2008; Kumar, Petcu, & Faltings, 2008).

2.2 Overview of ADOPT

Aside from being a very popular DCOP search algorithm, ADOPT also has many desirable properties. For example, it is memory-bounded, it restricts communication to only be between neighboring agents, its agents operate concurrently and asynchronously, and it orders the agents into a non-degenerate pseudo-tree. In this section, we describe its properties, the search strategy that it employs, and the pre-processing techniques used to speed it up.

DCOP Algorithm	Memory Requirement	Communication Restriction	Agent Operation	Pseudo-tree Structure
SBB	bounded	only with neighbors	sequential & synchronous	chain
ADOPT	bounded	only with neighbors	concurrent & asynchronous	tree
NCBB	bounded	only with neighbors	sequential & synchronous	tree
AFB	bounded	with all agents	concurrent & asynchronous	chain

Table 2.1: Properties of Complete DCOP Search Algorithms

2.2.1 Properties of ADOPT

Table 2.1 shows the properties of some complete DCOP search algorithms. We will now describe each property in more detail.

- **Memory Requirement:** The memory requirement is the amount of memory a complete DCOP search algorithm needs to have in order to guarantee that it finds a cost-minimal solution. In applications such as sensor networks, agents/sensors might have only a small amount of memory available. Thus, it is desirable for complete DCOP search algorithms to be developed with this limitation in mind. All four complete DCOP search algorithms, SBB, ADOPT, NCBB and AFB, are memory-bounded, in that each agent requires only a linear (in the number of agents) amount of memory. Thus, ADOPT is desirable because it is *memory-bounded*.

- **Communication Restriction:** Agents in complete DCOP search algorithms typically communicate with each other via messages. In applications such as sensor networks, agents/sensors can communicate only with their neighbors due to their limited communication radius. Since neighboring sensors need to coordinate with each other to sense the areas near them, neighboring sensors often share constraints. Thus, it is desirable for complete DCOP search algorithms to be developed with

this communication restriction in mind. SBB, ADOPT and NCBB restrict communication to be only between agents that share constraints, while AFB does not impose this restriction and allows its agents to broadcast messages to all other agents. Thus, ADOPT is desirable because agents in ADOPT *communicate only with neighboring agents.*

- **Agent Operation:** Agents in complete DCOP search algorithms can operate sequentially or concurrently. Agents that operate sequentially are often idle while waiting for the active agent to finish its computation. On the other hand, agents that operate concurrently are operating at all times. Concurrent operation is preferred since DCOP problems can potentially be solved more quickly if the agents can perform potentially useful computation instead of waiting for other agents. Agents in SBB and NCBB operate sequentially while agents in ADOPT and AFB operate concurrently. Thus, ADOPT is desirable because agents in ADOPT *operate concurrently.*

  Concurrent agents in complete DCOP search algorithms can operate synchronously or asynchronously. Agents that operate synchronously operate in time slices called cycles (Hirayama & Yokoo, 2000). At the start of each cycle, each agent collects all messages that were sent to it. It then performs its computation and sends out new messages at the end of the cycle. Therefore, every agent must wait until the last agent is done sending its messages, before starting its new cycle. On the other hand, agents that operate asynchronously can operate independently of each other. It is desirable to allow agents to operate asynchronously since asynchrony increases

the robustness of the algorithm (Silaghi, Landwehr, & Larrosa, 2004). For example, if a single communication link suffers from congestion, all synchronous agents are affected while only a small number of asynchronous agents are affected. Agents in SBB and NCBB operate synchronously while agents in ADOPT and AFB operate asynchronously. Thus, ADOPT is desirable because agents in ADOPT *operate asynchronously.*

- **Pseudo-tree Structure:** Complete DCOP search algorithms typically start with a pre-processing step of ordering the agents into a pseudo-tree. Algorithms can order the agents into a non-degenerate tree or degenerate tree, which is a chain. A tree ordering is able to capture independent subproblems (represented as sibling subtrees) while a chain ordering is not able to do so. Thus, a tree ordering is desirable since algorithms that operate on trees are able to operate on independent subproblems independently, while algorithms that operate on chains are not able to do so. SBB and AFB use a chain ordering while ADOPT and NCBB use a tree ordering. Thus, ADOPT is desirable because it *uses a tree ordering.*

2.2.2 Search Strategy of ADOPT

We now describe the search strategy of ADOPT from a centralized perspective. ADOPT traverses the search tree in a best-first search order. We now describe a simplified version of best-first search. The complete version can be found in (Marinescu & Dechter, 2007). Best-first search maintains a list that initially contains only the child AND nodes of the root node. It repeatedly performs the following operations: It expands the AND node

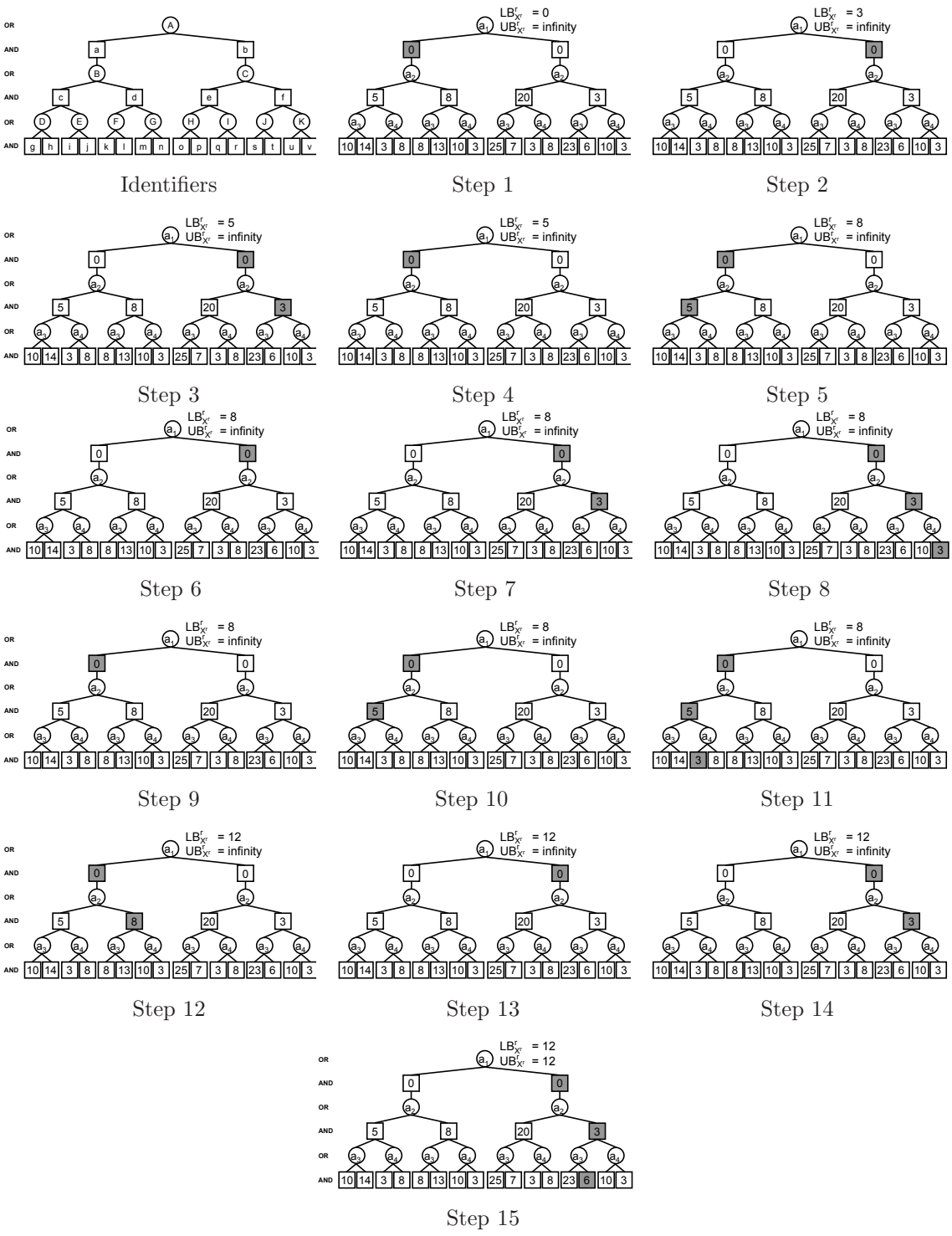


Figure 2.7: Trace of Simplified Memory-Bounded Best-First Search (Centralized ADOPT)

with the smallest estimate of the complete solution cost in the list by removing that node from the list and adding the grandchild AND nodes of that node into the list. The estimate of the complete solution cost of an AND node is the sum of the solution cost of the partial solution of that AND node and the heuristic value of that AND node. For our example DCOP problem, if the zero heuristic values are used, best-first search expands the AND nodes in the search tree in Figure 2.3 for the first time in the following order, where the numbers in parentheses indicate the estimates of complete solution costs of the expanded nodes: a (0), b (0), f (3), c (5), v (6), i (8), d (8) and t (9).

Figure 2.7 shows a simplified trace of ADOPT on our example DCOP problem. ADOPT terminates after fifteen steps with minimal complete solution cost 12. The numbers in the AND nodes are the delta costs of the nodes. The lower bound $LB^r_{X^r}$ is an optimistic estimate of the minimal solution cost. It is the smallest underestimated solution cost, over all solutions. The underestimated solution cost of a solution is the sum of the delta costs of all AND nodes of that solution whose parent OR node is the root node or whose grandparent AND node is expanded. For example, the underestimated solution cost of the solution $\{(a_1, 1), (a_2, 1), (a_3, 1), (a_4, 1)\}$ is 3 if node b is expanded and nodes f, t and v are not expanded. The upper bound $UB^r_{X^r}$ is a pessimistic estimate of the minimal solution cost. It is the smallest solution cost of any complete solution found so far. ADOPT terminates when the upper bound $UB^r_{X^r}$ is no larger than the lower bound $LB^r_{X^r}$. In order to be memory-bounded, ADOPT maintains only one branch of the search tree (shaded grey in the figure) from the root node to the currently expanded node and thus needs to repeatedly reconstruct nodes that it purged from memory. For example, in Step 3, ADOPT has the branch to node f in memory. The next node that

best-first search expands is node c, and ADOPT discards the branch to node f in Step 4. In Steps 6 and 7, it then needs to reconstruct the discarded branch to node f in order to expand node v in Step 8.

2.2.3 Pre-processing Techniques for ADOPT

There are two known pre-processing techniques that speed up ADOPT.

- The first technique is to use heuristic values. Originally, ADOPT did not use heuristic values, but researchers later extended it to use admissible heuristic values to speed it up. The admissible heuristic values are computed by solving a relaxed version (where backedges are ignored) of the DCOP problem using a dynamic programming-based approach (Ali et al., 2005). Experimental results show that the use of these heuristic values can speed up ADOPT by one order of magnitude when solving graph coloring problems at low pre-processing costs (Ali et al., 2005).

- The second technique is to transform the DCOP problem into an equivalent DCOP problem that is simpler to solve using directed arc consistency methods (Matsui et al., 2009). The transformed DCOP problem is simpler to solve because it has reduced constraint costs and often has a smaller number of constraints. The use of this approach can also speed up ADOPT by one order of magnitude when solving graph coloring problems (Matsui et al., 2009).

2.3 Overview of Approaches Used to Speed Up Centralized Search Algorithms

The hypothesis of this dissertation is that one can speed up DCOP search algorithms by using insights from centralized search algorithms. Thus, in this section, we describe four common approaches used to speed up centralized search algorithms that can be applied to DCOP search algorithms.

2.3.1 Search Strategies

Since different search strategies have different benefits and drawbacks, a search strategy that does well in a particular problem type might not do so well in another problem type. Therefore, it is important to use an appropriate search strategy for the given problem type, except that it is usually very difficult to make recommendations a priori. In this subsection, we first describe the depth-first branch-and-bound search strategy and then compare it to the memory-bounded best-first search strategy described in Section 2.2.2.

We now describe depth-first branch-and-bound search using the same definitions of $LB^r_{X^r}$ and $UB^r_{X^r}$ as described earlier in the context of Figure 2.7. After it has expanded an AND node, depth-first branch-and-bound search expands the grandchild AND node with the smallest estimate of the complete solution cost. Depth-first branch-and-bound search prunes those AND nodes whose estimate of the complete solution costs are no smaller than $UB^r_{X^r}$. It backtracks once all grandchild AND nodes have been expanded or pruned. The algorithm terminates when it is not able to backtrack to an AND node whose estimate of the complete solution cost is smaller than $UB^r_{X^r}$. It returns the complete

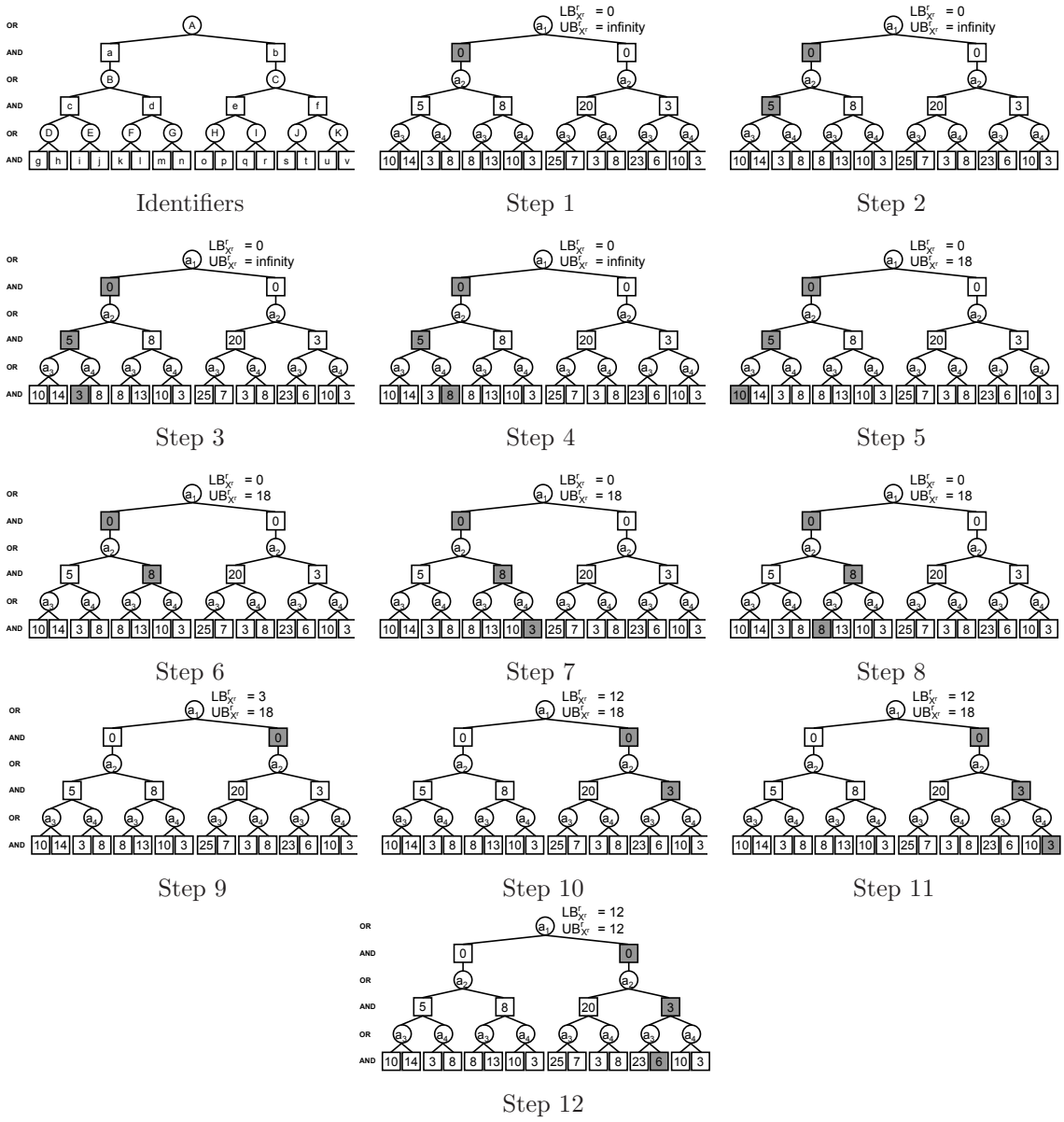


Figure 2.8: Trace of Depth-First Branch-and-Bound Search

solution with the smallest cost upon termination. For our example DCOP problem, if the zero heuristic values are used, depth-first branch-and-bound search expands the AND nodes in the search tree in the following order, where it prunes the nodes in brackets: a (0), c (5), i (8), j (13), g (15), [h (19)], d (8), n (11), k (16), [m (18)], [l (21)], b (0), f (3), v (6) and t (9).

The primary drawback of depth-first branch-and-bound search is that it can expand nodes along a very long and potentially infinite branch down the search tree when a solution might be near the root of the search tree. Another drawback is that it can expand nodes that memory-bounded best-first search does not expand, such as node j in Step 4. However, unlike memory-bounded best-first search, it expands each node at most once and thus do not re-expand nodes that it purged from memory.

Since both search strategies have benefits and drawbacks, researchers have compared, theoretically and experimentally, depth-first branch-and-bound search and RBFS (Korf, 1993), an algorithm that employs memory-bounded best-first search, on random OR search trees of varying branching factors, depths, and delta costs (Zhang & Korf, 1995). They concluded that depth-first branch-and-bound search often runs faster than RBFS on depth-bounded search trees, and vice versa on search trees with unbounded depth. This insight is promising since the depth of search trees of DCOP problems is bounded by twice the number of agents in the problem. Thus, although the conclusions were only proven and tested for centralized search algorithms, one can hypothesize that DCOP search algorithms that employ depth-first branch-and-bound search might be faster than DCOP search algorithms that employ memory-bounded best-first search.

The primary drawback of memory-bounded best-first search is that it needs to repeatedly re-expand nodes that it purged from memory due to its memory limitation. If there are no memory limitations, best-first search will no longer need to re-expand nodes. In fact, for problems where the entire search tree can be stored in memory, researchers have theoretically proved that A* (Hart et al., 1968), a centralized search algorithm that employs best-first search, requires the least number of node expansions compared to

other centralized search algorithms, assuming that the algorithms use identical consistent heuristic values (Dechter & Pearl, 1985). Therefore, one can expect that best-first search should be faster than depth-first branch-and-bound search since depth-first branch-and-bound search can expand nodes that best-first search does not. In our example DCOP problem, best-first search will expand nodes in exactly the same way as shown in Figure 2.7, except that it will skip Steps 4, 6, 7, 9, 10, 13 and 14. Thus, it will terminate after eight steps, which is four steps fewer than the twelve steps that depth-first branch-and-bound search needs. However, this insight does not directly apply to memory-bounded DCOP search algorithms unless their memory requirements are relaxed.

2.3.2 Approximation Algorithms

Another common approach to speed up centralized search algorithms is to sacrifice solution optimality by using approximation algorithms. Approximation algorithms typically find complete solutions faster than complete algorithms. However, the costs of the solutions found can be suboptimal, but they are within a given error bound of the minimal costs (Pohl, 1970, 1973; Pearl & Kim, 1982; Neller, 2002).

There are generally two approaches used by approximation algorithms for centralized search. The first approach uses an early termination detection scheme to terminate once a solution within a given error bound ϵ is found (Zhang, 2000; Neller, 2002). For example, imagine a version of depth-first branch-and-bound search whose termination condition is slightly different from the one described in Section 2.3.1. This version of depth-first branch-and-bound search terminates when it is not able to backtrack to a node whose estimate of the complete solution cost is smaller than $UB^r_{X^r} - \epsilon$. In our example DCOP

problem, this version of depth-first branch-and-bound search with $\epsilon = 20$ will terminate after five steps when the estimate of the complete solution cost of each node that it can backtrack to is larger than $UB^r_{X^r} - \epsilon$. Thus, it takes seven steps fewer than the optimal version of depth-first branch-and-bound search, but with a suboptimal solution of cost 18. The costs of the solutions found by two approximation algorithms that use this approach, Truncated DFBnB (Zhang, 2000) and ϵ-RBFS (Neller, 2002), have been shown to be at most ϵ larger than the minimal cost. Larger values of ϵ typically allow these approximation algorithms to terminate earlier at the cost of potentially larger errors. Thus, one can potentially use this approach to speed up DCOP search algorithms as well.

The second approach uses inadmissible heuristic values computed by multiplying admissible heuristic values with sufficiently large weights (Pohl, 1970, 1973). These weighted heuristic values prevent the approximation algorithms from expanding nodes that they would otherwise expand. For example, imagine that the heuristic values of nodes b and d in our example DCOP problem are 20 and the heuristic values of every other node are 0. Then, depth-first branch-and-bound search will terminate after five steps when the estimate of the complete solution cost of each node that it can backtrack to is larger than $UB^r_{X^r}$. Thus, it takes seven steps fewer that the optimal version of depth-first branch-and-bound search, but with a suboptimal solution of cost 18. The costs of the solutions found by two approximation algorithms that use this approach, Weighted A* (Pohl, 1970) and Weighted A* with dynamic weights (Pohl, 1973), have been shown to be at most a constant factor larger than the minimal cost, where the constant is the largest weight

used. Larger weights typically allow these approximation algorithms to terminate earlier at the cost of potentially larger errors. Since DCOP search algorithms use heuristic values, one can potentially speed them up by using weighted heuristic values.

2.3.3 Caching Algorithms

The primary drawback of memory-bounded best-first search is that it needs to repeatedly re-expand nodes that it purged from memory. Therefore, if search algorithms that employ memory-bounded best-first search have more memory available than the minimal amount required by the search, then they can cache more nodes in memory, resulting in fewer nodes being purged. As a result, they should re-expand fewer nodes and thus have smaller runtimes.

Researchers have used this insight to develop algorithms, such as MA* (Chakrabarti et al., 1989) and SMA* (Russell, 1992), that employ any-space best-first search, which is a version of memory-bounded best-first search that uses more memory than the minimal amount. For example, SMA* will not purge nodes from memory as long as its memory is not full. Although SMA* was designed to operate on OR search trees, it can be easily extended to operate on AND/OR search trees as well. We thus describe its operation on AND/OR search trees. When the memory of SMA* is full, among all AND nodes in memory with no grandchild AND nodes in memory, SMA* purges the AND node with the largest estimate of the complete solution cost. If there are multiple AND nodes with the same largest estimate, it purges the AND node with the smallest depth in the search tree. If there are multiple AND nodes with the same smallest depth, it breaks ties randomly. To investigate the relationship between runtime and memory availability,

experiments were performed on sliding tile puzzles using perturbed Manhattan distances as heuristic values. As expected, the runtime[1] of SMA* decreases as more memory is available (Russell, 1992).

This result is promising since ADOPT employs memory-bounded best-first search and thus its runtime might also decrease when it uses more memory that is available. However, unlike MA* and SMA*, the memory is distributed among the agents in DCOP search algorithms. Thus, the caching schemes of centralized search algorithms cannot be applied directly to DCOP search algorithms.

2.3.4 Incremental Search Algorithms

Researchers have modeled dynamic path-planning problems as sequences of static path-planning problems with changes between consecutive problems (Stentz, 1995; Koenig et al., 2004b). If the changes between two consecutive problems are small, one can potentially use information from solving the first problem to help solve the second problem faster. Centralized search algorithms that use this approach are called incremental search algorithms (Koenig et al., 2004b). Incremental search algorithms can potentially find solutions to a series of problems faster than is possible by solving each problem from scratch, while guaranteeing that the cost-minimal solution is found for each problem (Koenig et al., 2004b). Typically, the runtime of these algorithms decreases as they reuse more information.

There are generally three classes of incremental search algorithms. Algorithms of the first class reuse information from previous searches to update the heuristic values

[1]Measured in the number of nodes expanded.

of the current search to make them more informed (Koenig & Likhachev, 2005; Koenig, Likhachev, & Sun, 2007; Sun, Koenig, & Yeoh, 2008). Algorithms that belong to this class include Adaptive A* (Koenig & Likhachev, 2005), MT-Adaptive A* (Koenig et al., 2007) and GAA* (Sun et al., 2008). Experimental results show that GAA* can be up to twice faster than algorithms that solve each problem from scratch (Sun et al., 2008). Typically, the larger the number of heuristic values that are made more informed, the smaller the runtime of the current search. This approach is promising since DCOP search algorithms use heuristic values to guide their searches.

Incremental search algorithms of the second class transform the OR search tree of the previous search to the OR search tree of the current search (Stentz, 1995; Koenig & Likhachev, 2002; Koenig, Likhachev, & Furcy, 2004a; Sun, Yeoh, & Koenig, 2010b). Algorithms that belong to this class include D* (Stentz, 1995), D* Lite (Koenig & Likhachev, 2002), LPA* (Koenig et al., 2004a) and MT-D* Lite (Sun et al., 2010b). Experimental results show that D* Lite can be up two orders of magnitude faster than algorithms that solve each problem from scratch in four-neighbor random gridworlds when the number of changes is small (Sun, Yeoh, & Koenig., 2009a). Typically, the larger the similarity between the previous and current OR search trees, the smaller the runtime of the current search. This approach is promising since DCOP search algorithms use AND/OR search trees, which are similar to OR search trees.

Incremental search algorithms of the third class identify the portion of the OR search tree from the previous search that can be reused for the current search (Sun & Koenig, 2007; Sun et al., 2009b; Sun, Yeoh, & Koenig, 2010a). Algorithms that belong to this class include FSA* (Sun & Koenig, 2007), FRA* (Sun et al., 2009b) and G-FRA* (Sun et al.,

2010a). Experimental results show that FRA* can be up to one order of magnitude faster than algorithms that solve each problem from scratch in four-neighbor random gridworlds (Sun et al., 2009b). Typically, the larger the portion of the OR search tree from the previous search that is reused, the smaller the runtime of the current search. This approach is promising since DCOP search algorithms use AND/OR search trees, which are similar to OR search trees.

Chapter 3

Speeding Up via Appropriate Search Strategies

This chapter introduces Branch-and-Bound ADOPT (BnB-ADOPT), a DCOP search algorithm that uses the framework of ADOPT but changes the search strategy of ADOPT from memory-bounded best-first search to depth-first branch-and-bound search. Our experimental results show that BnB-ADOPT is up to one order of magnitude faster than ADOPT when solving sufficiently large DCOP problems. Therefore, these results validate the hypothesis that DCOP search algorithms that employ depth-first branch-and-bound search can be faster than DCOP search algorithms that employ memory-bounded best-first search.

This chapter is organized as follows: We first describe the motivation for our work in Section 3.1. In Section 3.2, we provide a detailed description of the BnB-ADOPT algorithm. We then prove its correctness and completeness and describe its space complexity in Section 3.3 before presenting our experimental results in Section 3.4 and our summary in Section 3.5.

3.1 Motivation

Researchers have concluded that, within the context of centralized search algorithms, depth-first branch-and-bound searches are often faster than memory-bounded best-first searches when solving problems with bounded-depth search trees (Zhang & Korf, 1995). As it turns out, DCOP problems are problems with bounded-depth search trees – the depth of search trees of DCOP problems is bounded by twice the number of agents in the problem. Therefore, I hypothesize that DCOP search algorithms that employ depth-first branch-and-bound search can be faster than DCOP search algorithms that employ memory-bounded best-first search.

However, this hypothesis is not necessarily true since the conclusions in (Zhang & Korf, 1995) were for centralized search algorithms. In fact, there exist a DCOP search algorithm that employs depth-first branch-and-bound search, namely SBB, that is slower than a DCOP search algorithm that employ memory-bounded best-first search, namely ADOPT (Modi et al., 2005). However, it is difficult to determine which one is the better search strategy since both algorithms differ by more than their search strategies, as shown in Table 2.1. In fact, Table 2.1 shows that existing DCOP search algorithms that employ depth-first branch-and-bound search, namely SBB, NCBB and AFB, have other properties aside from the search strategies that are different from those of ADOPT.

Therefore, I introduce Branch-and-Bound ADOPT (BnB-ADOPT), a DCOP search algorithm that uses the framework of ADOPT and only changes the search strategy of ADOPT from memory-bounded best-first search to depth-first branch-and-bound search. Thus, the comparison of the runtimes of ADOPT and BnB-ADOPT will experimentally

assess my hypothesis. This work is non-trivial since ADOPT is a rather complicated algorithm whose agents operate concurrently and asynchronously at all times. The agents can only perform local searches, yet must follow a global search strategy.

3.2 BnB-ADOPT

We now introduce our new algorithm Branch-and-Bound ADOPT (BnB-ADOPT). BnB-ADOPT has the same memory requirement, observes the same communication restriction and uses the same agent operation and pseudo-tree structure as ADOPT, but employs a depth-first branch-and-bound search strategy instead of a memory-bounded best-first search strategy. We do not describe BnB-ADOPT as a modification of ADOPT since this approach requires the readers to have an in-depth understanding of ADOPT. Instead, we give a stand-alone description of BnB-ADOPT that requires no knowledge of ADOPT, with the intention of creating a self-contained and hopefully easy-to-read section.

3.2.1 Notations and Key Terms

We adopt the following notation from ADOPT to describe BnB-ADOPT.

- $ValInit(a) \in Dom(a)$ is the initial value of agent $a \in A$;
- $CD(a) \subseteq A$ is the set of child and pseudo-child agents of agent $a \in A$;
- $C(a) \subseteq CD(a)$ is the set of child agents of agent $a \in A$;
- $pa(a) \in A$ is the parent agent of agent $a \in A$ except for the root agent;
- $P(a) \subseteq A$ is the set of ancestor agents (including the parent agent) of agent $a \in A$;

- $SCP(a) \subseteq P(a)$ is the set of ancestor agents (including the parent agent) of agent $a \in A$ that are parent or pseudo-parent agents of agent a or one (or more) of its descendant agents; and

- $CP(a) \subseteq SCP(a)$ is the set of ancestor agents (including the parent agent) of agent $a \in A$ that are parent or pseudo-parent agents of agent a.

We adopt the following key terms from ADOPT to describe BnB-ADOPT.

- **Context (X):** The context X^a of agent a is the set of values of all ancestor agents of agent a. The context X^r of the root agent r is always equal to $\{\}$.

- **Delta cost (δ):** The delta cost $\delta^a_{X^a}(d)$ is the sum of the constraint costs of all constraints that involve both agent a and one of its ancestor agents, under the assumption that agent a takes on value d and its ancestor agents take on the values in context X^a. In the search tree, $\delta^a_{X^a}(d)$ is the delta cost of the AND node that has partial solution $X^a \cup (a, d)$. For example, $\delta^{a2}_{\{(a_1,1)\}}(1)$ is the delta cost of node f in Figure 2.3.

- **Gamma cost (γ):** The gamma costs $\gamma^a_{X^a}(d)$ and $\gamma^a_{X^a}$ are defined as follows:

$$\gamma^a_{X^a}(d) := \delta^a_{X^a}(d) + \sum_{c \in C(a)} \gamma^c_{X^a \cup (a,d)} \tag{3.1}$$

$$\gamma^a_{X^a} := \min_{d \in Dom(a)} \{\gamma^a_{X^a}(d)\} \tag{3.2}$$

for all agents a, all values d and all contexts X^a. Thus, the gamma cost $\gamma^a_{X^a}(d)$ is the sum of the constraint costs of all constraints that involve agent a or one of its descendant agents (that is, either both agent a and one of its ancestor agents, both agent a and one of its descendant agents, both a descendant agent and an ancestor agent of agent a or two descendant agents of agent a) minimized over all possible values of its descendant agents, under the assumption that agent a takes on value d and its ancestor agents take on the values in context X^a. In the search tree, $\gamma^a_{X^a}(d)$ is the gamma cost of the AND node that has partial solution $X^a \cup (a, d)$. For example, $\gamma^{a_2}_{\{(a_1,1)\}}(1)$ is the gamma cost of node f in Figure 2.3. The gamma cost $\gamma^a_{X^a}$ is the sum of the constraint costs of all constraints that involve agent a or one of its descendant agents minimized over all possible values of agent a and its descendant agents, under the assumption that the ancestor agents of agent a take on the values in context X^a. In the search tree, the gamma cost $\gamma^a_{X^a}$ is the gamma cost of the OR node whose agent is agent a and whose parent AND node has partial solution X^a. For example, $\gamma^{a_2}_{\{(a_1,1)\}}$ is the gamma cost of node C in Figure 2.3. Therefore, the gamma cost of an AND node is the sum of its delta cost and the gamma costs of its child OR nodes, and the gamma cost of an OR node is the minimum of the gamma costs of its child AND nodes. For example, the gamma cost of node f in Figure 2.3 is the sum of its delta cost and the gamma costs of nodes J and K, and the gamma cost of node C in Figure 2.3 is the minimum of the gamma costs of nodes e and f.

Solving a DCOP problem optimally means to determine $\gamma^r_{X^r}$ for the root agent r or, equivalently, the gamma cost of the root node since $\gamma^r_{X^r}$ is the minimal solution cost. It is not difficult for the agents to cache information that allows them to determine a cost-minimal solution.

3.2.2 Updating the Bounds

Every agent a in BnB-ADOPT stores and updates several bounds on the gamma costs, namely $lb^{a,c}_{X^a}(d)$, $LB^a_{X^a}(d)$, $LB^a_{X^a}$, $ub^{a,c}_{X^a}(d)$, $UB^a_{X^a}(d)$ and $UB^a_{X^a}$ for all values d, all child agents c and all contexts X^a, maintaining the following bound property:

$$LB^a_{X^a} \leq \quad \gamma^a_{X^a} \quad \leq UB^a_{X^a} \tag{3.3}$$

$$LB^a_{X^a}(d) \leq \quad \gamma^a_{X^a}(d) \quad \leq UB^a_{X^a}(d) \tag{3.4}$$

$$lb^{a,c}_{X^a}(d) \leq \quad \gamma^c_{X^a \cup (a,d)} \quad \leq ub^{a,c}_{X^a}(d) \tag{3.5}$$

In the search tree,

- $LB^a_{X^a}$ and $UB^a_{X^a}$ are lower and upper bounds, respectively, (on the gamma cost) of the OR node whose agent is agent a and whose parent AND node has partial solution X^a;
- $LB^a_{X^a}(d)$ and $UB^a_{X^a}(d)$ are lower and upper bounds, respectively, (on the gamma cost) of the AND node that has partial solution $X^a \cup (a, d)$; and

- $lb^{a,c}_{X^a}(d)$ and $ub^{a,c}_{X^a}(d)$ are lower and upper bounds, respectively, (on the gamma cost) of the OR node whose agent is agent c and whose parent AND node has partial solution $X^a \cup (a, d)$.

For example, $LB^{a_2}_{\{(a_1,1)\}}$ and $UB^{a_2}_{\{(a_1,1)\}}$ are bounds of node C in Figure 2.3, $LB^{a_2}_{\{(a_1,1)\}}(1)$ and $UB^{a_2}_{\{(a_1,1)\}}(1)$ are bounds of node f, and $lb^{a_2,a_3}_{\{(a_1,1)\}}(1)$ and $ub^{a_2,a_3}_{\{(a_1,1)\}}(1)$ are bounds of node J. $lb^{a_2,a_3}_{\{(a_1,1)\}}(1)$, $ub^{a_2,a_3}_{\{(a_1,1)\}}(1)$, $LB^{a_3}_{\{(a_1,1),(a_2,1)\}}$ and $UB^{a_3}_{\{(a_1,1),(a_2,1)\}}$ are bounds of node J, but agent a_2 maintains the first two bounds while agent a_3 maintains the last two bounds.

Each agent a uses the following update equations for all values d, all child agents c and all contexts X^a to initialize its bounds $lb^{a,c}_{X^a}(d)$ and $ub^{a,c}_{X^a}(d)$, where the heuristic values $h^{a,c}_{X^a}(d)$ are floating point numbers that are admissible and thus satisfy $0 \leq h^{a,c}_{X^a}(d) \leq \gamma^c_{X^a \cup (a,d)}$:

$$lb^{a,c}_{X^a}(d) := h^{a,c}_{X^a}(d) \tag{3.6}$$

$$ub^{a,c}_{X^a}(d) := \infty \tag{3.7}$$

Agent a then uses repeatedly the following update equations for all values d, all child agents c, all contexts X^a and all contexts X^c $(= X^a \cup (a, d))$ to tighten the bounds:

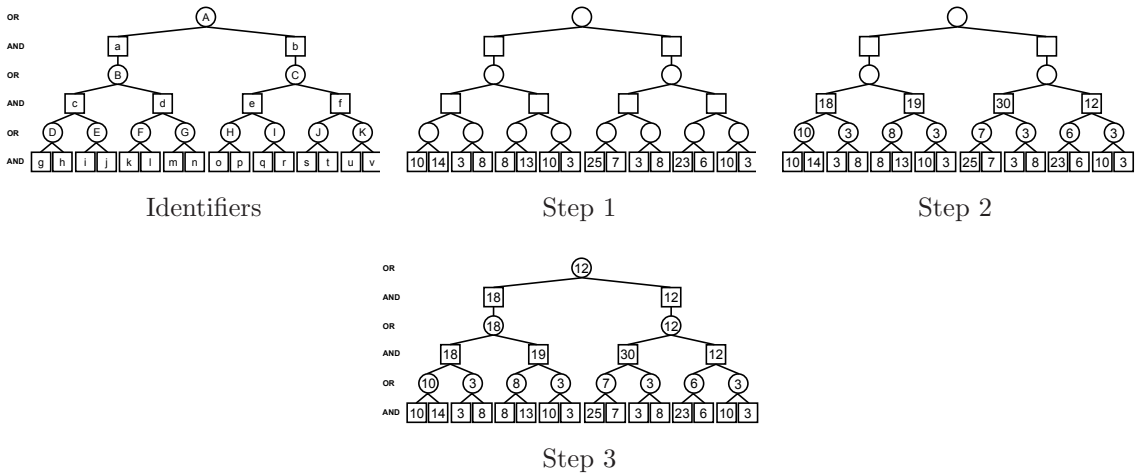


Figure 3.1: Simplified Trace of the Updates of the (Lower and Upper) Bounds

$$lb^{a,c}_{X^a}(d) := \max\{lb^{a,c}_{X^a}(d), LB^c_{X^c}\} \tag{3.8}$$

$$LB^a_{X^a}(d) := \delta^a_{X^a}(d) + \sum_{c \in C(a)} lb^{a,c}_{X^a}(d) \tag{3.9}$$

$$LB^a_{X^a} := \min_{d \in Dom(a)} \{LB^a_{X^a}(d)\} \tag{3.10}$$

$$ub^{a,c}_{X^a}(d) := \min\{ub^{a,c}_{X^a}(d), UB^c_{X^c}\} \tag{3.11}$$

$$UB^a_{X^a}(d) := \delta^a_{X^a}(d) + \sum_{c \in C(a)} ub^{a,c}_{X^a}(d) \tag{3.12}$$

$$UB^a_{X^a} := \min_{d \in Dom(a)} \{UB^a_{X^a}(d)\} \tag{3.13}$$

The updates maintain the bound property and improve the bounds monotonically, that is, the lower bounds are monotonically non-decreasing and the upper bounds are monotonically non-increasing.[1] After a finite amount of time, $UB^a_{X^a} \leq LB^a_{X^a}$ for all

[1]Leaf agents use the same update equations. Since they do not have child agents, the sums over their child agents evaluate to 0. For example, $LB^a_{X^a}(d) = UB^a_{X^a}(d) = \delta^a_{X^a}(d)$ for all leaf agents a, all values d and all contexts X^a.

agents a and all contexts X^a. BnB-ADOPT terminates when its termination condition $UB^r_{X^r} \leq LB^r_{X^r}$ for the root agent r is satisfied. Then, $UB^r_{X^r} \leq LB^r_{X^r}$ and the bound property $UB^r_{X^r} \geq LB^r_{X^r}$ together imply that $UB^r_{X^r} = \gamma^r_{X^r} = LB^r_{X^r}$, and the DCOP problem is solved optimally.

Figure 3.1 shows a simplified trace of the updates of the (lower and upper) bounds for our example DCOP problem. We assume that the updates proceed sequentially from the leaf agents to the root agent. Due to this simplification, the lower and upper bounds of each node are identical to its gamma cost and independent of the heuristic values. The numbers in the nodes are their bounds. Two agents maintain the bounds of OR nodes except for the root node. The figure shows the bounds that the parent agent maintains rather than the bounds that the child agent maintains. For example, the number in node B is the bounds that agent a_1 rather than agent a_2 maintains. The bounds that the child agent maintains can be computed by taking the minimum of the bounds of the child AND nodes of the OR node. Agents update the bound of an AND node to the sum of its delta cost and the bounds of its child OR nodes according to update equations 3.9 and 3.12. They update the bound of an OR node to the minimum of the bounds of its child AND nodes according to update equations 3.10 and 3.13. A more detailed description of the trace is as follows:

- **Step 1:** Leaf agent a_3 updates the bounds of AND nodes g, h, k, l, o, p, s and t to their delta costs according to update equations 3.9 and 3.12 and the bounds of OR nodes D, F, H and J to the minimum of the bounds of their child AND nodes according to update equations 3.10 and 3.13. Similarly, leaf agent a_4 updates the

bounds of AND nodes i, j, m, n, q, r, u and v to their delta costs according to update equations 3.9 and 3.12 and the bounds of OR nodes E, G, I and K to the minimum of the bounds of their child AND nodes according to update equations 3.10 and 3.13. The bounds of OR nodes D to K are not shown in the figure since they are not (yet) maintained by agent a_2.

- **Step 2:** Agent a_2 updates the bounds of OR nodes D to K that it maintains to the bounds of the same OR nodes that leaf agents a_3 and a_4 maintain according to update equations 3.8 and 3.11, the bounds of AND nodes c to f to the sum of their delta costs and the bounds of their child OR nodes according to update equations 3.9 and 3.12 and the bounds of OR nodes B and C to the minimum of the bounds of their child AND nodes according to update equations 3.10 and 3.13. The bounds of OR nodes B and C are not shown in the figure since they are not (yet) maintained by agent a_1.

- **Step 3:** Agent a_1 updates the bounds of OR nodes B and C that it maintains to the bounds of the same OR nodes that agent a_2 maintains according to update equations 3.8 and 3.11, the bounds of AND nodes a and b to the sum of their delta costs and the bounds of their child OR nodes according to update equations 3.9 and 3.12 and the bounds of OR node A to the minimum of the bounds of its child AND nodes according to update equations 3.10 and 3.13. Since the lower and upper bounds of a node are equal to its gamma cost, the lower and upper bounds of the root node are equal to its gamma cost, which in turn is equal to the minimal solution cost. The propagation terminates after three steps with minimal solution cost 12.

3.2.3 Adhering to the Memory Limitations

Our description of BnB-ADOPT so far assumes no memory limitations. However, BnB-ADOPT is a memory-bounded DCOP search algorithm with memory requirements per agent that are linear in the number of agents. We now describe how BnB-ADOPT adheres to these memory limitations using techniques that were introduced for ADOPT but apply to BnB-ADOPT as well.

The simplified trace in Figure 3.1 assumes that every agent a maintains its bounds for all values d, all child agents c and all contexts X^a. The number of contexts can be exponential in the depth of the agent in the pseudo-tree. For our example DCOP problem, agent a_3 has four different contexts for the four different combinations of values of its ancestor agents a_1 and a_2. An agent cannot maintain an exponential number of bounds due to the memory limitations. Therefore, every agent maintains its bounds for only one context at any given time. This context is stored in the variable X^a for agent a. The size of the context is at most linear in the number of agents. The number of bounds of an agent is now linear in the product of its domain cardinality and the number of its child agents. Thus, the memory requirements per agent are only linear in the number of agents if the domain cardinality and the magnitude of the bounds (and the other variables) are constant for each agent.

3.2.4 Performing Depth-First Search

Our description of BnB-ADOPT so far applies to ADOPT as well. However, BnB-ADOPT uses depth-first branch-and-bound search and ADOPT uses best-first search. We now describe how BnB-ADOPT implements depth-first search.

Agents in BnB-ADOPT send messages that are similar to that of ADOPT but processes them differently. They send messages of three different types, namely VALUE, COST and TERMINATE messages. At the start, every agent a initializes its context X^a, uses update equations 3.6, 3.9, 3.10, 3.7, 3.12 and 3.13 to initialize its bounds and takes on its best value $d^a := \arg\min_{d \in Dom(a)} \{LB^a_{X^a}(d)\}$. It sends VALUE messages to all child agents and a COST message to its parent agent. It then repeatedly waits for incoming messages, processes them, possibly takes on a different value and again sends VALUE messages to all child agents and a COST message to its parent agent. A description of the three message types and how agents process them is as follows:

- **VALUE messages:** An agent a with context X^a and value d^a sends VALUE messages to all child agents with the desired context $X^a \cup (a, d^a)$, which is its context augmented with its value. Leaf agents do not have child agents and thus do not send VALUE messages. VALUE messages thus propagate contexts down the pseudo-tree.

  When an agent receives a VALUE message, it checks whether its context is identical to the desired context in the VALUE message. If it is not, then the agent changes its context to the desired context in the VALUE message. In either case, it then executes the common program (see below).

- **COST messages:** An agent a sends COST messages to its parent agent with its identity a, its context X^a and its bounds $LB^a_{X^a}$ and $UB^a_{X^a}$. The root agent does not have a parent agent and thus does not send COST messages. COST messages thus propagate bounds up the pseudo-tree.

When an agent receives a COST message, it checks whether its context and the context in the COST message are compatible. Two contexts are compatible if no agent takes on different values in the two contexts. If they are, then the agent uses update equations 3.8 to 3.13 with the bounds in the COST message to improve its bounds for the value in the message. In either case, it then executes the common program (see below).

- **TERMINATE messages:** When the termination condition $UB^r_{X^r} \leq LB^r_{X^r}$ is satisfied, the root agent r sends TERMINATE messages (without parameters) to all child agents to inform them that the search is complete and then terminates. When an agent receives such a TERMINATE message, it sends TERMINATE messages to all child agents and terminates as well. Leaf agents do not have child agents and thus do not send TERMINATE messages. TERMINATE messages thus propagate down the pseudo-tree until all agents terminate.

The common program is as follows:

- **Context change:** If an agent a changed its context X^a, it executes the following statements: It uses update equations 3.6, 3.9, 3.10, 3.7, 3.12 and 3.13 to initialize its bounds and takes on its best value $d^a := \arg\min_{d \in Dom(a)} \{LB^a_{X^a}(d)\}$. It then sends VALUE messages to all child agents and a COST message to its parent agent.

- **No context change:** If an agent a did not change its context X^a, it executes the following statements: If $UB^a_{X^a} \leq LB^a_{X^a}(d^a)$ for its value d^a, then the context of the agent augmented with its value cannot be completed to a solution whose solution cost is smaller than the solution cost of the best solution found so far

for its context X^a ($= UB^a_{X^a}$) and the agent thus takes on its best value $d^a := \arg\min_{d \in Dom(a)}\{LB^a_{X^a}(d)\}$. It then sends VALUE messages to all child agents and a COST message to its parent agent.

Assume that the context X^a of an agent a does not change. After a finite amount of time, $UB^a_{X^a} \leq LB^a_{X^a}(d^a)$ for its value d^a. The agent then takes on its best value and repeats the procedure. After a finite amount of time, $UB^a_{X^a} \leq LB^a_{X^a}(d)$ for all values d, which implies that $UB^a_{X^a} \leq LB^a_{X^a}$. The agent takes on every value d at most once until $UB^a_{X^a} \leq LB^a_{X^a}$ since $LB^a_{X^a}(d)$ remains unchanged and $UB^a_{X^a}$ is monotonically non-increasing once the agent changes its value from d to a different value, which prevents the agent from changing its value back to d before $UB^a_{X^a} \leq LB^a_{X^a}$. BnB-ADOPT thus performs depth-first search. Then, after a finite amount of time, $UB^r_{X^r} \leq LB^r_{X^r}$ and the bound property $UB^r_{X^r} \geq LB^r_{X^r}$ together imply that $UB^r_{X^r} = \gamma^r_{X^r} = LB^r_{X^r}$ for the root agent r, and the DCOP problem is solved optimally.

Figures 3.2 and 3.3 show traces of the updates of the lower and upper bounds, respectively, for our example DCOP problem. BnB-ADOPT uses the zero heuristic values. The initial context of every agent assigns value 0 to all ancestor agents of the agent. We partition time into cycles. Agents maintain their bounds for only one context at any given time. Nodes in the figures are crossed out if their agent does not maintain their bounds. AND nodes are shaded if their partial solution is equal to the context of the agent of their parent OR node augmented with its value. For example, agents a_1, a_3 and a_4 take on value 0 in Cycle 2, and agent a_2 takes on value 1. The context of agent a_1 is $\{\}$, the

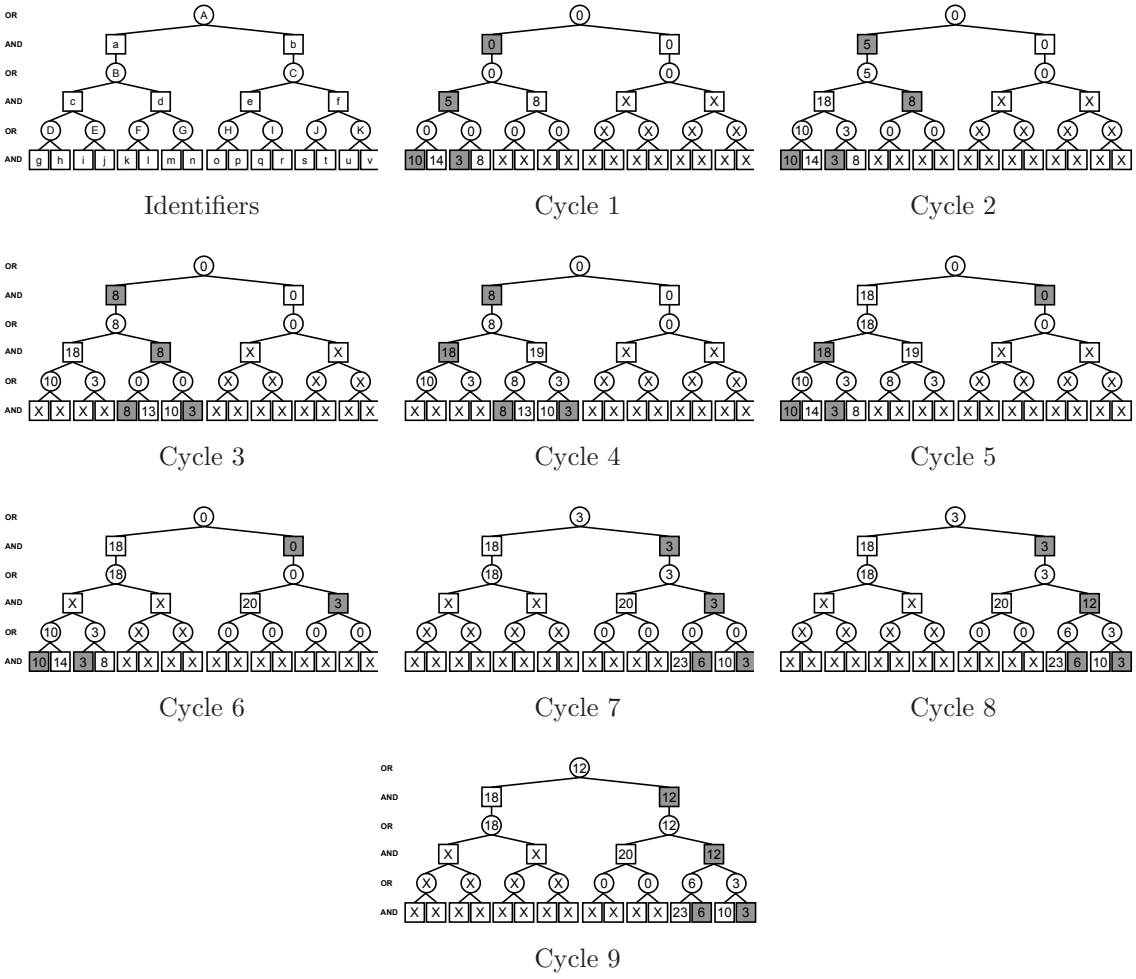


Figure 3.2: Trace of the Updates of the Lower Bounds

context of agent a_2 is $\{(a_1, 0)\}$ and the contexts of agents a_3 and a_4 are $\{(a_1, 0), (a_2, 0)\}$. A description of the trace is as follows:

- **Cycle 1:** Root agent a_1 initializes its context X^{a_1} to $\{\}$. It initializes the lower bounds of nodes B $(= lb^{a_1,a_2}_{X^{a_1}}(0))$ and C $(= lb^{a_1,a_2}_{X^{a_1}}(1))$ to 0 since it uses the zero heuristic values. It updates the lower bound of node a $(= LB^{a_1}_{X^{a_1}}(0))$ to the sum of its delta cost $(= 0)$ and the lower bound of node B $(= 0)$ according to the update equations. It updates the lower bound of node b $(= LB^{a_1}_{X^{a_1}}(1))$ to the sum of its delta cost $(= 0)$ and the lower bound of node C $(= 0)$ according to the update

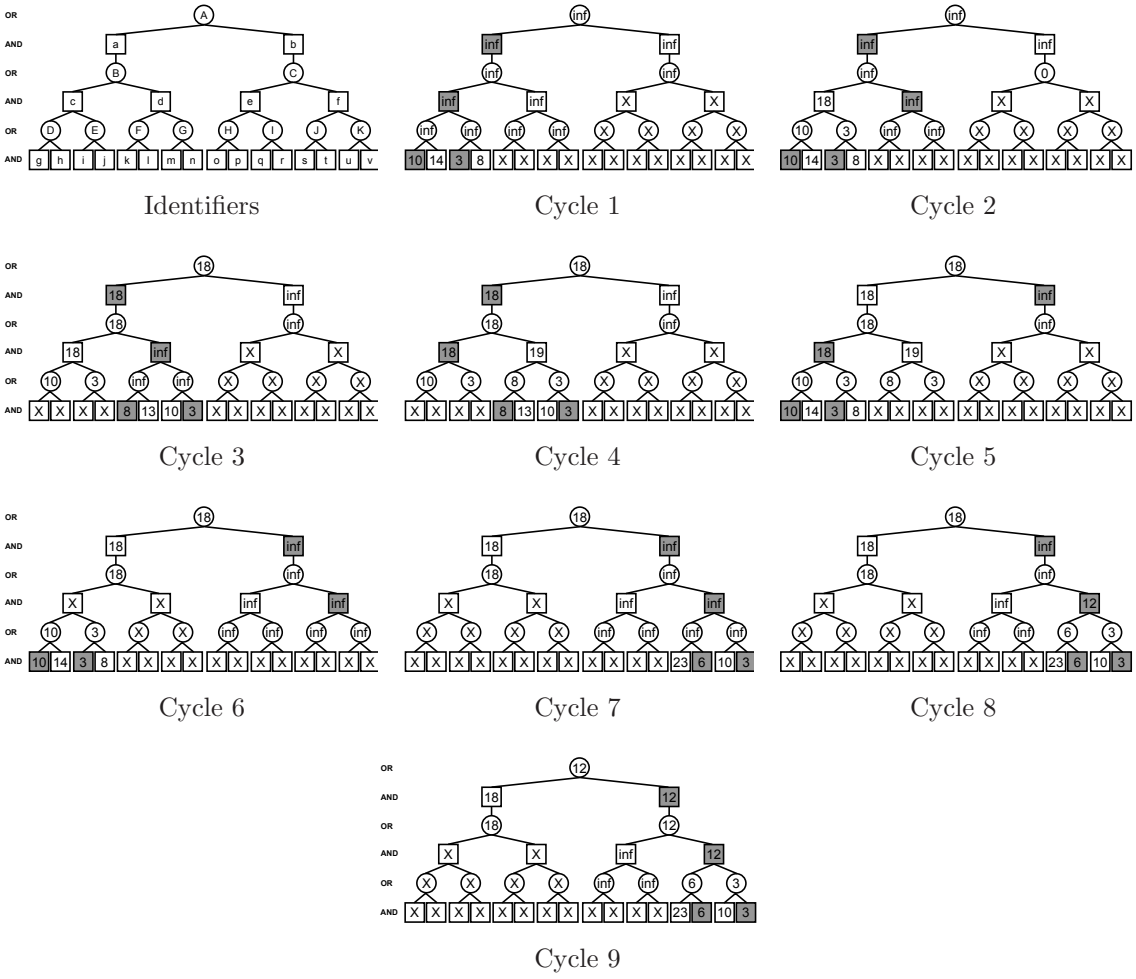


Figure 3.3: Trace of the Updates of the Upper Bounds

equations. It updates the lower bound of node A ($= LB^{a_1}_{X^{a_1}}$) to the minimum of the lower bound of node a ($= 0$) and the lower bound of node b ($= 0$) according to the update equations. It initializes the upper bounds of nodes B and C to infinity. It updates the upper bounds of nodes a, b and A to infinity according to the update equations. It takes on its best value. It can take on either value 0 or value 1 since the lower bounds of nodes a and b are both 0. It takes on value 0 and sends a VALUE message to its child agent a_2.

Agent a_2 initializes its context X^{a_2} to $\{(a_1, 0)\}$. It initializes the lower bounds of nodes D, E, F and G to 0. It updates the lower bounds of nodes c, d and B to 5, 8 and 5, respectively. It initializes the upper bounds of nodes D, E, F and G to infinity. It updates the upper bounds of nodes c, d and B to infinity. The bounds of node B that agent a_2 maintains are not shown in the figures. It takes on its best value 0, sends VALUE messages to its child agents a_3 and a_4 and sends a COST message to its parent agent a_1.

Leaf agent a_3 initializes its context X^{a_3} to $\{(a_1, 0), (a_2, 0)\}$. It updates the lower bounds of nodes g and h to their delta costs 10 and 14, respectively, since leaf agents do not have child agents. It updates the lower bound of node D to 10. It updates the upper bounds of nodes g and h to their delta costs 10 and 14, respectively, since leaf agents do not have child agents. It updates the upper bound of node D to 10. The bounds of node D that leaf agent a_3 maintains are not shown in the figures. It takes on its best value 0 and sends a COST message to its parent agent a_2.

Leaf agent a_4 initializes its context X^{a_4} to $\{(a_1, 0), (a_2, 0)\}$. It updates the lower bounds of nodes i and j to their delta costs 3 and 8, respectively. It updates the lower bound of node E to 3. It updates the upper bounds of nodes i and j to their delta costs 3 and 8, respectively. It updates the upper bound of node E to 3. The bounds of node E that leaf agent a_4 maintains are not shown in the figures. It takes on its best value 0 and sends a COST message to its parent agent a_2.

In summary, the following messages are sent during Cycle 1:

- message (VALUE, $\{(a_1, 0)\}$) from agent a_1 to agent a_2;
- message (VALUE, $\{(a_1, 0), (a_2, 0)\}$) from agent a_2 to agent a_3;
- message (VALUE, $\{(a_1, 0), (a_2, 0)\}$) from agent a_2 to agent a_4;
- message (COST, a_2, $\{(a_1, 0)\}$, 5, ∞) from agent a_2 to agent a_1;
- message (COST, a_3, $\{(a_1, 0), (a_2, 0)\}$, 10, 10) from agent a_3 to agent a_2; and
- message (COST, a_4, $\{(a_1, 0), (a_2, 0)\}$, 3, 3) from agent a_4 to agent a_2.

- **Cycle 2:** Root agent a_1 receives the COST message sent by its child agent a_2 in Cycle 1. Since the context of agent a_1 (= {}) is compatible with the context in the message (= $\{(a_1, 0)\}$), it improves its bounds. It updates the bounds of node B to the bounds in the message (= 5 and infinity, respectively). It updates the bounds of nodes a, b and A. It does not change its value since the lower bound of node a ($= LB^{a_1}_{X^{a_1}}(d^{a_1}) = 5$ for its value $d^{a_1} = 0$) is still smaller than the upper bound of node A ($= UB^{a_1}_{X^{a_1}} = \infty$). It sends a VALUE message to its child agent a_2.

  Agent a_2 receives the VALUE message sent by its parent agent a_1 in Cycle 1. Its context (= $\{(a_1, 0)\}$) remains unchanged since it is the same as the desired context in the message (= $\{(a_1, 0)\}$). Agent a_2 also receives the COST messages sent by its child agents a_3 and a_4 in Cycle 1. Since the context of agent a_2 (= $\{(a_1, 0)\}$) is compatible with the contexts in the messages (= $\{(a_1, 0), (a_2, 0)\}$), it improves its bounds. It updates the bounds of node D to the bounds in the first message (= 10 and 10, respectively) and the bounds of node E to the bounds in the second

message (= 3 and 3, respectively). It updates the bounds of nodes c, d and B. It changes its value since the lower bound of node c ($= LB^{a_2}_{X^{a_2}}(d^{a_2}) = 18$ for its value $d^{a_2} = 0$) is no longer smaller than the upper bound of node B ($= UB^{a_2}_{X^{a_2}} = 18$). It takes on its best value 1, sends VALUE messages to its child agents a_3 and a_4 and sends a COST message to its parent agent a_1.

Leaf agents a_3 and a_4 receive the VALUE messages sent by their parent agent a_2 in Cycle 1. Their contexts ($= \{(a_1, 0), (a_2, 0)\}$) remain unchanged since they are the same as the desired context in the message ($= \{(a_1, 0), (a_2, 0)\}$). They send the same COST messages as before to their parent agent a_2.

In summary, the messages sent during Cycle 2 are identical to the ones sent during Cycle 1, except for the messages sent by agent a_2, which are as follows:

- message (VALUE, $\{(a_1, 0), (a_2, 1)\}$) from agent a_2 to agent a_3;
- message (VALUE, $\{(a_1, 0), (a_2, 1)\}$) from agent a_2 to agent a_4; and
- message (COST, a_2, $\{(a_1, 0)\}$, 8, 18) from agent a_2 to agent a_1.

The VALUE messages are different because agent a_2 changed its value from 0 to 1. The COST message is different because agent a_2 changed its bounds.

- **Cycles 3-9:** The messages sent during Cycle 3 are identical to the ones sent during Cycle 2, except for the messages sent by agents a_3 and a_4, which are as follows:
  - message (COST, a_3, $\{(a_1, 0), (a_2, 1)\}$, 8, 8) from agent a_3 to agent a_2; and
  - message (COST, a_4, $\{(a_1, 0), (a_2, 1)\}$, 3, 3) from agent a_4 to agent a_2.

The COST messages are different because agents a_3 and a_4 changed their contexts. The termination condition holds after a finite amount of time when the upper bound of node A ($= UB^{a_1}_{X^{a_1}} = 12$) is no larger than the lower bound of node A ($= LB^{a_1}_{X^{a_1}} = 12$). Root agent a_1 sends TERMINATE messages to all child agents, and the TERMINATE messages propagate down the pseudo-tree until all agents terminate. BnB-ADOPT terminates after nine cycles with minimal solution cost 12.

3.2.5 Performing Branch-and-Bound Search

The operation of the agent as described in Section 3.2.4 can be further improved. We now refine our description of BnB-ADOPT by explaining how the agents implement branch-and-bound search to make BnB-ADOPT faster. Every agent a in BnB-ADOPT now also maintains the variable threshold $TH^a_{X^a}$, which it initializes to infinity. The threshold of the root agent always remains infinity. Every other agent uses its threshold for pruning, meaning that it can change its value earlier than previously.

- **First change:** If an agent a did not change its context X^a, it previously executed the following statements: If $UB^a_{X^a} \leq LB^a_{X^a}(d^a)$ for its value d^a, then the agent took on its best value. It then sent VALUE messages to all child agents and a COST message to its parent agent. Now, if $TH^a_{X^a} \leq LB^a_{X^a}(d^a)$, then the agent also takes on its best value. Thus, if $\min\{TH^a_{X^a}, UB^a_{X^a}\} \leq LB^a_{X^a}(d^a)$, then the agent takes on its best value and thus potentially changes its value, which is earlier than previously. $\min\{TH^a_{X^a}, UB^a_{X^a}\}$ is the pruning quantity.

- **Second change:** An agent a with context X^a and value d^a sends VALUE messages to all its child agents, which previously contained only the desired context $X^a \cup (a, d^a)$. VALUE messages now also contain the desired threshold $\min\{TH^a_{X^a}, UB^a_{X^a}\} - \delta^a_{X^a}(d^a) - \sum_{c' \in C(a) \setminus c} lb^{a,c'}_{X^a}(d^a)$ for the child agent c. When agent c receives a VALUE message, it sets its threshold to the desired threshold and then proceeds as described earlier. The desired threshold is set such that the lower bound $LB^a_{X^a}(d^a)$ of agent a for its value d^a reaches its pruning quantity (and agent a thus potentially changes its value) when the lower bound $LB^c_{X^c}$ of agent c reaches the desired threshold. This property can be verified as follows:

$$LB^c_{X^c} \geq \min\{TH^a_{X^a}, UB^a_{X^a}\} - \delta^a_{X^a}(d^a) - \sum_{c' \in C(a) \setminus c} lb^{a,c'}_{X^a}(d^a) \quad (3.14)$$

$$lb^{a,c}_{X^a}(d^a) \geq \min\{TH^a_{X^a}, UB^a_{X^a}\} - \delta^a_{X^a}(d^a) - \sum_{c' \in C(a) \setminus c} lb^{a,c'}_{X^a}(d^a) \quad (3.15)$$

$$-\min\{TH^a_{X^a}, UB^a_{X^a}\} \geq -\delta^a_{X^a}(d^a) - lb^{a,c}_{X^a}(d^a) - \sum_{c' \in C(a) \setminus c} lb^{a,c'}_{X^a}(d^a) \quad (3.16)$$

$$\min\{TH^a_{X^a}, UB^a_{X^a}\} \leq \delta^a_{X^a}(d^a) + lb^{a,c}_{X^a}(d^a) + \sum_{c' \in C(a) \setminus c} lb^{a,c'}_{X^a}(d^a) \quad (3.17)$$

$$\min\{TH^a_{X^a}, UB^a_{X^a}\} \leq \delta^a_{X^a}(d^a) + \sum_{c' \in C(a)} lb^{a,c'}_{X^a}(d^a) \quad (3.18)$$

$$\min\{TH^a_{X^a}, UB^a_{X^a}\} \leq LB^a_{X^a}(d^a) \quad (3.19)$$

3.2.6 Further Enhancements

We continue to refine our description of BnB-ADOPT by explaining a number of additional enhancements, which were introduced for ADOPT.

- **Reduced contexts:** The agents now use reduced contexts, which are subsets of the contexts described previously. The reduced context X_1^a of agent a contains the values of all ancestor agents $p \in SCP(a)$, while the context X_2^a described previously contains the values of all ancestor agents $p \in P(a)$. The agents can use reduced contexts since $\gamma^a_{X_1^a} = \gamma^a_{X_2^a}$ and $\gamma^a_{X_1^a}(d) = \gamma^a_{X_2^a}(d)$ for all values d. Agents now use reduced contexts because they need to change their contexts and thus initialize their bounds less often when they receive VALUE messages since their contexts are then more often identical to the desired contexts in the VALUE messages. For our example DCOP problem, the reduced context of agent a_4 contains the values of only agent a_2 rather than the values of agents a_1 and a_2. Therefore, the following pairs of nodes in the search tree are actually the same node: nodes i and q, nodes j and r, nodes m and u, and nodes n and v.

- **VALUE and COST messages:** An agent sends VALUE messages to all child agents, which previously contained the desired context and the desired threshold. The desired context is the context of the agent augmented with its value. When an agent receives a VALUE message, it previously checked whether its context is identical to the desired context in the VALUE message. If it was not, then the agent changed its context to the desired context in the VALUE message. Agents now update their contexts differently to reduce the size of the VALUE messages. An

agent sends VALUE messages to all child and pseudo-child agents with its identity, value and desired threshold, which is infinity for its pseudo-child agents. When an agent receives a VALUE message, it sets its threshold to the desired threshold if the message is from its parent agent. It also checks whether the value of the ancestor agent in the VALUE message is more recent than the value of the ancestor agent in its context. If it is, then the agent changes the value of the ancestor agent in its context to the value of the ancestor agent in the VALUE message. However, the context of an agent does not only contain the values of its parent and pseudo-parent agents but also the values of its ancestor agents that are the parent or pseudo-parent agents of one (or more) of its descendant agents, and ancestor agents that are not constrained with the agent cannot send VALUE messages to the agent. However, they send VALUE messages to their pseudo-child agents, at least one of which is a descendant agent of the agent, and the information then propagates up the pseudo-tree with COST messages until it reaches the agent. When an agent receives a COST message, it now checks whether the value of an ancestor agent in the context of the COST message is more recent than the value of the ancestor agent in its context. If it is, then the agent changes the value of the ancestor agent in its context to the value of the ancestor agent in the context of the COST message. Our example DCOP problem is too simple to allow us to illustrate the propagation of the information up the pseudo-tree. However, imagine that a new agent a_5 is a child agent of agent a_4 and is constrained with agents a_1 and a_4. The context of agent a_4 then contains the value of agent a_1 but agent a_1 cannot send VALUE messages to agent a_4. However, agent a_1 sends VALUE messages to agent a_5. Agent a_5 changes

the value of agent a_1 in its context and sends COST messages with its context to agent a_4, which then changes the value of agent a_1 in its context as well.

The agents now need to determine whether the value of an agent in VALUE messages or in the contexts of COST messages is more recent than the value of the agent in their contexts. Every agent a therefore now also maintains a counter ID^a and increments it whenever it changes its value. Therefore, a larger ID indicates a more recent value. The values of agents in contexts are now labeled with their IDs, and VALUE messages contain the identity of the sending agent, its value, its ID and the desired threshold.

- **Bounds:** Whenever an agent changes its context X^a, it previously initialized its bounds and took on its best value. The (reduced) context of a child agent of an agent can now be a strict subset of the (reduced) context of the agent since the parent or some pseudo-parent agents of the agent might not be (parent or) pseudo-parent agents of the child agent or its descendant agents. If the context of child agent c does not contain the values of any agents whose values changed in the context of agent a, then agent a does not initialize its lower bounds $lb^{a,c}_{X^a}(d)$ and upper bounds $ub^{a,c}_{X^a}(d)$ for agent c and all values d before it takes on its best value. Agents use this optimization because they need to initialize their bounds less often this way. For our example DCOP problem, if agent a_2 changes its context from $\{(a_1, 0)\}$ to $\{(a_1, 1)\}$ (where the IDs are omitted for simplicity), then it does not initialize its lower bounds $lb^{a_2,a_4}_{X^{a_2}}(d)$ and upper bounds $ub^{a_2,a_4}_{X^{a_2}}(d)$ for child agent a_4

and all values d since the context of agent a_4 does not contain the value of agent a_1.

Additionally, if an agent a changes its context due to a COST message from its child agent c and its new context X^a is compatible with the context in the COST message, then agent a can set its lower bound $lb^{a,c}_{X^a}(d)$ and upper bound $ub^{a,c}_{X^a}(d)$ for agent c and the value d of agent a in the COST message to the bounds in the COST message before it takes on its best value. Agents use this optimization because the bounds in the COST message are more informed than the initialized bounds. Our example DCOP problem is too simple to allow us to illustrate this optimization. However, imagine again that a new agent a_5 is a child agent of agent a_4 and is constrained with agents a_1 and a_4. Assume that the context of agent a_4 is $\{(a_1, 0), (a_2, 0)\}$ (where the IDs are again omitted for simplicity) and it receives a COST message from agent a_5 with context $\{(a_1, 1), (a_4, 0)\}$. Agent a_4 then changes its context to $\{(a_1, 1), (a_2, 0)\}$, sets its lower bound $lb^{a_4,a_5}_{\{(a_1,1),(a_2,0)\}}(0)$ and its upper bound $ub^{a_4,a_5}_{\{(a_1,1),(a_2,0)\}}(0)$ to the bounds in the COST message and initializes all other bounds before it takes on its best value.

3.2.7 Pseudocode

Figure 3.4 shows the BnB-ADOPT pseudocode of every agent. The pseudocode does not index variables with the context since this context is implicitly given by the variable X^a. It uses the predicate Compatible$(X, X') = \neg\exists_{(a,d,ID)\in X,(a',d',ID')\in X'}(a = a' \wedge d \neq d')$ that determines if two contexts X and X' are compatible, that is, if no agent takes on two different values in the two contexts [Lines 35, 44, 46, 48 and 51]. The pseudocode

procedure Start()
[01] $X^a := \{(p,$ **ValInit**$(p), 0) \mid p \in SCP(a)\}$;
[02] $ID^a := 0$;
[03] **forall** $c \in C(a)$, $d \in Dom(a)$
[04] **InitChild**(c, d);
[05] **InitSelf**();
[06] **Backtrack**();
[07] **loop forever**
[08] **if**(message queue is not empty)
[09] **while**(message queue is not empty)
[10] pop msg off message queue;
[11] **When Received**(msg);
[12] **Backtrack**();

procedure InitChild(c, d)
[13] $lb^{a,c}(d) := h^{a,c}(d)$;
[14] $ub^{a,c}(d) := \infty$;

procedure InitSelf()
[15] $d^a := \arg\min_{d \in Dom(a)}\{\delta^a(d) + \sum_{c \in C(a)} lb^{a,c}(d)\}$;
[16] $ID^a := ID^a + 1$;
[17] $TH^a := \infty$;

procedure Backtrack()
[18] **forall** $d \in Dom(a)$
[19] $LB^a(d) := \delta^a(d) + \sum_{c \in C(a)} lb^{a,c}(d)$;
[20] $UB^a(d) := \delta^a(d) + \sum_{c \in C(a)} ub^{a,c}(d)$;
[21] $LB^a := \min_{d \in Dom(a)}\{LB^a(d)\}$;
[22] $UB^a := \min_{d \in Dom(a)}\{UB^a(d)\}$;
[23] **if**$(LB^a(d^a) \geq \min\{TH^a, UB^a\})$
[24] $d^a := \arg\min_{d \in Dom(a)}\{LB^a(d)\}$ (choose the previous d^a if possible);
[25] **if** a new d^a has been chosen
[26] $ID^a := ID^a + 1$;
[27] **if**((a is root and $UB^a \leq LB^a$) or termination message received)
[28] **Send**(TERMINATE) to each $c \in C(a)$;
[29] terminate execution;
[30] **Send**(VALUE, a, d^a, ID^a, $\min\{TH^a, UB^a\} - \delta^a(d^a) - \sum_{c' \in C(a) \setminus c} lb^{a,c'}(d^a)$) to each $c \in C(a)$;
[31] **Send**(VALUE, a, d^a, ID^a, ∞) to each $c \in CD(a) \setminus C(a)$;
[32] **Send**(COST, a, X^a, LB^a, UB^a) to $pa(a)$ if a is not root;

procedure When Received(VALUE, p, d^p, ID^p, TH^p)
[33] $X' := X^a$;
[34] **PriorityMerge**$((p, d^p, ID^p), X^a)$;
[35] **if**(!**Compatible**(X', X^a))
[36] **forall** $c \in C(a)$, $d \in Dom(a)$
[37] **if**$(p \in SCP(c))$
[38] **InitChild**(c, d);
[39] **InitSelf**();
[40] **if**$(p = pa(a))$
[41] $TH^a := TH^p$;

procedure When Received(COST, c, X^c, LB^c, UB^c)
[42] $X' := X^a$;
[43] **PriorityMerge**(X^c, X^a);
[44] **if**(!**Compatible**(X', X^a))
[45] **forall** $c \in C(a)$, $d \in Dom(a)$
[46] **if**(!**Compatible**$(\{(p, d^p, ID^p) \in X' \mid p \in SCP(c)\}, X^a)$)
[47] **InitChild**(c,d);
[48] **if**(**Compatible**(X^c, X^a))
[49] $lb^{a,c}(d) := \max\{lb^{a,c}(d), LB^c\}$ for the unique $(a', d, ID) \in X^c$ with $a' = a$;
[50] $ub^{a,c}(d) := \min\{ub^{a,c}(d), UB^c\}$ for the unique $(a', d, ID) \in X^c$ with $a' = a$;
[51] **if**(!**Compatible**(X', X^a))
[52] **InitSelf**();

procedure When Received(TERMINATE)
[53] record termination message received;

Figure 3.4: Pseudocode of BnB-ADOPT

also uses the procedure PriorityMerge(X, X') that executes $X' := \{(a', d', ID') \in X' \mid \neg\exists_{(a,d,ID)\in X}(a = a')\} \cup \{(a', d', ID') \in X' \mid \exists_{(a,d,ID)\in X}(a = a' \wedge ID \leq ID')\} \cup \{(a, d, ID) \in X \mid \exists_{(a',d',ID')\in X'}(a = a' \wedge ID > ID')\}$ and thus replaces the values of agents in context X' with more recent values, if available, of the same agents in context X [Lines 34 and 43].

The code is identical for every agent except that the variable a is a "self" variable that points to the agent itself. At the start, BnB-ADOPT calls Start() for every agent. When an agent a receives a VALUE message from an ancestor agent, then the "When Received" handler for VALUE messages is called with p being the ancestor agent, d^p being the value of the ancestor agent, ID^p being the ID of the ancestor agent and TH^p being the desired threshold for agent a if the ancestor agent is its parent agent (and infinity otherwise) [Line 11]. When agent a receives a COST message from a child agent, then the "When Received" handler for COST messages is called with c being the child agent, X^c being the context of the child agent, LB^c being the lower bound $LB^c_{X^c}$ of the child agent and UB^c being the upper bound $UB^c_{X^c}$ of the child agent [Line 11]. Finally, when agent a receives a TERMINATE message from its parent agent, then the "When Received" handler for TERMINATE messages is called without any arguments [Line 11].

BnB-ADOPT uses the same message passing and communication framework as ADOPT and has the same memory requirements. It uses similar VALUE, COST and TERMINATE messages, a similar strategy to update the context of an agent based on VALUE messages from its ancestor agents and COST messages from its child agents, the same semantics for the bounds and the same update equations to update these bounds. BnB-ADOPT and ADOPT both use thresholds but BnB-ADOPT uses the thresholds for pruning while ADOPT uses them to reconstruct partial solutions that were purged from

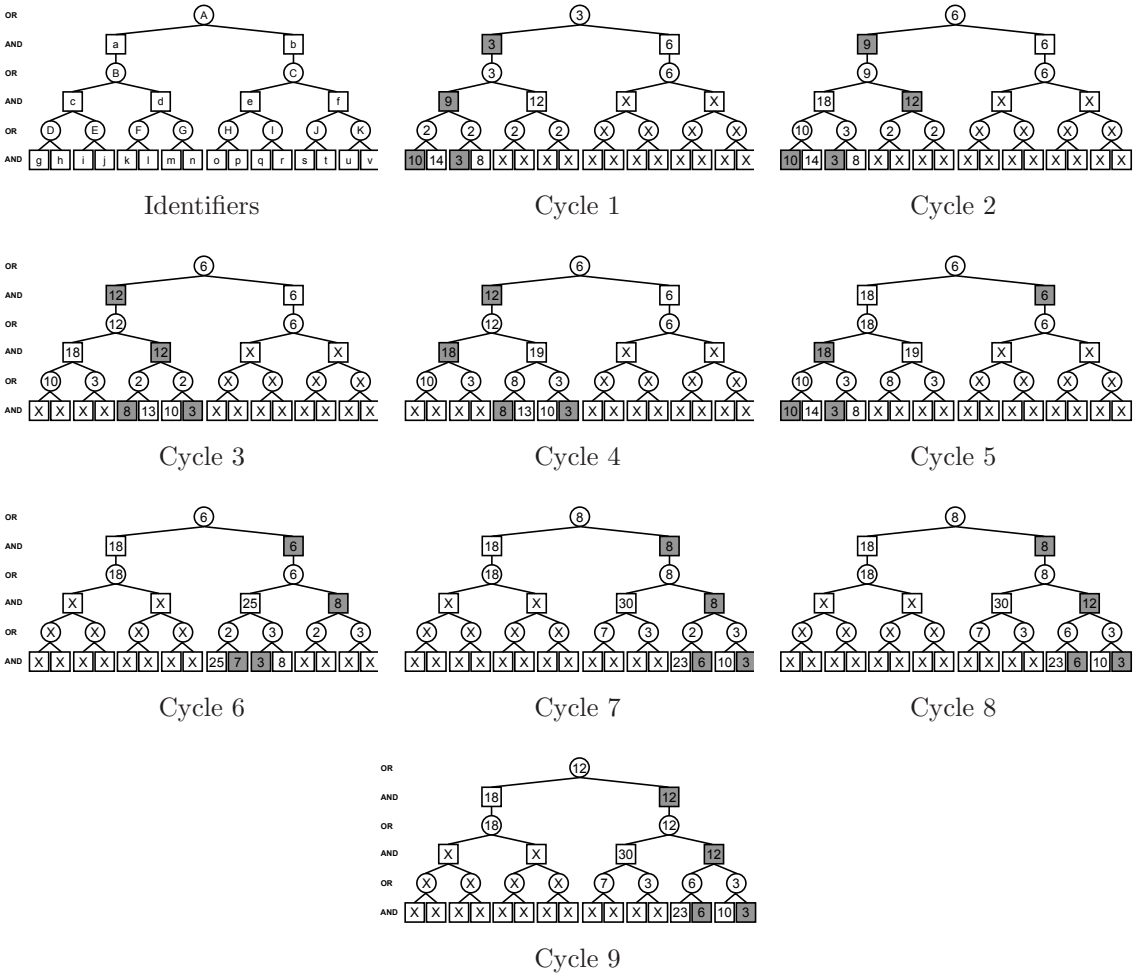


Figure 3.5: Trace of the Updates of the Lower Bounds of BnB-ADOPT

memory. Thus, BnB-ADOPT uses a different threshold initialization [Line 17], different desired threshold calculation [Line 30] and different termination condition [Line 27]. BnB-ADOPT also differs from ADOPT in that it maintains IDs that agents use to indicate the recency of their values and labels the values of agents in contexts with their IDs.

Cycle	1	2	3	4	5	6	7	8	9
X^{a_1}									
d^{a_1}	0	0	0	0	1	1	1	1	1
ID^{a_1}	1	1	1	1	2	2	2	2	2
TH^{a_1}	∞	∞	∞	∞	∞	∞	∞	∞	∞
$LB^{a_1}(0)$	3	9	12	12	18	18	18	18	18
$LB^{a_1}(1)$	6	6	6	6	6	6	8	8	12
LB^{a_1}	3	6	6	6	6	6	8	8	12
$UB^{a_1}(0)$	∞	∞	18	18	18	18	18	18	18
$UB^{a_1}(1)$	∞	∞	∞	∞	∞	∞	∞	30	12
UB^{a_1}	∞	∞	18	18	18	18	18	18	12
$lb^{a_1,a_2}(0)$	3	9	12	12	18	18	18	18	18
$lb^{a_1,a_2}(1)$	6	6	6	6	6	6	8	8	12
$ub^{a_1,a_2}(0)$	∞	∞	18	18	18	18	18	18	18
$ub^{a_1,a_2}(1)$	∞	∞	∞	∞	∞	∞	∞	30	12
X^{a_2}	$(a_1,0,0)$	$(a_1,0,1)$	$(a_1,0,1)$	$(a_1,0,1)$	$(a_1,0,1)$	$(a_1,1,2)$	$(a_1,1,2)$	$(a_1,1,2)$	$(a_1,1,2)$
d^{a_2}	0	1	1	0	0	1	1	1	1
ID^{a_2}	1	2	2	3	3	4	4	4	4
TH^{a_2}	∞	∞	∞	18	18	18	18	18	18
$LB^{a_2}(0)$	9	18	18	18	18	25	30	30	30
$LB^{a_2}(1)$	12	12	12	19	19	8	8	12	12
LB^{a_2}	9	12	12	18	18	8	8	12	12
$UB^{a_2}(0)$	∞	18	18	18	18	∞	30	30	30
$UB^{a_2}(1)$	∞	∞	∞	19	19	∞	∞	12	12
UB^{a_2}	∞	18	18	18	18	∞	30	12	12
$lb^{a_2,a_3}(0)$	2	10	10	10	10	2	7	7	7
$lb^{a_2,a_3}(1)$	2	2	2	8	8	2	2	6	6
$ub^{a_2,a_3}(0)$	∞	10	10	10	10	∞	7	7	7
$ub^{a_2,a_3}(1)$	∞	∞	∞	8	8	∞	∞	6	6
$lb^{a_2,a_4}(0)$	2	3	3	3	3	3	3	3	3
$lb^{a_2,a_4}(1)$	2	2	2	3	3	3	3	3	3
$ub^{a_2,a_4}(0)$	∞	3	3	3	3	3	3	3	3
$ub^{a_2,a_4}(1)$	∞	∞	∞	3	3	3	3	3	3
X^{a_3}	$(a_1,0,0)$	$(a_1,0,1)$	$(a_1,0,1)$	$(a_1,0,1)$	$(a_1,0,1)$	$(a_1,1,2)$	$(a_1,1,2)$	$(a_1,1,2)$	$(a_1,1,2)$
	$(a_2,0,0)$	$(a_2,0,1)$	$(a_2,1,2)$	$(a_2,1,2)$	$(a_2,0,3)$	$(a_2,0,3)$	$(a_2,1,4)$	$(a_2,1,4)$	$(a_2,1,4)$
d^{a_3}	0	0	0	0	0	1	1	1	1
ID^{a_3}	1	1	2	2	3	4	5	5	5
TH^{a_3}	∞	∞	8	8	10	10	12	12	6
$LB^{a_3}(0)$	10	10	8	8	10	25	23	23	23
$LB^{a_3}(1)$	14	14	13	13	14	7	6	6	6
LB^{a_3}	10	10	8	8	10	7	6	6	6
$UB^{a_3}(0)$	10	10	8	8	10	25	23	23	23
$UB^{a_3}(1)$	14	14	13	13	14	7	6	6	6
UB^{a_3}	10	10	8	8	10	7	6	6	6
X^{a_4}	$(a_2,0,0)$	$(a_2,0,1)$	$(a_2,1,2)$	$(a_2,1,2)$	$(a_2,0,3)$	$(a_2,0,3)$	$(a_2,1,4)$	$(a_2,1,4)$	$(a_2,1,4)$
d^{a_4}	0	0	1	1	0	0	1	1	1
ID^{a_4}	1	1	2	2	3	3	4	4	4
TH^{a_4}	∞	∞	8	8	3	3	13	13	3
$LB^{a_4}(0)$	3	3	10	10	3	3	10	10	10
$LB^{a_4}(1)$	8	8	3	3	8	8	3	3	3
LB^{a_4}	3	3	3	3	3	3	3	3	3
$UB^{a_4}(0)$	3	3	10	10	3	3	10	10	10
$UB^{a_4}(1)$	8	8	3	3	8	8	3	3	3
UB^{a_4}	3	3	3	3	3	3	3	3	3

Table 3.1: Trace of the Updates of all Variables of BnB-ADOPT

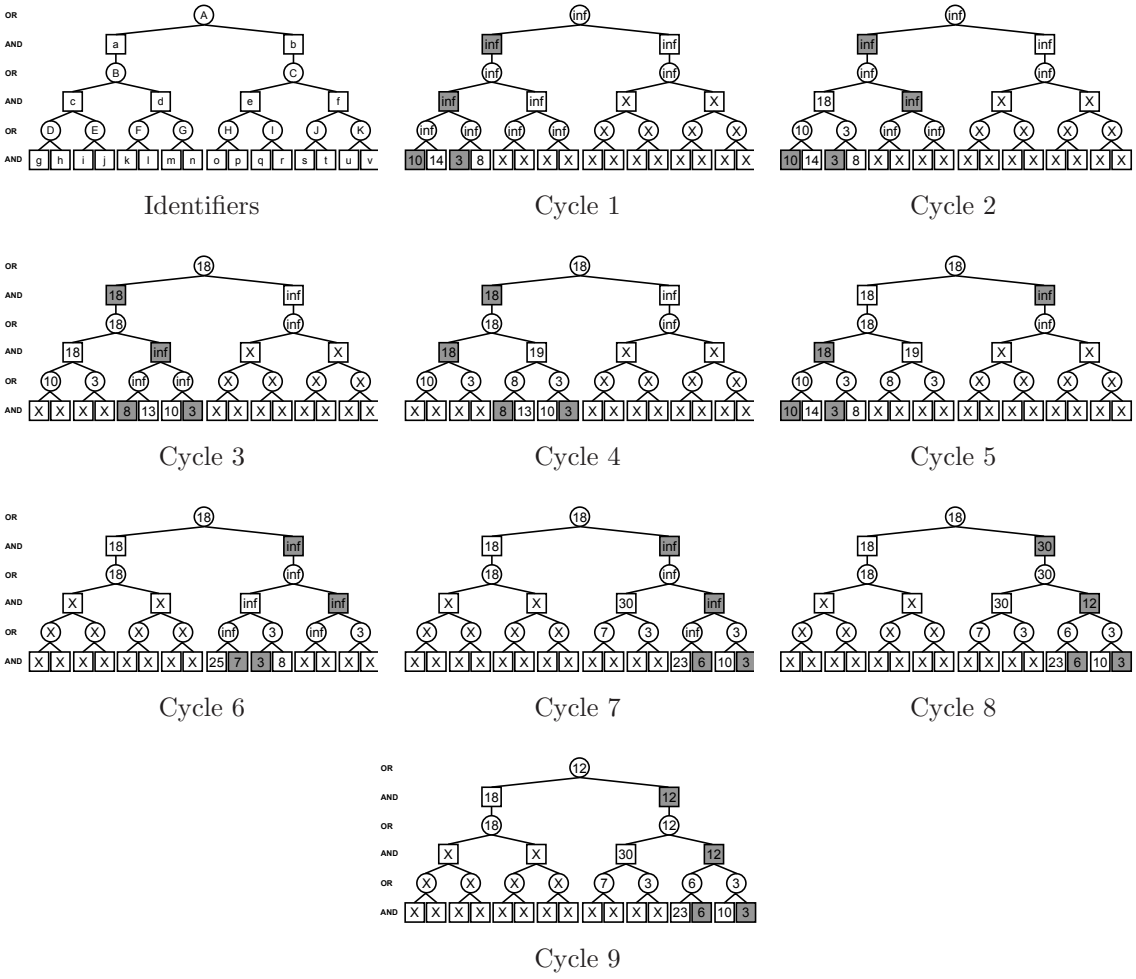


Figure 3.6: Trace of the Updates of the Upper Bounds of BnB-ADOPT

3.2.8 Execution Trace

Figures 3.5 and 3.6 show traces of the updates of the lower and upper bounds, respectively, for our example DCOP problem, and Table 3.1 shows a trace of the updates of all variables. BnB-ADOPT uses the heuristic values $h^{a_1,a_2}_{X^{a_1}}(0) := 3$, $h^{a_1,a_2}_{X^{a_1}}(1) := 6$, $h^{a_2,a_3}_{X^{a_2}}(0) := 2$, $h^{a_2,a_3}_{X^{a_2}}(1) := 2$, $h^{a_2,a_4}_{X^{a_2}}(0) := 2$ and $h^{a_2,a_4}_{X^{a_2}}(1) := 2$ for all contexts X^{a_1} and X^{a_2}. These heuristic values were chosen by hand. Every agent assigns the value of all its

ancestor agents in its initial context to 0. We partition time into cycles as in Figures 3.2 and 3.3 and continue to use the conventions made in the context of those figures.

- **Cycle 1:** Root agent a_1 initializes its context X^{a_1} to $\{\}$ [Line 1]. It initializes the lower bounds of nodes B $(= lb^{a_1,a_2}_{X^{a_1}}(0))$ and C $(= lb^{a_1,a_2}_{X^{a_1}}(1))$ to their heuristic values 3 and 6, respectively [Line 13]. It updates the lower bound of node a $(= LB^{a_1}_{X^{a_1}}(0))$ to the sum of its delta cost $(= 0)$ and the lower bound of node B $(= 3)$ according to the update equations [Line 19]. It updates the lower bound of node b $(= LB^{a_1}_{X^{a_1}}(1))$ to the sum of its delta cost $(= 0)$ and the lower bound of node C $(= 6)$ according to the update equations [Line 19]. It updates the lower bound of node A $(= LB^{a_1}_{X^{a_1}})$ to the minimum of the lower bound of node a $(= 3)$ and the lower bound of node b $(= 6)$ according to the update equations [Line 21]. It initializes the upper bounds of nodes B and C to infinity [Line 14]. It updates the upper bounds of nodes a, b and A to infinity according to the update equations [Lines 20 and 22]. It takes on its best value 0 since the lower bound of node a $(= 3)$ is smaller than the lower bound of node b $(= 6)$ [Line 15], initializes its ID ID^{a_1} to 1 [Lines 2 and 16], initializes its threshold TH^{a_1} to infinity [Line 17] and sends VALUE messages to its child agent a_2 and pseudo-child agent a_3 [Lines 30-31].

  Agent a_2 initializes its context X^{a_2} to $\{(a_1, 0, 0)\}$ [Line 1]. It initializes the lower bounds of nodes D, E, F and G to their heuristic value 2 [Line 13]. It updates the lower bounds of nodes c, d and B to 9, 12 and 9, respectively [Lines 19 and 21]. It initializes the upper bounds of nodes D, E, F and G to infinity [Line 14]. It updates the upper bounds of nodes c, d and B to infinity [Lines 20 and 22]. The

bounds of node B that agent a_2 maintains are not shown in the figure. It takes on its best value 0 [Line 15], initializes its ID to 1 [Lines 2 and 16], initializes its threshold to infinity [Line 17] and sends VALUE messages to its child agents a_3 and a_4 and a COST message to its parent agent a_1 [Lines 30-32].

Leaf agent a_3 initializes its context X^{a_3} to $\{(a_1, 0, 0), (a_2, 0, 0)\}$ [Line 1]. It updates the lower bounds of nodes g and h to their delta costs 10 and 14, respectively, since leaf agents do not have child agents [Line 19]. It updates the lower bound of node D to 10 [Line 21]. It updates the upper bounds of nodes g and h to their delta costs 10 and 14, respectively, since leaf agents do not have child agents [Line 20]. It updates the upper bound of node D to 10 [Line 22]. The bounds of node D that leaf agent a_3 maintains are not shown in the figure. It takes on its best value 0 [Line 15], initializes its ID to 1 [Lines 2 and 16], initializes its threshold to infinity [Line 17] and sends a COST message to its parent agent a_2 [Line 32].

Leaf agent a_4 initializes its (reduced) context X^{a_4} to $\{(a_2, 0, 0)\}$ [Line 1]. It updates the lower bounds of nodes i and j to their delta costs 3 and 8, respectively [Line 19]. It updates the lower bound of node E to 3 [Line 21]. It updates the upper bounds of nodes i and j to their delta costs 3 and 8, respectively [Line 20]. It updates the upper bound of node E to 3 [Line 22]. The bounds of node E that leaf agent a_4 maintains are not shown in the figure. It takes on its best value 0 [Line 15], initializes its ID to 1 [Lines 2 and 16], initializes its threshold to infinity [Line 17] and sends a COST message to its parent agent a_2 [Line 32].

In summary, the following messages are sent during Cycle 1:

- message (VALUE, a_1, 0, 1, ∞) from agent a_1 to agent a_2;
- message (VALUE, a_1, 0, 1, ∞) from agent a_1 to agent a_3;
- message (VALUE, a_2, 0, 1, ∞) from agent a_2 to agent a_3;
- message (VALUE, a_2, 0, 1, ∞) from agent a_2 to agent a_4;
- message (COST, a_2, $\{(a_1, 0, 0)\}$, 9, ∞) from agent a_2 to agent a_1;
- message (COST, a_3, $\{(a_1, 0, 0), (a_2, 0, 0)\}$, 10, 10) from agent a_3 to agent a_2; and
- message (COST, a_4, $\{(a_2, 0, 0)\}$, 3, 3) from agent a_4 to agent a_2.

- **Cycle 2:** Root agent a_1 receives the COST message sent by its child agent a_2 in Cycle 1. Since the context of agent a_1 (= {}) is compatible with the context in the message (= $\{(a_1, 0, 0)\}$), it improves its bounds. It updates the bounds of node B to the bounds in the message (= 9 and infinity, respectively) [Lines 48-50]. It updates the bounds of nodes a, b and A [Lines 18-22]. It does not change its value since the lower bound of node a (= $LB^{a_1}_{X^{a_1}}(d^{a_1}) = 9$ for its value $d^{a_1} = 0$) is still smaller than its pruning quantity (= $\min\{TH^{a_1}_{X^{a_1}}, UB^{a_1}_{X^{a_1}}\} = \min(\infty, \infty) = \infty$). It sends VALUE messages to its child agent a_2 and pseudo-child agent a_3 [Lines 30-31].

  Agent a_2 receives the VALUE message sent by its parent agent a_1 in Cycle 1. It updates its context from $\{(a_1, 0, 0)\}$ to $\{(a_1, 0, 1)\}$ since the ID of agent a_1 in its context (= 0) is smaller than the ID in the message (= 1) [Line 34]. Its threshold (= ∞) remains unchanged since it is the same as the desired threshold (= ∞) in

the message. Agent a_2 also receives the COST messages sent by its child agents a_3 and a_4 in Cycle 1. Since its context (= $\{(a_1, 0, 1)\}$) is compatible with the contexts in the messages (= $\{(a_1, 0, 0), (a_2, 0, 0)\}$ and $\{(a_2, 0, 0)\}$, respectively), it improves its bounds. It updates the bounds of node D to the bounds in the first message (= 10 and 10, respectively) and the bounds of node E to the bounds in the second message (= 3 and 3, respectively) [Lines 48-50]. It updates the bounds of nodes c, d and B [Lines 18-22]. It changes its value since the lower bound of node c ($= LB^{a_2}_{X^{a_2}}(d^{a_2}) = 18$ for its value $d^{a_2} = 0$) is no longer smaller than its pruning quantity ($= \min\{TH^{a_2}_{X^{a_2}}, UB^{a_2}_{X^{a_2}}\} = \min(\infty, 18) = 18$). It takes on its best value 1 [Line 24], increments its ID to 2 [Lines 25-26], sends VALUE messages to its child agents a_3 and a_4 [Lines 30-31] and sends a COST message to its parent agent a_1 [Line 32].

Leaf agent a_3 receives the VALUE messages sent by its parent agent a_2 and pseudo-parent agent a_1 in Cycle 1. It updates its context from $\{(a_1, 0, 0), (a_2, 0, 0)\}$ to $\{(a_1, 0, 1), (a_2, 0, 1)\}$ since the IDs of agents a_1 and a_2 in its context (= 0 and 0, respectively) are smaller than the IDs in the messages (= 1 and 1, respectively) [Line 34]. Its threshold ($= \infty$) remains unchanged since it is the same as the desired threshold ($= \infty$) in the message. Its bounds are not reinitialized since its context is compatible with its previous context [Line 35]. It sends the same COST message as before to its parent agent a_2 [Line 32].

Leaf agent a_4 receives the VALUE message sent by its parent agent a_2 in Cycle 1. It updates its contexts from $\{(a_2, 0, 0)\}$ to $\{(a_2, 0, 1)\}$ since the ID of agent a_2 in its

context (= 0) is smaller than the ID in the message (= 1) [Line 34]. Its threshold (= ∞) remains unchanged since it is the same as the desired threshold (= ∞) in the message. Its bounds are not reinitialized since its context is compatible with its previous context [Line 35]. It sends the same COST message as before to its parent agent a_2 [Line 32].

In summary, the messages sent during Cycle 2 are identical to the ones sent during Cycle 1, except for the messages sent by agents a_2, a_3 and a_4, which are as follows:

- message (VALUE, a_2, 1, 2, 8) from agent a_2 to agent a_3;
- message (VALUE, a_2, 1, 2, 8) from agent a_2 to agent a_4; and
- message (COST, a_2, $\{(a_1, 0, 1)\}$, 12, 18) from agent a_2 to agent a_1.
- message (COST, a_3, $\{(a_1, 0, 1), (a_2, 0, 1)\}$, 10, 10) from agent a_3 to agent a_2; and
- message (COST, a_4, $\{(a_2, 0, 1)\}$, 3, 3) from agent a_4 to agent a_2.

The VALUE messages are different because agent a_2 changed its value from 0 to 1. The COST messages are different because agent a_2 changed its bounds and its context and agents a_3 and a_4 changed their contexts.

- **Cycles 3-9:** The messages sent during Cycle 3 are identical to the ones sent during Cycle 2, except for the messages sent by agents a_3 and a_4, which are as follows:
  - message (COST, a_3, $\{(a_1, 0, 1), (a_2, 1, 2)\}$, 8, 8) from agent a_3 to agent a_2; and
  - message (COST, a_4, $\{(a_2, 1, 2)\}$, 3, 3) from agent a_4 to agent a_2.

The COST messages are different because agents a_3 and a_4 changed their contexts. The termination conditions holds after a finite amount of time when the upper bound of node A ($= UB^{a_1}_{X^{a_1}} = 12$) is no larger than the lower bound of node A ($= LB^{a_1}_{X^{a_1}} = 12$) [Line 27]. Root agent a_1 sends TERMINATE messages to all child agents [Line 28], and the TERMINATE messages propagate down the pseudo-tree [Line 28] until all agents terminate. BnB-ADOPT terminates after nine cycles with minimal solution cost 12.

3.3 Correctness, Completeness and Complexity

In this section, we prove the correctness and completeness of BnB-ADOPT. Additionally, we also describe its space and message complexities.

3.3.1 Correctness and Completeness

Each agent a uses the following equations for all values d, all child agents c and all contexts X^a to initialize its bounds.

$$lb^{a,c}_{X^a}(d) := w \cdot h^{a,c}_{X^a}(d) \tag{3.20}$$

$$ub^{a,c}_{X^a}(d) := \infty \tag{3.21}$$

where the weight w is a floating point number that satisfies $1 \leq w < \infty$ and the heuristic values $h^{a,c}_{X^a}(d)$ are floating point numbers that satisfy

$$0 \leq h^{a,c}_{X^a}(d) \leq \gamma^c_{X^a \cup (a,d)} \tag{3.22}$$

We only need to prove for $w = 1$ in order to prove the correctness and completeness of BnB-ADOPT. However, we prove for larger weights so that they also apply to BnB-ADOPT when it uses the approximation mechanisms described in Chapter 4. Messages are sent at the end of a cycle and received in the beginning of a cycle. Δ is the largest duration between the time a message is sent and the time it is processed, and ϵ is the largest duration of a cycle.

Lemma 1 *If two contexts X and X' of an arbitrary agent $a \in A$ agree on the values of all ancestor agents $p \in SCP(a)$ of agent a, then $\gamma^a_X = \gamma^a_{X'}$.*

Proof: By definition, $X^a \subseteq X$ is the (reduced) context that contains the values of all ancestor agents $p \in SCP(a)$ of agent a. The gamma cost γ^a_X is the sum of the constraint costs of all constraints that involve agent a or one of its descendant agents minimized over all possible values of agent a and its descendant agents, under the assumption that the ancestor agents of agent a take on the values in context X. The gamma cost γ^a_X thus depends only on the values of the ancestor agents (including the parent agent) of agent a that are the parent or pseudo-parent agents of agent a or one (or more) of its descendant agents, that is, the values of all ancestor agents $p \in SCP(a)$ of agent a. Therefore, $\gamma^a_X = \gamma^a_{X^a}$. Similarly, $\gamma^a_{X'} = \gamma^a_{X^a}$. ■

Definition 1 *Contexts are correct iff the IDs of the values of all agents in the contexts are equal to the IDs of the agents, which implies that the values of all agents in the contexts are equal to the values of the agents.*

Lemma 2 *If the context X^a of an arbitrary agent $a \in A$ does not change for a period of time, then the lower bounds $lb^{a,c}_{X^a}(d)$, $LB^a_{X^a}(d)$ and $LB^a_{X^a}$ are monotonically non-decreasing and the upper bounds $ub^{a,c}_{X^a}(d)$, $UB^a_{X^a}(d)$ and $UB^a_{X^a}$ are monotonically non-increasing during that period of time for all values $d \in Dom(a)$ and all child agents $c \in C(a)$.*

Proof: Since the context X^a does not change, the delta values $\delta^a_{X^a}(d)$ are constant and the bounds (once initialized) are updated according to update equations 3.8 to 3.13. Thus, the lower bounds are monotonically non-decreasing and the upper bounds are monotonically non-increasing. ■

Lemma 3 *If the value of an arbitrary ancestor agent $p \in SCP(a)$ of an arbitrary agent $a \in A$ does not change between the current time T and a future time t with $t \geq T + |A| \cdot (\Delta + \epsilon) + \epsilon$, then the value of agent p and its ID in the context of agent a are equal to the value of agent p and its ID, respectively, between some time t' and time t with $t' \leq t$.*

Proof: Assume that the value of an arbitrary ancestor agent $p \in SCP(a)$ of an arbitrary agent $a \in A$ does not change between the current time T and a future time t with $t \geq T + |A| \cdot (\Delta + \epsilon) + \epsilon$. There are the following two cases.

- Case 1: If agent p is a parent or pseudo-parent agent of agent a, then it sent a VALUE message to agent a with its value and ID at the time $t'' \leq T + \epsilon$ that is at

the end of the cycle in which it took on the value that it has at time T since the duration of that cycle is no larger than ϵ. (The agents send VALUE messages at the end of every cycle.) Agent a receives the VALUE message by time $t'' + \Delta$ since messages are delivered with finite delay Δ. It then updates the value of agent p and its ID in its context by time $t'' + \Delta + \epsilon$ since the update is done in the same cycle and thus takes an amount of time no larger than ϵ. Thus, the value of agent p and its ID in the context of agent a are equal to the value of agent p and its ID, respectively, between some time t' and time t with $t'' \leq t' \leq t'' + \Delta + \epsilon \leq T + \Delta + 2 \cdot \epsilon \leq t$ since agent p does not change its value between time t'' and time t.

- Case 2: If agent p is not a parent or pseudo-parent agent of agent a, then one of its pseudo-child agent c is a descendant agent of agent a. Agent p sent a VALUE message to agent c with its value and ID at the time $t'' \leq T + \epsilon$ that is at the end of the cycle in which it took on the value that it has at time T. Agent c receives the VALUE message by time $t'' + \Delta$. It then updates the value of agent p and its ID in its context and sends a COST message to its parent agent $pa(c)$ with its updated context by time $t'' + \Delta + \epsilon$. (The agents send COST messages at the end of every cycle.) Agent $pa(c)$ receives the COST message by time $t'' + 2 \cdot \Delta + \epsilon$. It then updates the value of agent p and its ID in its context and sends a COST message to its parent agent $pa(pa(c))$ with its updated context by time $t'' + 2 \cdot (\Delta + \epsilon)$. This process continues until agent a updates the value of agent p and its ID in its context by time $t'' + n \cdot (\Delta + \epsilon)$, where $n \leq |A|$ is the number of messages in the chain of messages. Thus, the value of agent p and its ID in the context of agent a are equal

to the value of agent p and its ID, respectively, between some time t' and time t with $t'' \leq t' \leq t'' + n \cdot (\Delta + \epsilon) \leq T + |A| \cdot (\Delta + \epsilon) + \epsilon \leq t$ since agent p does not change its value between time t'' and time t. ■

Corollary 1 *If the values of all ancestor agents $p \in SCP(a)$ of an arbitrary agent $a \in A$ do not change between the current time T and a future time t with $t \geq T + |A| \cdot (\Delta + \epsilon) + \epsilon$, then the context of agent a is correct between some time t' and time t with $t' \leq t$.*

Lemma 4 *If $LB^c_{X^c} \leq w \cdot \gamma^c_{X^c} \leq w \cdot UB^c_{X^c}$ at all times for all child agents $c \in C(a)$ of an arbitrary agent a and their contexts X^c, then $lb^{a,c}_{X^a}(d) \leq w \cdot \gamma^c_{X^a \cup (a,d)} \leq w \cdot ub^{a,c}_{X^a}(d)$ at all times for the context X^a of agent a, all values $d \in Dom(a)$ and all child agents $c \in C(a)$.*

Proof by induction on the number of times that agent a changes its context or updates its bounds $lb^{a,c}_{X^a}(d)$ and $ub^{a,c}_{X^a}(d)$ for an arbitrary value d and an arbitrary child agent c after agent a initializes its bounds: The lemma holds after agent a with context X^a initializes its bounds since

$$\begin{aligned} lb^{a,c}_{X^a}(d) &= w \cdot h^{a,c}_{X^a}(d) && \text{(Eq. 3.20)} \\ &\leq w \cdot \gamma^c_{X^a \cup (a,d)} && \text{(Eq. 3.22)} \\ &\leq \infty \\ &= w \cdot ub^{a,c}_{X^a}(d) && \text{(Eq. 3.7)} \end{aligned}$$

for the (unchanged or new) context X^a of agent a (induction basis). Now assume that the lemma holds after agent a changed its context or updated its bounds a number of

times (induction assumption). We show that it then also holds after agent a changes its context or updates its bounds one more time (induction step). There are the following two cases (where we split the operations after receiving a COST message into two parts).

- Case 1: The lemma holds when agent a changes its context from X^a to $\hat{X}^a$ after receiving a VALUE or COST message and the two contexts are compatible since agent a then does not change its bounds and thus

$$\begin{aligned}
lb^{a,c}_{\hat{X}^a}(d) &= lb^{a,c}_{X^a}(d) && \text{(premise of case)} \\
&\leq w \cdot \gamma^c_{X^a \cup (a,d)} && \text{(induction assumption)} \\
&= w \cdot \gamma^c_{\hat{X}^a \cup (a,d)} && \text{(Lemma 1)} \\
ub^{a,c}_{\hat{X}^a}(d) &= ub^{a,c}_{X^a}(d) && \text{(premise of case)} \\
&\geq \gamma^c_{X^a \cup (a,d)} && \text{(induction assumption)} \\
&= \gamma^c_{\hat{X}^a \cup (a,d)} && \text{(Lemma 1)}
\end{aligned}$$

  after receiving the VALUE or COST message.

- Case 2: The lemma holds when agent a updates its bounds from $lb^{a,c}_{X^a}(d)$ and $ub^{a,c}_{X^a}(d)$ to $\hat{lb}^{a,c}_{X^a}(d)$ and $\hat{ub}^{a,c}_{X^a}(d)$, respectively, after receiving a COST message from some child agent c with bounds $LB^c_{X^c}$ and $UB^c_{X^c}$ and context X^c that is compatible with its context X^a and in which agent a has value d since

$$
\begin{aligned}
\hat{lb}^{a,c}_{X^a}(d) &= \max\{lb^{a,c}_{X^a}(d), LB^c_{X^c}\} && \text{(Eq. 3.8)}\\
&\leq \max\{w \cdot \gamma^c_{X^a \cup (a,d)}, w \cdot \gamma^c_{X^c}\} \\
& && \text{(induction assumption and premise of lemma)}\\
&= \max\{w \cdot \gamma^c_{X^a \cup (a,d)}, w \cdot \gamma^c_{X^a \cup (a,d)}\} && \text{(Lemma 1)}\\
&= w \cdot \gamma^c_{X^a \cup (a,d)} \\
\hat{ub}^{a,c}_{X^a}(d) &= \min\{ub^{a,c}_{X^a}(d), UB^c_{X^c}\} && \text{(Eq. 3.11)}\\
&\geq \min\{\gamma^c_{X^a \cup (a,d)}, \gamma^c_{X^c}\} \quad \text{(induction assumption and premise of lemma)}\\
&= \min\{\gamma^c_{X^a \cup (a,d)}, \gamma^c_{X^a \cup (a,d)}\} && \text{(Lemma 1)}\\
&= \gamma^c_{X^a \cup (a,d)}
\end{aligned}
$$

after receiving the COST message.

Thus, $lb^{a,c}_{X^a}(d) \leq w \cdot \gamma^c_{X^a \cup (a,d)} \leq w \cdot ub^{a,c}_{X^a}(d)$ at all times for all values $d \in Dom(a)$ and all child agents $c \in C(a)$. ■

Lemma 5 *$LB^a_{X^a}(d) \leq w \cdot \gamma^a_{X^a}(d) \leq w \cdot UB^a_{X^a}(d)$ and $LB^a_{X^a} \leq w \cdot \gamma^a_{X^a} \leq w \cdot UB^a_{X^a}$ at all times for all values $d \in Dom(a)$ and the context X^a of an arbitrary agent $a \in A$.*

Proof by induction on the depth of an agent in the pseudo-tree: The lemma holds for a leaf agent a in the pseudo-tree with context X^a since

$$LB^a_{X^a}(d) = \delta^a_{X^a}(d) \quad \text{(Eq. 3.9)}$$

$$= \gamma^a_{X^a}(d) \quad \text{(Eq. 3.1)}$$

$$UB^a_{X^a}(d) = \delta^a_{X^a}(d) \quad \text{(Eq. 3.12)}$$

$$= \gamma^a_{X^a}(d) \quad \text{(Eq. 3.1)}$$

for all values d at all times. Thus, $LB^a_{X^a}(d) = \gamma^a_{X^a}(d) \leq w \cdot \gamma^a_{X^a}(d) = w \cdot UB^a_{X^a}(d)$ for all values d at all times. Furthermore,

$$LB^a_{X^a} = \min_{d \in Dom(a)} \{LB^a_{X^a}(d)\} \quad \text{(Eq. 3.10)}$$

$$= \min_{d \in Dom(a)} \{\gamma^a_{X^a}(d)\} \quad \text{(see above)}$$

$$= \gamma^a_{X^a} \quad \text{(Eq. 3.2)}$$

$$UB^a_{X^a} = \min_{d \in Dom(a)} \{UB^a_{X^a}(d)\} \quad \text{(Eq. 3.13)}$$

$$= \min_{d \in Dom(a)} \{\gamma^a_{X^a}(d)\} \quad \text{(see above)}$$

$$= \gamma^a_{X^a} \quad \text{(Eq. 3.2)}$$

at all times. Thus, $LB^a_{X^a} = \gamma^a_{X^a} \leq w \cdot \gamma^a_{X^a} = w \cdot UB^a_{X^a}$ at all times (induction basis). Now assume that the lemma holds for all agents at depth d in the pseudo-tree (induction

assumption). We show that it then also holds for all agents at depth $d-1$ in the pseudo-tree each time after they update their bounds (induction step). The lemma holds for agent a with context X^a since

$$
\begin{aligned}
LB^a_{X^a}(d) &= \delta^a_{X^a}(d) + \sum_{c \in C(a)} lb^{a,c}_{X^a}(d) && \text{(Eq. 3.9)} \\
&\leq \delta^a_{X^a}(d) + \sum_{c \in C(a)} w \cdot \gamma^c_{X^a \cup (a,d)} && \text{(induction assumption and Lemma 4)} \\
&\leq w \cdot \gamma^a_{X^a}(d) && \text{(Eq. 3.1)} \\
UB^a_{X^a}(d) &= \delta^a_{X^a}(d) + \sum_{c \in C(a)} ub^{a,c}_{X^a}(d) && \text{(Eq. 3.12)} \\
&\geq \delta^a_{X^a}(d) + \sum_{c \in C(a)} \gamma^c_{X^a \cup (a,d)} && \text{(induction assumption and Lemma 4)} \\
&= \gamma^a_{X^a}(d) && \text{(Eq. 3.1)}
\end{aligned}
$$

Thus, $LB^a_{X^a}(d) \leq w \cdot \gamma^a_{X^a}(d) \leq w \cdot UB^a_{X^a}(d)$ at all times for all values $d \in Dom(a)$. Furthermore,

$$
\begin{aligned}
LB^a_{X^a} &= \min_{d \in Dom(a)} \{LB^a_{X^a}(d)\} && \text{(Eq. 3.10)} \\
&\leq \min_{d \in Dom(a)} \{w \cdot \gamma^a_{X^a}(d)\} && \text{(see above)} \\
&= w \cdot \min_{d \in Dom(a)} \{\gamma^a_{X^a}(d)\} \\
&= w \cdot \gamma^a_{X^a} && \text{(Eq. 3.2)}
\end{aligned}
$$

$$
\begin{aligned}
UB^a_{X^a} &= \min_{d \in Dom(a)} \{UB^a_{X^a}(d)\} && \text{(Eq. 3.13)} \\
&\geq \min_{d \in Dom(a)} \{\gamma^a_{X^a}(d)\} && \text{(see above)} \\
&= \gamma^a_{X^a} && \text{(Eq. 3.2)}
\end{aligned}
$$

Thus, $LB^a_{X^a} \leq w \cdot \gamma^a_{X^a} \leq w \cdot UB^a_{X^a}$ at all times. ■

Definition 2 *The potential of an agent $a \in A$ with context X^a is $\sum_{d \in Dom(a)} \{w \cdot UB^a_{X^a}(d) - LB^a_{X^a}(d)\}$.*

Lemma 6 *If the context X^a of an arbitrary agent $a \in A$ no longer changes, then the potential of the agent is monotonically non-increasing and decreases by more than a positive constant every time the agent changes its value.*

Proof: The lower bounds $LB^a_{X^a}(d)$ are monotonically non-decreasing and the upper bounds $UB^a_{X^a}(d)$ are monotonically non-increasing for all values d according to Lemma 2 since the context X^a of agent a no longer changes. Therefore, the potential of agent a is monotonically non-increasing. Furthermore, agent a changes its value d to a new value only if $\min_{d \in Dom(a)} \{LB^a_{X^a}(d)\} < LB^a_{X^a}(d)$ [Line 24]. Thus, the lower bound $LB^a_{X^a}(d)$ must have strictly increased between the time when the agent changed its value to d and the time when it changes its value d to a new value. Thus, its potential has decreased by more than a positive constant, namely the smallest possible increase of the lower bound $LB^a_{X^a}(d)$. Assume that all constraint costs, weights and heuristic values are integers. Then, the smallest possible increase is bounded from below by one because the only possible values of $LB^a_{X^a}(d)$ are combinations of all constraint costs and weighted heuristic

values. A similar statement holds if all constraint costs, weights and heuristic values are floating point numbers since they can then all be transformed into integers by multiplying them with the same sufficiently large integer. ■

Lemma 7 *All agents change their values only a finite number of times.*

Proof by contradiction: Assume that the lemma does not hold and choose an agent a that changes its value an infinite number of times but whose ancestor agents $p \in SCP(a)$ change their values only a finite number of times. Then, there exists a time when the ancestor agents do not change their values any longer. There exists a (later) time when agent a no longer changes its context X^a according to Corollary 1. Every time agent a changes its value afterwards, its potential decreases by more than a positive constant according to Lemma 6, towards minus infinity. However, its potential cannot become negative since $LB^a_{X^a}(d) \leq w \cdot UB^a_{X^a}(d)$ for all values d according to Lemma 5, which is a contradiction. Thus, all agents change their values only a finite number of times. ■

Lemma 8 *If BnB-ADOPT does not terminate earlier, then $UB^a_{X^a} \leq LB^a_{X^a}$ after a finite amount of time for all agents $a \in A$ and their contexts X^a.*

Proof by induction on the depth of an agent in the pseudo-tree: There exists a time when no agent changes its value any longer according to Lemma 7. There exists a (later) time when the contexts of all agents are correct and no longer change according to Corollary 1. Let X^a be the context of agent a at this point in time for all agents a. There exists an (even later) time when the bounds $lb^{a,c}_{X^a}(d)$, $LB^a_{X^a}(d)$, $LB^a_{X^a}$, $ub^{a,c}_{X^a}(d)$, $UB^a_{X^a}(d)$ and $UB^a_{X^a}$ no longer change for all agents a, all values d and all child agents c since (1) the lower bounds

$lb^{a,c}_{X^a}(d)$, $LB^a_{X^a}(d)$ and $LB^a_{X^a}$ are monotonically non-decreasing and the upper bounds $lb^{a,c}_{X^a}(d)$, $UB^a_{X^a}(d)$ and $UB^a_{X^a}$ are monotonically non-increasing for all agents a, all values d and all child agents c according to Lemma 2, (2) $LB^a_{X^a}(d) \leq w \cdot \gamma^a_{X^a}(d) \leq w \cdot UB^a_{X^a}(d)$ and $LB^a_{X^a} \leq w \cdot \gamma^a_{X^a} \leq w \cdot UB^a_{X^a}$ for all agents a and all values d according to Lemma 5, (3) $lb^{a,c}_{X^a}(d) \leq w \cdot ub^{a,c}_{X^a}(d)$ for all agents a, all values d and all child agents c according to Lemma 4 and (4) the smallest possible increases of the lower bounds and the smallest possible decreases of the upper bounds are larger than a positive constant since the only possible values of the bounds are combinations of all constraint costs and heuristic values, as explained in more detail in the proof of Lemma 6. Consider the first COST message that each agent sends after this time and the earliest time when all of these COST messages have been processed by their receiving agents. The lemma holds for a leaf agent a in the pseudo-tree with context X^a since

$$\begin{aligned} LB^a_{X^a}(d) &= \delta^a_{X^a}(d) && \text{(Eq. 3.9)} \\ &= \gamma^a_{X^a}(d) && \text{(Eq. 3.1)} \\ UB^a_{X^a}(d) &= \delta^a_{X^a}(d) && \text{(Eq. 3.12)} \\ &= \gamma^a_{X^a}(d) && \text{(Eq. 3.1)} \end{aligned}$$

for all values d after the considered time. Furthermore,

$$\begin{aligned} LB^a_{X^a} &= \min_{d \in Dom(a)} \{LB^a_{X^a}(d)\} && \text{(Eq. 3.10)} \\ &= \min_{d \in Dom(a)} \{\gamma^a_{X^a}(d)\} && \text{(see above)} \\ &= \gamma^a_{X^a} && \text{(Eq. 3.2)} \end{aligned}$$

$$UB^a_{X^a} = \min_{d \in Dom(a)} \{UB^a_{X^a}(d)\} \quad \text{(Eq. 3.13)}$$

$$= \min_{d \in Dom(a)} \{\gamma^a_{X^a}(d)\} \quad \text{(see above)}$$

$$= \gamma^a_{X^a} \quad \text{(Eq. 3.2)}$$

after the considered time. Thus, $UB^a_{X^a} = LB^a_{X^a}$ after the considered time (induction basis). Now assume that the lemma holds for all agents at depth d in the pseudo-tree after the considered time (induction assumption). We show that it then also holds for all agents at depth $d-1$ in the pseudo-tree after the considered time (induction step). For agent a with context X^a

$$LB^a_{X^a}(d) = \delta^a_{X^a}(d) + \sum_{c \in C(a)} lb^{a,c}_{X^a}(d) \quad \text{(Eq. 3.9)}$$

$$= \delta^a_{X^a}(d) + \sum_{c \in C(a)} \max\{lb^{a,c}_{X^a}(d), LB^c_{X^c}\} \quad \text{(Eq. 3.8)}$$

$$\geq \delta^a_{X^a}(d) + \sum_{c \in C(a)} LB^c_{X^c}$$

$$\geq \delta^a_{X^a}(d) + \sum_{c \in C(a)} UB^c_{X^c} \quad \text{(induction assumption)}$$

$$\geq \delta^a_{X^a}(d) + \sum_{c \in C(a)} \min\{ub^{a,c}_{X^a}(d), UB^c_{X^c}\}$$

$$= \delta^a_{X^a}(d) + \sum_{c \in C(a)} ub^{a,c}_{X^a}(d) \quad \text{(Eq. 3.11)}$$

$$= UB^a_{X^a}(d) \quad \text{(Eq. 3.12)}$$

for its value d after the considered time since all bounds no longer change. Thus, $UB^a_{X^a}(d) \leq LB^a_{X^a}(d)$ for its value d after the considered time. Since agent a does not change its value d after the considered time, it must hold that $LB^a_{X^a}(d) < \min\{TH^a_{X^a}, UB^a_{X^a}\}$ [Line 23] or $LB^a_{X^a}(d) = \min_{d \in Dom(a)}\{LB^a_{X^a}(d)\}$ [Line 24]. The first disjunct implies that

$$\begin{aligned}
\min\{TH^a_{X^a}, UB^a_{X^a}\} &\leq UB^a_{X^a} \\
&\leq UB^a_{X^a}(d) && \text{(Eq. 3.13)} \\
&\leq LB^a_{X^a}(d) && \text{(see above)} \\
&< \min\{TH^a_{X^a}, UB^a_{X^a}\} && \text{(first disjunct)}
\end{aligned}$$

for its value d, which is a contradiction. The second disjunct implies that

$$\begin{aligned}
UB^a_{X^a} &\leq UB^a_{X^a}(d) && \text{(Eq. 3.13)} \\
&\leq LB^a_{X^a}(d) && \text{(see above)} \\
&= \min_{d \in Dom(a)}\{LB^a_{X^a}(d)\} && \text{(second disjunct)} \\
&= LB^a_{X^a} && \text{(Eq. 3.10)}
\end{aligned}$$

for its value d and thus that $UB^a_{X^a} \leq LB^a_{X^a}$. ■

Theorem 1 *BnB-ADOPT terminates after a finite amount of time.*

Proof: If BnB-ADOPT does not terminate earlier, then $UB^a_{X^a} \leq LB^a_{X^a}$ after a finite amount of time for all agents $a \in A$ and their contexts X^a according to Lemma 8. In particular, $UB^r_{X^r} \leq LB^r_{X^r}$ for the root agent r. Thus, the termination condition $UB^r_{X^r} \leq LB^r_{X^r}$ of BnB-ADOPT is satisfied. ■

Theorem 2 *BnB-ADOPT terminates with the minimal solution cost $\gamma^r_{X^r}$.*

Proof: BnB-ADOPT terminates after a finite amount of time according to Theorem 1. The solution cost of BnB-ADOPT is the upper bound $UB^r_{X^r}$ of the root agent r. $UB^r_{X^r} \leq LB^r_{X^r}$ upon termination according to its termination condition. $w \cdot UB^r_{X^r} \geq w \cdot \gamma^r_{X^r} \geq LB^r_{X^r}$ according to Lemma 5. Therefore, $UB^r_{X^r} = \gamma^r_{X^r} = LB^r_{X^r}$ since $w = 1$. ■

3.3.2 Complexity

We measure the space complexity of BnB-ADOPT in the number of floating point numbers. As BnB-ADOPT uses the same framework as ADOPT, their memory requirements are identical. Every agent a needs to store $lb^{a,c}(d)$ and $ub^{a,c}(d)$ for all child agents $c \in C(a)$ and values $d \in Dom(a)$. Thus, the space complexity of every agent a in BnB-ADOPT is $O(|C(a)| \cdot |Dom(a)|) = O(|A| \cdot maxDom)$, where $maxDom = max_{a' \in A}|Dom(a')|$ is the maximum domain cardinality over all agents $a' \in A$.

We measure the message complexity of BnB-ADOPT in the number of floating point numbers as well. As BnB-ADOPT uses the same messages as ADOPT, their message complexities are identical. The complexity of VALUE messages is $O(1)$ since they contain five floating point numbers. The complexity of COST messages is $O(|A|)$ since they

contain four floating point numbers and the context of the sending agent. Therefore, the message complexity of BnB-ADOPT is $O(|A|)$.

3.4 Experimental Evaluation

We now compare BnB-ADOPT to two other memory-bounded DCOP search algorithms that also restrict communication to agents that share constraints, namely ADOPT and NCBB. NCBB is a memory-bounded synchronous branch-and-bound DCOP search algorithm with the feature that an agent can take on a different value for each one of its child agents.

As described in Section 2.1.2.3, ADOPT can be sped up with the use of heuristic values. In our experiments, we use a state-of-the-art pre-processing framework called DP2 (Ali et al., 2005) to calculate heuristic values for ADOPT. The same heuristic values can also be used for BnB-ADOPT since the definition and use of lower bounds in BnB-ADOPT is the same as that in ADOPT. It calculates heuristic values by solving a relaxed version (where backedges are ignored) of the DCOP problem using a dynamic programming-based approach. Agents in NCBB do not perform any pre-processing, but calculate their own heuristic values during the search.

3.4.1 Metrics

We measure the runtimes and solution costs of ADOPT, BnB-ADOPT and NCBB. Additionally, we also measure the number of unique and repeated contexts explored per agent of ADOPT and BnB-ADOPT to better understand the reason for the speedups.

The number of unique contexts is the number of different contexts explored. The number of repeated contexts is the total number of contexts explored minus the number of unique contexts. The sum of both numbers is correlated with the runtime of ADOPT and BnB-ADOPT.

We measure the runtimes with two common runtime metrics, namely non-concurrent constraint checks (Meisels, Kaplansky, Razgon, & Zivan, 2002) and cycles (Hirayama & Yokoo, 2000).

- **Non-concurrent constraint checks (NCCCs):** NCCCs are a weighted sum of processing and communication time. Every agent a maintains a counter $NCCC^a$, which is initialized to 0. The agent assigns $NCCC^a := NCCC^a + 1$ every time it performs a constraint check to account for the time it takes to perform the constraint check. It assigns $NCCC^a := \max\{NCCC^a, NCCC^{a'} + t\}$ every time it receives a message from agent a' to account for the time it takes to wait for agent a' to send the message ($NCCC^{a'}$) and the transmission time of the message (t). We use $t = 0$ to simulate fast communication and $t = 1000$ to simulate slow communication. The number of NCCCs then is the largest counter value of any agent. NCCCs are a good runtime metric if the ratio of processing and communication time can be estimated reliably.

- **Cycles:** Cycles are time slices. A cycle is the time required for an agent to process all incoming messages in its queue and send all outgoing messages, which are then processed by the receiving agents in the next cycle. Thus, the number of cycles indicates the length of the longest chain of messages between agents. Cycles are a

good runtime metric if the communication time is much larger than the processing time. Cycles will become a better and better runtime metric in the future since the communication time is expected to remain relatively stable while the processing time is expected to decrease (Silaghi, Lass, Sultanik, Regli, Matsui, & Yokoo, 2008).

3.4.2 Problem Types

As described in Section 2.1.5, we run our experiments on four problem types, namely graph coloring, sensor network, meeting scheduling and combinatorial auction problems.

- **Graph coloring problems:** We vary the number of agents (= vertices to color) from 5 to 15, the density, defined as the ratio between the number of constraints and the number of agents, from 2 (sparse graphs) to 3 (dense graphs) and the range of constraint costs from a range of 0 to 1 (small range) to a range of 0 to 10,000 (large range). Each agent always has three possible values (= colors). All costs are randomly generated. We average the experimental results over 50 DCOP problem instances with randomly generated constraints.

- **Sensor network problems:** We vary the number of agents (= targets to track) from 4 to 15. Each agent always has nine values (= time slots). The cost of assigning a time slot to a target that is also assigned to an adjacent target is infinity (to be precise: 1,000,000) since the same sensor cannot track both targets during the same time slot. The cost of targets that are not tracked during any time slot is 100. All other costs are randomly generated from 0 to 100. We average the experimental results over 50 DCOP problem instances.

- **Meeting scheduling problems:** We vary the number of agents (= meetings to schedule) from 5 to 20. Each agent always has nine values (= time slots). The cost of assigning a time slot to a meeting that has at least one participant who has another meeting during the same time slot is infinity (to be precise: 1,000,000) since the same person cannot attend more than one meeting at a time. The cost of a non-scheduled meeting is 100. All other costs are randomly generated from 0 to 100. We average the experimental results over 50 DCOP problem instances.

- **Combinatorial auction problems:** We vary the number of agents (= bids to consider) from 5 to 35. Each agent always has two values (= bid results). Each bid is a bid on a random bundle of three out of fifty items. The cost of assigning two bids that share a common item as winning bids is infinity (to be precise: 1,000,000) since two bids cannot both win a common item. The cost of assigning two bids that share a common item as losing bids is 100. All other costs are randomly generated from 0 to 100. We average experimental results over 50 DCOP problem instances, which we generate randomly using the CATS problem suite (Leyton-Brown, Pearson, & Shoham, 2000) and the L3 distribution (Sandholm, 2002).

3.4.3 Experimental Results

Figure 3.7 shows our experimental results for ADOPT, BnB-ADOPT and NCBB on graph coloring problems with constraint costs ranging from 0 to 10,000, where we varied the number of vertices, while Figure 3.8 shows our experimental results for ADOPT, BnB-ADOPT and NCBB on graph coloring problems with 10 vertices, where we varied the range of constraint costs. Figures 3.7(a-c) and 3.8(a-c) show the results for coloring sparse

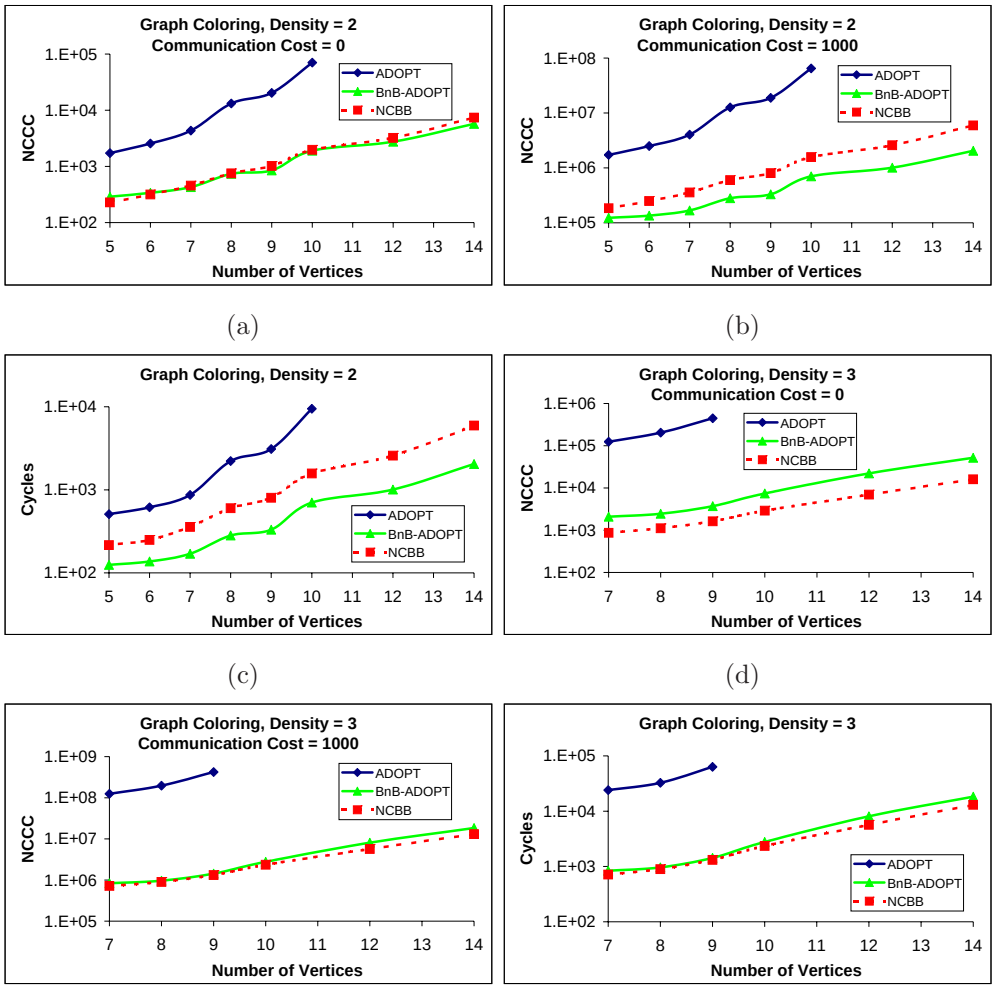


Figure 3.7: Experimental Results Comparing ADOPT, BnB-ADOPT and NCBB on Graph Coloring Problems with Constraint Costs Ranging from 0 to 10,000

graphs, and Figures 3.7(d-f) and 3.8(d-f) show the results for coloring dense graphs. The y-axes are in log scale and show the runtimes in NCCCs or cycles. We make the following observations:

- DCOP search algorithms on sparse graphs are faster than on dense graphs because, for example, there is a larger likelihood of independent DCOP subproblems in sparse graphs.

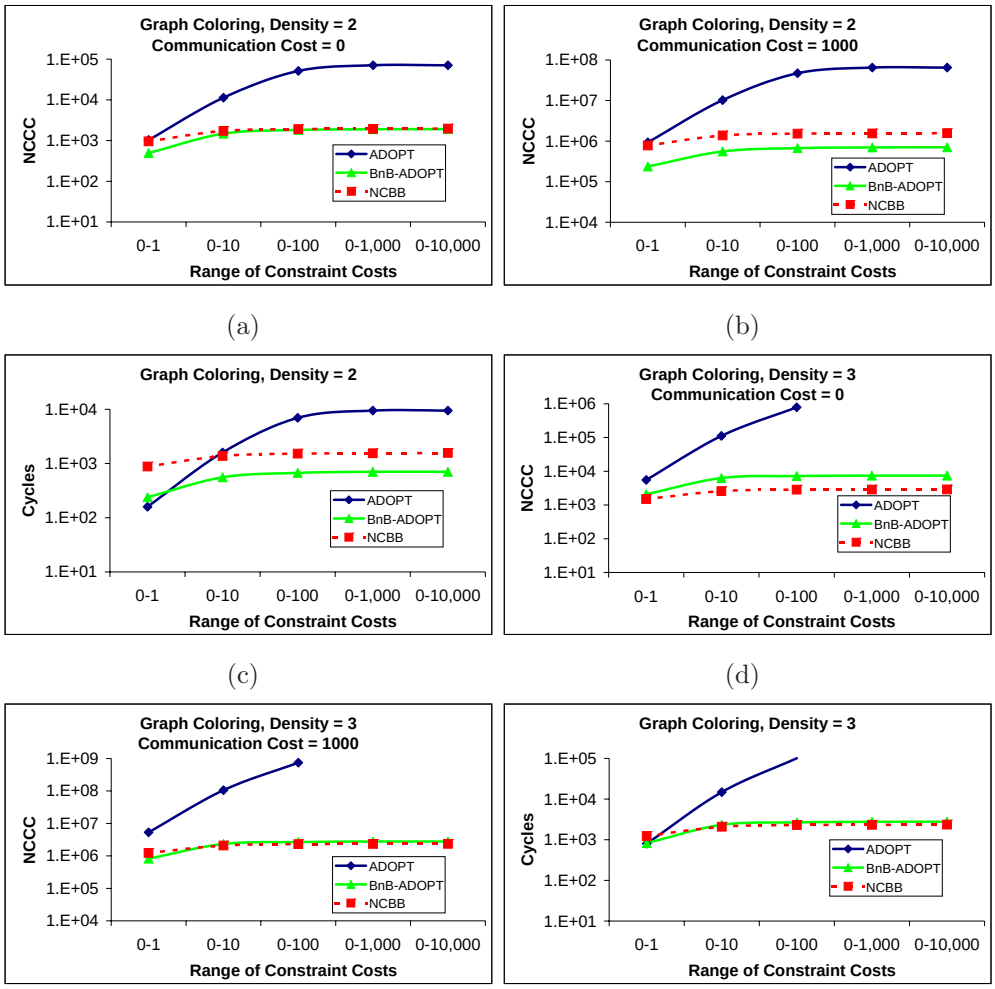


Figure 3.8: Experimental Results Comparing ADOPT, BnB-ADOPT and NCBB on Graph Coloring Problems with 10 Vertices

- BnB-ADOPT is generally faster than NCBB on sparse graphs but not on dense graphs because BnB-ADOPT allows agents to send messages only to their parent agents in the pseudo-tree (along edges of the pseudo-tree) but NCBB allows agents also to send messages to their pseudo-parent agents (along backedges of the pseudo-tree). Thus, agents in NCBB receive updates faster than agents in BnB-ADOPT. This effect is more prevalent in dense graphs since there are more backedges in

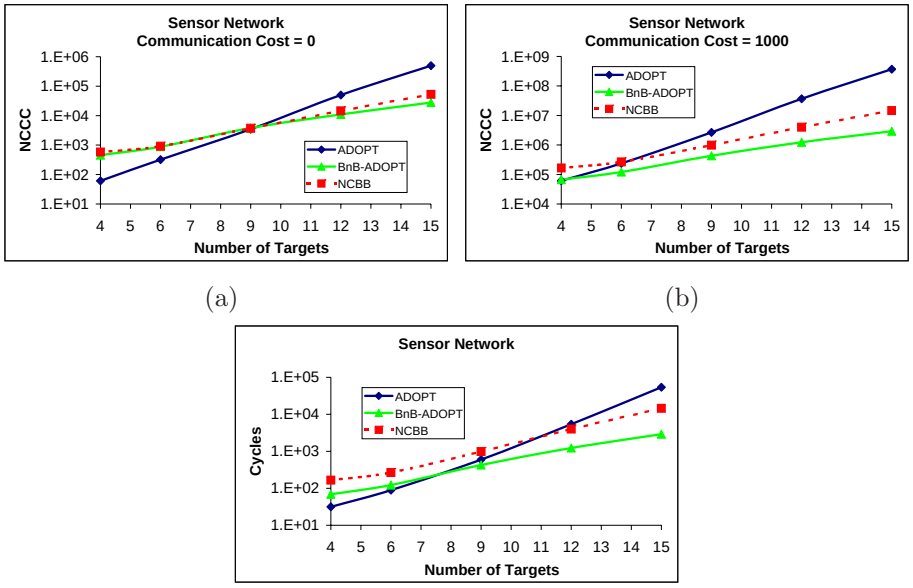


Figure 3.9: Experimental Results Comparing ADOPT, BnB-ADOPT and NCBB on Sensor Network Problems

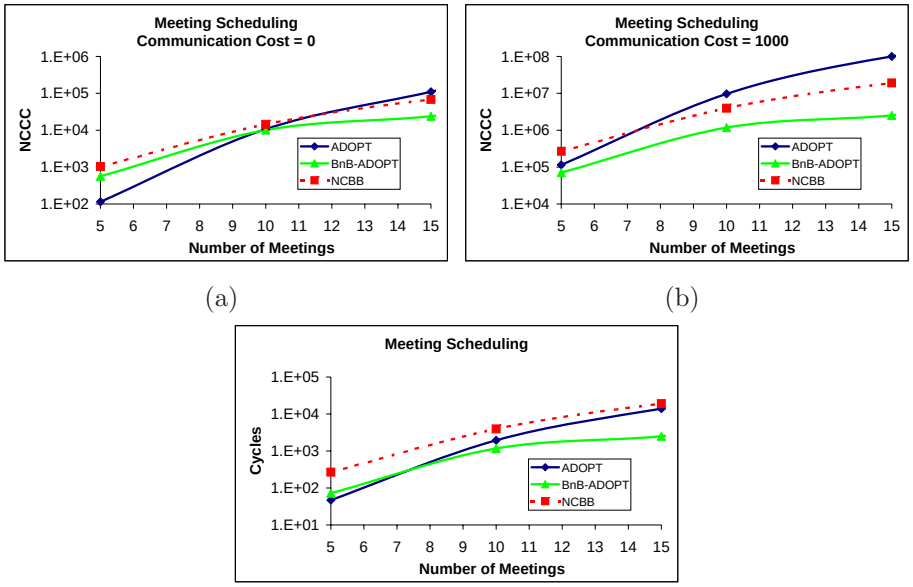


Figure 3.10: Experimental Results Comparing ADOPT, BnB-ADOPT and NCBB on Meeting Scheduling Problems

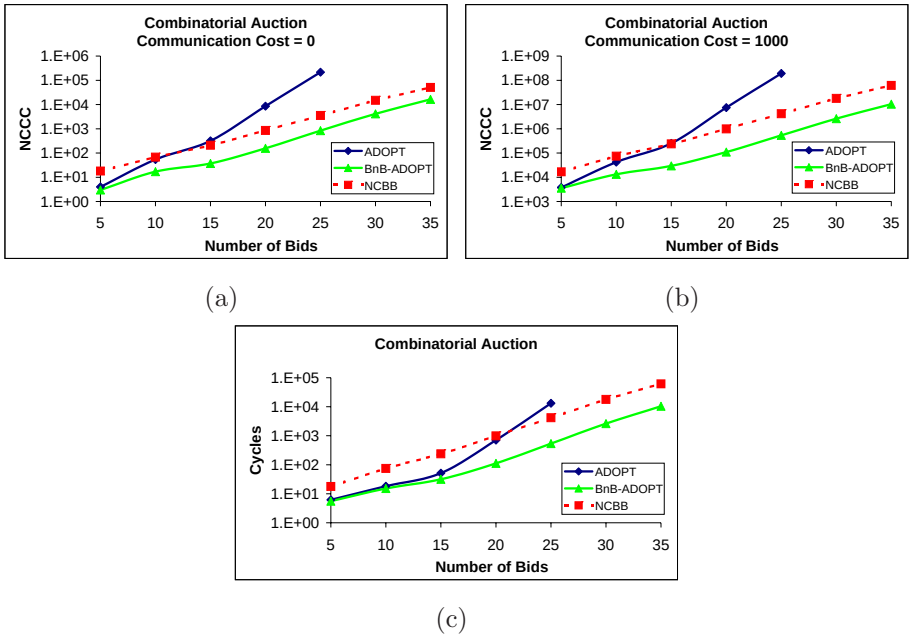


Figure 3.11: Experimental Results Comparing ADOPT, BnB-ADOPT and NCBB on Combinatorial Auction Problems

dense graphs. However, the difference between BnB-ADOPT and NCBB becomes negligible when communication is slow.

- Figure 3.7 shows that BnB-ADOPT is at least half an order of magnitude faster than ADOPT when the number of vertices is small. The speedup over ADOPT increases as the number of vertices gets larger and the DCOP problems thus become more complex. Similarly, Figure 3.8 shows that the speedup over ADOPT increases as the range of constant costs increases and the DCOP problems thus become more complex. However, ADOPT can be faster than BnB-ADOPT for simple DCOP problems. For example, ADOPT requires fewer cycles than BnB-ADOPT for DCOP problems with constraint costs ranging from 0 to 1. Figures 3.9, 3.10 and 3.11 show that the trend for graph coloring problems carries over to sensor network, meeting scheduling and combinatorial auction problems, respectively, as

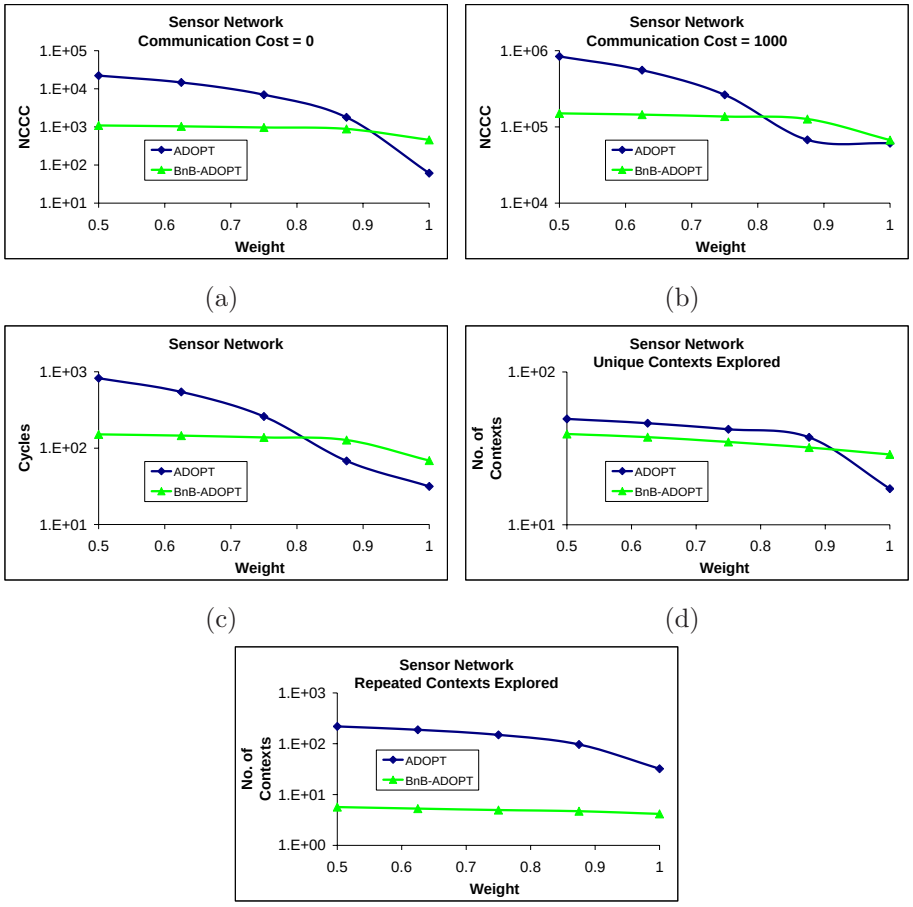


Figure 3.12: Experimental Results on the Reason for the Speedup of BnB-ADOPT over ADOPT

well. The reason for this behavior is as follows. ADOPT uses memory-bounded best-first search and thus exploits the heuristic values well but needs to repeatedly reconstruct partial solutions that it purged from memory, especially if the heuristic values are poorly informed. BnB-ADOPT uses depth-first branch-and-bound search and thus does not exploit the heuristic values quite as well but does not have to

repeatedly reconstruct partial solutions. ADOPT can thus be faster than BnB-ADOPT for DCOP problems with well informed heuristic values, such as simple DCOP problems.

We confirm this intuition with an additional experiment on sensor network problems with four targets and different informedness of heuristic values. We use the heuristic values $c \cdot h^{a,c}_{X^a}(d)$ for $0.5 \leq c \leq 1$, where $h^{a,c}_{X^a}(d)$ are the heuristic values calculated by DP2, as used until now. Figures 3.12(a-c) show the number of NCCCs for different weights c. When the heuristic values are well informed (large weights), ADOPT can indeed be faster than BnB-ADOPT. Since ADOPT relies on the heuristic values more than BnB-ADOPT, the speedup of ADOPT is much larger than that of BnB-ADOPT as the heuristic values get more informed. Figures 3.12(d,e) show the number of unique (= different) and repeated contexts per agent for different weights c. When the heuristic values are well informed (large weights), agents in ADOPT explore fewer unique contexts than agents in BnB-ADOPT since they are more focused in their search. However, when the heuristic values are poorly informed (small weights), they explore more unique contexts. Agents in ADOPT explore many more repeated contexts than agents in BnB-ADOPT since they need to reconstruct partial solutions that they purged from memory. Agents in BnB-ADOPT explore a few repeated contexts even though it does not have to reconstruct partial solutions. The reason for this behavior is the distributed nature of BnB-ADOPT. For example, assume that the context of an agent is $\{(a_1, 0), (a_2, 0)\}$ and the next context of a centralized variant of BnB-ADOPT would be $\{(a_1, 1), (a_2, 1)\}$ (where the IDs are omitted for simplicity). The agent updates its context to $\{(a_1, 1), (a_2, 0)\}$ when

it receives the message from agent a_1 that it takes on value 1. The agent then updates its context to $\{(a_1, 1), (a_2, 1)\}$ when it receives the message from agent a_2 that it takes on value 1. Thus, the agent explores the intermediate context $\{(a_1, 1), (a_2, 0)\}$ that a centralized variant of BnB-ADOPT would not explore. It counts as a repeated context if the agent explores this context intentionally in the future.

Overall, BnB-ADOPT tends to be experimentally faster than ADOPT if the heuristic values are poorly informed (small weights). Thus, BnB-ADOPT has great potential as a DCOP search algorithm since heuristic values often become more poorly informed as the DCOP problems have larger numbers of agents (Ali et al., 2005), larger domains, larger numbers of constraints or larger ranges of constraint costs, or in other words, as the DCOP problems become more complex.

3.5 Summary

This chapter introduced BnB-ADOPT, a DCOP search algorithm that has the same memory requirement, observes the same communication restriction and uses the same agent operation and pseudo-tree structure as ADOPT, but employs a depth-first branch-and-bound search strategy instead of a memory-bounded best-first search strategy. Our experimental results showed that BnB-ADOPT tends to be faster than ADOPT when the heuristic values are poorly informed since ADOPT needs to repeatedly reconstruct partial solutions that it purged from memory. In large DCOP problems, where the heuristic values are often poorly informed, BnB-ADOPT is up to one order of magnitude

faster than ADOPT. These results experimentally validate the hypothesis that DCOP search algorithms that employ depth-first branch-and-bound search can be faster than DCOP search algorithms that employ memory-bounded best-first search.

Chapter 4

Speeding Up via Approximation Mechanisms

This chapter introduces a new approximation mechanism called the Weighted Heuristics mechanism. It allows ADOPT and BnB-ADOPT to use weighted heuristic values to trade off solution costs for smaller runtimes. Additionally, it also guarantees that the costs of the solutions found are at most a constant factor larger than the minimal costs, where the constant is the largest weight used. Our experimental results show that, when ADOPT and BnB-ADOPT use the Weighted Heuristics mechanism, they terminate faster with larger weights, validating the hypothesis that DCOP search algorithms that use weighted heuristic values can have runtimes that decrease as larger weights are used. Additionally, we also introduce the Relative Error mechanism that is an extension of the existing Absolute Error mechanism. The new Weighted Heuristics and Relative Error mechanisms provide relative error bounds and thus complement the existing Absolute Error mechanism, which only provides absolute error bounds.

This chapter is organized as follows: We first describe the motivation for our work in Section 4.1. Next, we provide a detailed description of the Weighted Heuristics mechanism and existing approximation mechanisms in Section 4.2. We then prove their correctness

and completeness and describe their space complexity in Section 4.3 before presenting our experimental results in Section 4.4 and our summary in Section 4.5.

4.1 Motivation

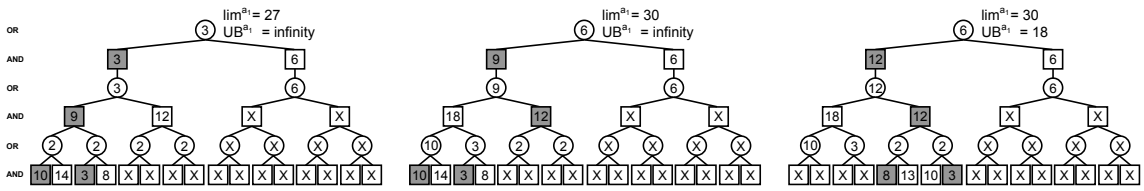
Researchers have developed centralized search algorithms, such as Weighted A* (Pohl, 1970) and Weighted A* with dynamic weights (Pohl, 1973), that use weighted heuristic values to trade off solution costs for smaller runtimes. These algorithms also provide a quality guarantee, namely that the costs of the solutions found are at most a constant factor larger than the minimal costs, where the constant is the largest weight used. Typically, the runtime of these algorithms decreases as larger weights are used.

Since DCOP search algorithms such as ADOPT and BnB-ADOPT use heuristic values to focus their search, I hypothesize that DCOP search algorithms that use weighted heuristic values can have runtimes that decrease as larger weights are used. Additionally, the costs of the solutions found should be at most a constant factor larger than the minimal costs, where the constant is the largest weight used. Therefore, I introduce the Weighted Heuristics mechanism, an approximation mechanism that trades off solution costs for smaller runtimes. Thus, the comparison of the runtimes of ADOPT and BnB-ADOPT with the Weighted Heuristics mechanism using different weights will experimentally assess my hypothesis. This work is non-trivial ADOPT and BnB-ADOPT use heuristic values differently than centralized search algorithms, such as A*. Furthermore, the proof of the quality guarantee for DCOP search algorithms cannot be trivially obtained from the proof for centralized search algorithms since DCOP search algorithms

are distributed and agents only have local views of the problem or, in other words, they know only the agents that they share constraints with and the costs of those constraints.

4.2 Approximation Mechanisms

Since solving DCOP problems optimally is NP-hard, it is desirable to find approximation mechanisms that trade off solution costs for smaller runtimes. However, to the best of our knowledge, there only exists one approximation mechanism, namely the Absolute Error mechanism, that is designed for DCOP search algorithms. The Absolute Error mechanism was originally developed for ADOPT (Modi et al., 2005), but it can be extended to work with BnB-ADOPT. It uses early termination detection to trade off solution costs for smaller runtimes by allowing users to specify absolute error bounds on the solution costs, for example that the solution costs should be at most 10 larger than minimal. The downside of this approximation mechanism is that it is impossible to set relative error bounds, for example that the solution costs should be at most 10 percent larger than minimal, without knowing the minimal costs. Therefore, in addition to the Weighted Heuristics mechanism, we also present the Relative Error mechanism, which is an extension of the Absolute Error mechanism. The two new approximation mechanisms allow users to set relative error bounds. We describe the Absolute Error mechanism in Section 4.2.1, the Relative Error mechanism in Section 4.2.2 and the Weighted Heuristics mechanism in Section 4.2.3. We only describe how these approximation mechanisms are used by BnB-ADOPT, but the same description applies for ADOPT unless otherwise mentioned.



Cycle 1 Cycle 2 Cycle 3

Figure 4.1: Trace of the Updates of the Lower Bounds of BnB-ADOPT$_{AEM}$ with Absolute Error Bound $b = 24$

All approximation mechanisms let the root agent r (and only the root agent) maintain the limit lim^r. The root agent uses this limit in the same way in the termination condition for all approximation mechanisms but updates it differently. The termination condition $UB^r_{X^r} \leq LB^r_{X^r}$ on Line 27 of the pseudocode of BnB-ADOPT is replaced with $UB^r_{X^r} \leq lim^r$. Therefore, $lim^r = LB^r_{X^r}$ for BnB-ADOPT, and $lim^r = TH^r_{X^r}$ for ADOPT. The root agent updates the limit between Lines 26 and 27 in the pseudocode, outside of the preceding if statement.

4.2.1 Absolute Error Mechanism

The Absolute Error mechanism of ADOPT requires a user-defined absolute error bound $0 \leq b < \infty$ that specifies that the solution cost should be at most b larger than the minimal solution cost. This approximation mechanism can easily be modified for BnB-ADOPT by setting the limit as follows:

$$lim^r \quad := \quad b + LB^r_{X^r} \tag{4.1}$$

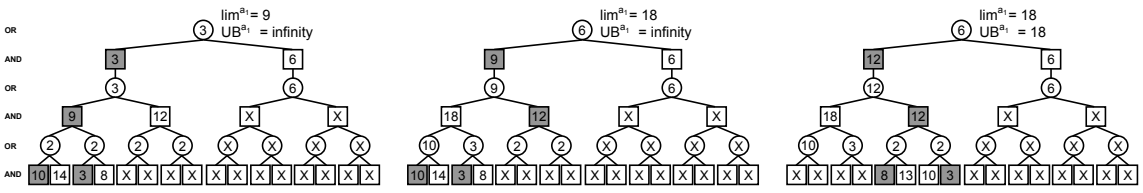


Figure 4.2: Trace of the Updates of the Lower Bounds of BnB-ADOPT$_{REM}$ with Relative Error Bound $p = 3$

BnB-ADOPT$_{AEM}$ is the resulting variant of BnB-ADOPT with the Absolute Error mechanism. BnB-ADOPT$_{AEM}$ terminates once the upper bound of the root node (which is equal to the solution cost of the solution with the smallest solution cost found so far) is no larger than the limit (which is equal to the absolute error bound b plus the lower bound of the root node, which is a lower bound on the minimal solution cost). BnB-ADOPT$_{AEM}$ terminates with a solution cost that is equal to the upper bound of the root node although the minimal solution cost could be as small as the lower bound of the root node. It thus terminates with a solution cost that is at most b larger than the minimal solution cost.

Figure 4.1 shows a trace of BnB-ADOPT$_{AEM}$ with $b = 24$ for our example DCOP problem. In Cycle 3, the lower bound $LB^{a_1}_{X^{a_1}}$ of 6 indicates that the minimal cost is at least 6, and the upper bound $UB^{a_1}_{X^{a_1}}$ of 18 indicates that BnB-ADOPT$_{AEM}$ found a solution with cost 18. Since the solution cost is at most 24 larger than the minimal cost, BnB-ADOPT$_{AEM}$ terminates after three cycles with suboptimal solution cost 18, which is six cycles faster than BnB-ADOPT.

4.2.2 Relative Error Mechanism

It is often much more desirable to specify a relative error bound on the solution cost rather than an absolute error bound. Fortunately, the Absolute Error mechanism of BnB-ADOPT can easily be changed to the Relative Error mechanism by setting the limit as follows. The Relative Error mechanism requires a user-defined relative error bound $1 \leq p < \infty$ that specifies that the solution cost should be at most p times larger than the minimal solution cost:

$$lim^r \quad := \quad p \cdot LB^r_{X^r} \tag{4.2}$$

BnB-ADOPT$_{REM}$ is the resulting variant of BnB-ADOPT with the Relative Error mechanism. BnB-ADOPT$_{REM}$ terminates once the upper bound of the root node (which is equal to the solution cost of the solution with the smallest solution cost found so far) is no larger than the limit (which is equal to the relative error bound p times the lower bound of the root node, which is a lower bound on the minimal solution cost). BnB-ADOPT$_{REM}$ terminates with a solution cost that is equal to the upper bound of the root node although the minimal solution cost could be as small as the lower bound of the root node. It thus terminates with a solution cost that is at most p times larger than the minimal solution cost.

Figure 4.2 shows a trace of BnB-ADOPT$_{REM}$ with $p = 3$ for our example DCOP problem. In Cycle 3, the lower bound $LB^{a_1}_{X^{a_1}}$ of 6 indicates that the minimal cost is

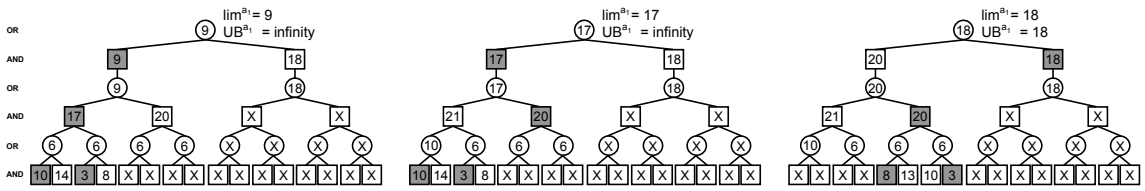


Figure 4.3: Trace of the Updates of the Lower Bounds of BnB-ADOPT$_{WHM}$ with Relative Error Bound $w = 3$

at least 6, and the upper bound $UB^{a_1}_{X^{a_1}}$ of 18 indicates that BnB-ADOPT$_{REM}$ found a solution with cost 18. Since the solution cost is at most 3 times larger than the minimal cost, BnB-ADOPT$_{REM}$ terminates after three cycles with suboptimal solution cost 18, which is six cycles faster than BnB-ADOPT.

4.2.3 Weighted Heuristics Mechanism

As mentioned in Section 3.2.2, BnB-ADOPT uses heuristic values to approximate the gamma costs of nodes in the search tree. Each agent a initializes $lb^{a,c}_{X^a}(d) := h^{a,c}_{X^a}(d)$ for admissible heuristic values $0 \leq h^{a,c}_{X^a}(d) \leq \gamma^c_{X^a \cup (a,d)}$ for all values $d \in Dom(a)$, all child agents $c \in C(a)$ and current context X^a. If these heuristic values are close to the gamma costs the runtimes will be small.

It is common practice in the context of A* to trade off solution cost for a smaller runtime by using weighted heuristic values (Pohl, 1973), which are derived from admissible heuristic values by multiplying them with a user-defined weight $1 \leq w < \infty$. The resulting heuristic values can be inadmissible. A* is then no longer guaranteed to find cost-minimal solutions but it is guaranteed to terminate with a solution cost that is at most w times larger than the minimal solution cost (Pohl, 1970). This approximation mechanism can easily be modified for BnB-ADOPT by setting the limit as follows:

$$lim^r \quad := \quad LB^r_{X^r} \tag{4.3}$$

and by initializing the lower bounds $lb^{a,c}_{X^a}(d)$ as follows:

$$lb^{a,c}_{X^a}(d) \quad := w \cdot h^{a,c}_{X^a}(d) \tag{4.4}$$

for all agents a, all values d, all child agents c and all contexts X^a. BnB-ADOPT$_{WHM}$ is the resulting variant of BnB-ADOPT with the Weighted Heuristics mechanism. BnB-ADOPT$_{WHM}$ terminates once the upper bound of the root node (which is equal to the solution cost of the solution with the smallest solution cost found so far) is no larger than the limit (which is equal to the lower bound of the root node, which is a lower bound on w times the minimal solution cost). BnB-ADOPT$_{WHM}$ terminates with a solution cost that is equal to the upper bound of the root node although the minimal solution cost could be as small as the lower bound of the root node divided by w. It thus terminates with a solution cost that is at most w times larger than the minimal solution cost.

Figure 4.3 shows a trace of BnB-ADOPT$_{WHM}$ with $w = 3$ for our example DCOP problem. In Cycle 3, the lower bound $LB^{a_1}_{X^{a_1}}$ of 18 indicates that the (overestimated)

minimal cost is at least 18, and the upper bound $UB^{a_1}_{X^{a_1}}$ of 18 indicates that BnB-ADOPT$_{WHM}$ found a solution with cost 18. (The lower bound is an overestimate of the minimal cost since BnB-ADOPT$_{WHM}$ uses weighted heuristic values.) Since the solution cost is no larger than the (overestimated) minimal cost, BnB-ADOPT$_{WHM}$ terminates after three cycles with suboptimal solution cost 18, which is six cycles faster than BnB-ADOPT.

4.3 Correctness, Completeness and Complexity

In this section, we prove the correctness and completeness of the approximation mechanisms when used by BnB-ADOPT. Their correctness and completeness when used by ADOPT can be proven in a similar way. We also describe their space and message complexities.

4.3.1 Correctness and Completeness

All definitions, lemmas and theorems of Section 3.3.1 continue to hold when BnB-ADOPT uses the approximation mechanisms. We follow the assumptions in Section 3.3.1 and assume that each agent a uses the following update equation for all values d, all child agents c and all contexts X^a to initialize its bounds.

$$lb^{a,c}_{X^a}(d) := w \cdot h^{a,c}_{X^a}(d) \tag{3.20}$$

$$ub^{a,c}_{X^a}(d) := \infty \tag{3.21}$$

where the weight w is a floating point number that satisfies $1 \leq w < \infty$ and the heuristic values $h^{a,c}_{X^a}(d)$ are floating point numbers that satisfy

$$0 \leq h^{a,c}_{X^a}(d) \leq \gamma^{c}_{X^a \cup (a,d)} \tag{3.22}$$

Theorem 3 *The suboptimal variants of BnB-ADOPT terminate after a finite amount of time.*

Proof: If the suboptimal variants of BnB-ADOPT do not terminate earlier, then $UB^a_{X^a} \leq LB^a_{X^a}$ after a finite amount of time for all agents $a \in A$ and their contexts X^a according to Lemma 8. In particular, $UB^r_{X^r} \leq LB^r_{X^r} \leq lim^r$ for the root agent r, where $lim^r = LB^r_{X^r}$ for BnB-ADOPT$_{WHM}$, $lim^r = b + LB^r_{X^r}$ with $b \geq 0$ for BnB-ADOPT$_{AEM}$ and $lim^r = p \cdot LB^r_{X^r}$ with $p \geq 1$ for BnB-ADOPT$_{REM}$ according to Section 4.2. Thus, the termination condition $UB^r_{X^r} \leq lim^r$ of the suboptimal variants of BnB-ADOPT are satisfied. ■

Theorem 4 *BnB-ADOPT$_{AEM}$ terminates with a solution cost that is bounded from above by the user-defined absolute error bound b plus the minimal solution cost $\gamma^r_{X^r}$.*

Proof: BnB-ADOPT$_{AEM}$ terminates after a finite amount of time according to Theorem 3. The solution cost of BnB-ADOPT$_{AEM}$ is the upper bound $UB^r_{X^r}$ of the root agent r. $UB^r_{X^r} \leq lim^r = b + LB^r_{X^r}$ upon termination according to its termination condition. $LB^r_{X^r} \leq w \cdot \gamma^r_{X^r}$ according to Lemma 5. Therefore, $UB^r_{X^r} \leq b + \gamma^r_{X^r}$ since $w = 1$. ■

Theorem 5 *BnB-ADOPT$_{REM}$ terminates with a solution cost that is bounded from above by the user-defined relative error bound p times the minimal solution cost $\gamma^r_{X^r}$.*

Proof: BnB-ADOPT$_{REM}$ terminates after a finite amount of time according to Theorem 3. The solution cost of BnB-ADOPT$_{REM}$ is the upper bound $UB^r_{X^r}$ of the root agent r. $UB^r_{X^r} \leq lim^r = p \cdot LB^r_{X^r}$ upon termination according to its termination condition. $LB^r_{X^r} \leq w \cdot \gamma^r_{X^r}$ according to Lemma 5. Therefore, $UB^r_{X^r} \leq p \cdot \gamma^r_{X^r}$ since $w = 1$. ■

Theorem 6 *BnB-ADOPT$_{WHM}$ terminates with a solution cost that is bounded from above by the user-defined weight w times the minimal solution cost $\gamma^r_{X^r}$.*

Proof: BnB-ADOPT$_{WHM}$ terminates after a finite amount of time according to Theorem 3. The solution cost of BnB-ADOPT$_{WHM}$ is the upper bound $UB^r_{X^r}$ of the root agent r. $UB^r_{X^r} \leq lim^r = LB^r_{X^r}$ upon termination according to its termination condition. $LB^r_{X^r} \leq w \cdot \gamma^r_{X^r}$ according to Lemma 5. Therefore, $UB^r_{X^r} \leq w \cdot \gamma^r_{X^r}$. ■

4.3.2 Complexity

Since ADOPT and BnB-ADOPT with any of the three approximation mechanisms only maintain one additional variable, namely lim^r, their memory requirements are identical to ADOPT and BnB-ADOPT without any approximation mechanism. Thus, their space complexity is identical to the one described in Section 3.3.2. They use the same messages as ADOPT and BnB-ADOPT and their message complexity is thus identical to the one described in Section 3.3.2 as well.

4.4 Experimental Evaluation

We now compare ADOPT and BnB-ADOPT with the different approximation mechanisms and the DP2 pre-processing framework described in Section 3.4. We also compare ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ to MGM-k (Pearce, Maheswaran, & Tambe, 2006; Pearce & Tambe, 2007), an incomplete k-optimal DCOP algorithm. It partitions the DCOP problem into groups of at most k agents and runs a local search algorithm to find a DCOP solution that is cost-minimal within each group.

4.4.1 Metrics

We measure the runtimes in cycles. We do not measure the runtimes in NCCCs because the number of cycles reflects the number of NCCCs; the number of constraint checks and the number of messages sent in each cycle are very similar for both ADOPT and BnB-ADOPT. We report normalized runtimes, that is, the runtimes divided by the runtimes for finding cost-minimal solutions. Thus, the normalized runtime 0.25 refers to one quarter of the number of cycles that it takes to find a cost-minimal solution. We also measure the solution costs found by the algorithms. We report normalized solution costs, that is, the solution costs divided by the minimal solution costs. Thus, the normalized solution cost 2.5 refers to a solution cost that is two and a half times larger than the minimal solution cost. Lastly, we also measure the number of unique and repeated contexts explored per agent of ADOPT and BnB-ADOPT to better understand the reason for the speedups. The sum of both numbers is correlated with the runtime of ADOPT and BnB-ADOPT.

We vary the relative error bound, which is the worst acceptable normalized solution cost, from 1.0 to 4.0. These relative error bounds correspond to p for the Relative Error mechanism and w for the Weighted Heuristics mechanism. We pre-calculate the minimal costs to set the correct value of b for the Absolute Error mechanism. For example, if the minimal cost is 100 and the relative error bound is 2.5, $p = 2.5$ for the Relative Error mechanism, $w = 2.5$ for Weighted Heuristics mechanism, and $b = (2.5 - 1) \cdot 100 = 150$ for the Absolute Error mechanism.

4.4.2 Problem Types

As described in Section 2.1.5, we run our experiments on four problem types, namely graph coloring, sensor network, meeting scheduling and combinatorial auction problems.

- **Graph coloring problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= vertices to color) to 10 and the density to 2. Each agent always has three possible values (= colors). All costs are randomly generated from 0 to 10,000. We average the experimental results over 50 DCOP problem instances.

- **Sensor network problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= targets to track) to 9. Each agent always has nine values (= time slots). We average the experimental results over 50 DCOP problem instances.

- **Meeting scheduling problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= meetings to schedule) to 10.

Each agent always has nine values (= time slots). We average the experimental results over 50 DCOP problem instances.

- **Combinatorial auction problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= bids to consider) to 25. Each agent always has two values (= bid results). We average the experimental results over 50 DCOP problem instances.

4.4.3 Experimental Results

Figures 4.4 and 4.5 show our experimental results for ADOPT and BnB-ADOPT with the approximation mechanisms, respectively, on graph coloring problems. We make the following observations:

- Figures 4.4(a) and 4.5(a) show that the normalized solution costs of all suboptimal variants of ADOPT and BnB-ADOPT increase as the relative error bound increases. However, the solution costs remains much smaller than the error bound. For example, the normalized solution costs of all suboptimal variants are less than 1.3 (rather than 3) when the relative error bound is 3. The normalized solution costs of ADOPT and BnB-ADOPT with the Absolute Error mechanism are usually larger than the normalized solution costs with the Relative Error mechanism for the same relative error bound. The reason for this behavior is that ADOPT and BnB-ADOPT with the Absolute Error mechanism terminate when $UB^r_{X^r} \leq lim^r = b + LB^r_{X^r} = (p-1) \cdot \gamma^r_{X^r} + LB^r_{X^r}$, where $\gamma^r_{X^r}$ is the minimal solution cost. Thus, the solution cost with the Absolute Error mechanism can be

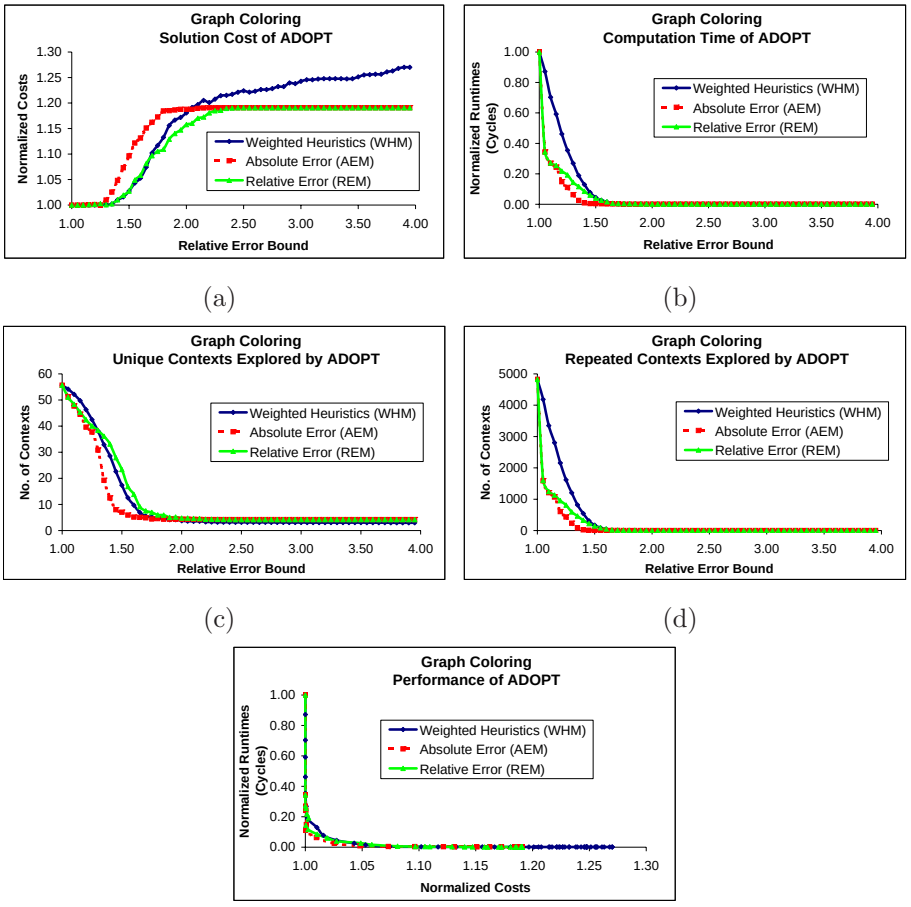


Figure 4.4: Experimental Results Comparing ADOPT with the Approximation Mechanisms on Graph Coloring Problems

at most $UB^r_{X^r} - LB^r_{X^r} \leq (p-1) \cdot \gamma^r_{X^r}$ larger than the minimal solution cost. On the other hand, ADOPT and BnB-ADOPT with the Relative Error mechanism terminate when $UB^r_{X^r} \leq lim^r = p \cdot LB^r_{X^r}$. Thus, the solution cost with the Relative Error mechanism can be at most $UB^r_{X^r} - LB^r_{X^r} \leq (p-1) \cdot LB^r_{X^r}$ larger than the minimal solution cost. The absolute error bound of the Absolute Error mechanism is thus no smaller than the absolute error bound of the Relative Error mechanism

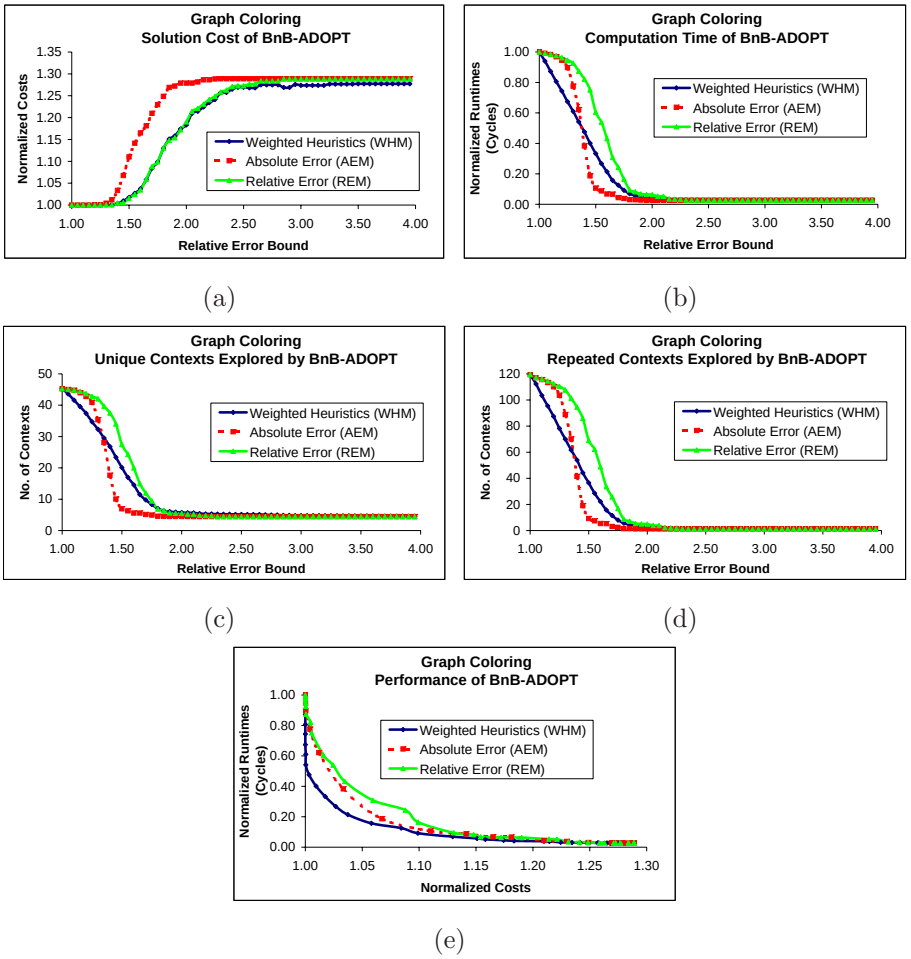


Figure 4.5: Experimental Results Comparing BnB-ADOPT with the Approximation Mechanisms on Graph Coloring Problems

since $\gamma^r_{X^r} \geq LB^r_{X^r}$ but is initially strictly greater than the absolute error bound of the Relative Error mechanism since $\gamma^r_{X^r} > LB^r_{X^r}$ during most of the search.

- Figures 4.4(b) and 4.5(b) show that the normalized runtimes of all suboptimal variants of ADOPT and BnB-ADOPT decrease as the relative error bound increases. They decrease to almost 0 when the relative error bound is about 1.5 for ADOPT and 2.0 for BnB-ADOPT. Therefore, all suboptimal variants terminate almost immediately after finding the first solution. The normalized runtimes of ADOPT

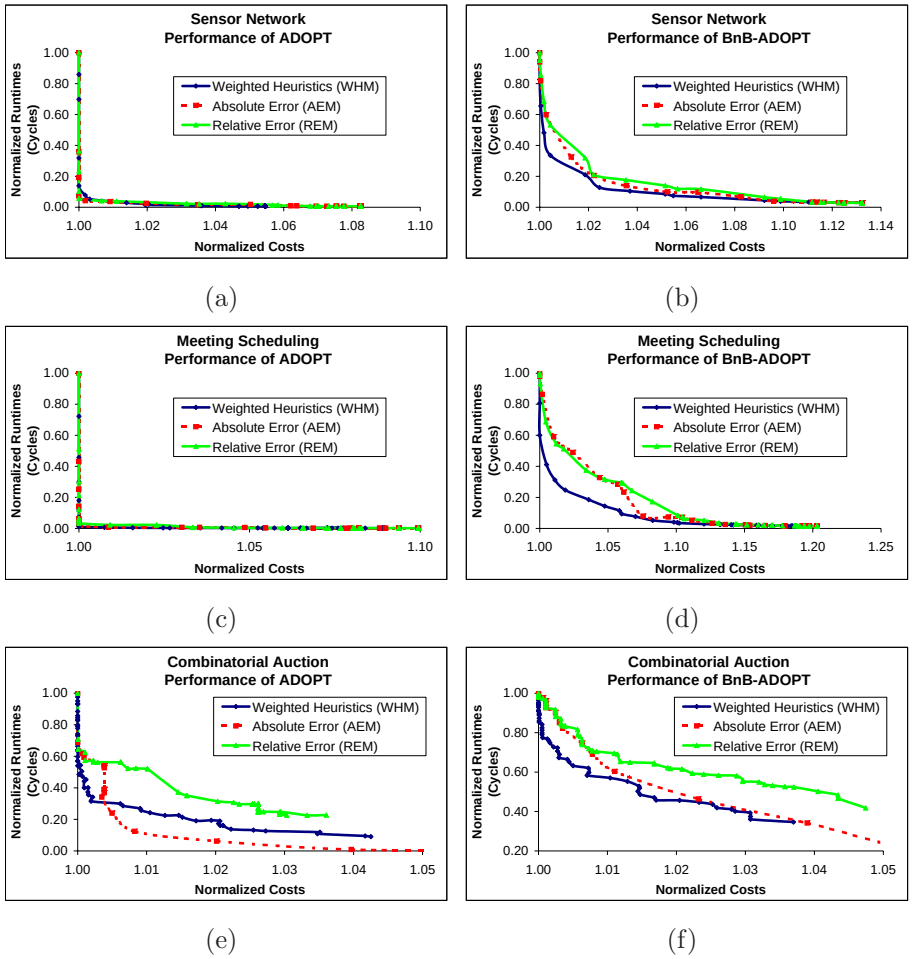


Figure 4.6: Experimental Results Comparing ADOPT and BnB-ADOPT with the Approximation Mechanisms on Sensor Network, Meeting Scheduling and Combinatorial Auction Problems

and BnB-ADOPT with the Absolute Error mechanism are usually smaller than the normalized runtimes with the Relative Error mechanism for the same relative error bound. The reason for this behavior is that ADOPT and BnB-ADOPT with the Absolute Error mechanism can terminate with a suboptimal solution cost that is within its absolute error bound but not yet within the absolute error bound of ADOPT and BnB-ADOPT with the Relative Error mechanism. In other words, ADOPT and BnB-ADOPT with the Absolute Error mechanism can terminate with

a suboptimal solution cost $(p-1) \cdot LB^r_{X^r} < UB^r_{X^r} \leq (p-1) \cdot \gamma^r_{X^r}$ while ADOPT and BnB-ADOPT with the Relative Error mechanism can not.

- Figures 4.4(c,d) and 4.5(c,d) give insight into the reasons for the decrease in normalized runtime. They show that the number of unique and repeated contexts per agent decreases as the relative error bound increases, indicating that both the size of the search space explored by all suboptimal variants of ADOPT and BnB-ADOPT and the size of the search space that they need to reconstruct decreases. The size of the search space that the suboptimal variants of ADOPT need to reconstruct is much larger than the one that the suboptimal variants of BnB-ADOPT need to reconstruct since the suboptimal variants of ADOPT need to reconstruct abandoned partial solutions repeatedly like ADOPT.

- Figures 4.4(e) and 4.5(e) compare the different approximation mechanisms by plotting the normalized runtime needed to achieve a given normalized solution cost. For ADOPT, there does not seem to be a significant difference between the three suboptimal variants. However, for BnB-ADOPT, BnB-ADOPT$_{WHM}$ performs better than BnB-ADOPT$_{AEM}$, which in turn performs better than BnB-ADOPT$_{REM}$. For example, the normalized runtime needed to achieve a normalized solution cost of 1.05 is about 0.18 for BnB-ADOPT$_{WHM}$, 0.30 for BnB-ADOPT$_{AEM}$ and 0.35 for BnB-ADOPT$_{REM}$. Thus, BnB-ADOPT$_{WHM}$ is the suboptimal variant with the best performance.

- Figures 4.6(a,c) show that the trend for graph coloring problems carries over to sensor network and meeting scheduling problems as well. Figure 4.6(e) shows that

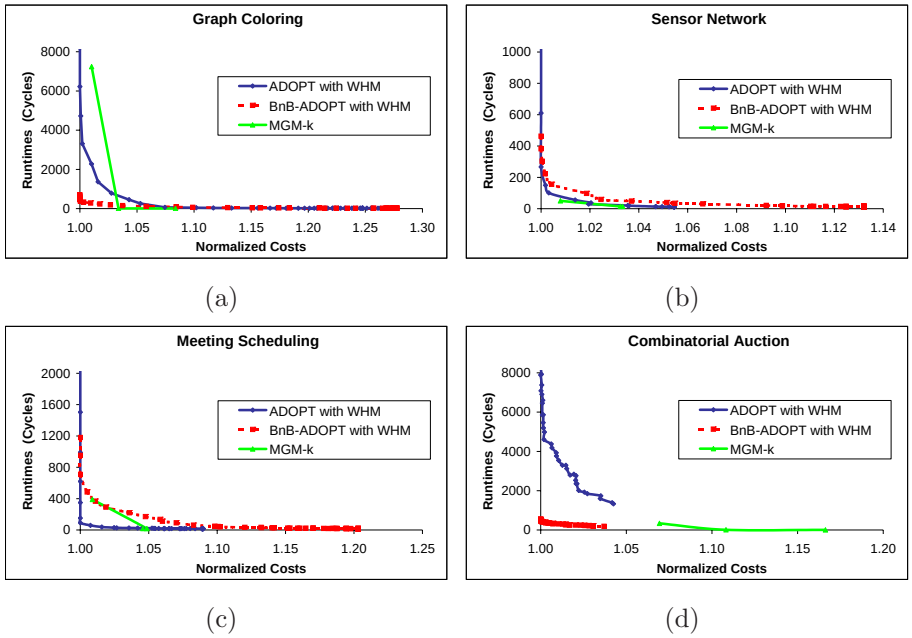


Figure 4.7: Experimental Results Comparing ADOPT$_{WHM}$, BnB-ADOPT$_{WHM}$ and MGM-k

ADOPT$_{WHM}$ performs better than ADOPT$_{AEM}$, which in turn performs better than ADOPT$_{REM}$ when they find solutions with normalized costs no larger than about 1.005 on combinatorial auction problems. ADOPT$_{AEM}$ performs better than ADOPT$_{WHM}$, which in turn performs better than ADOPT$_{REM}$ when they find solutions with larger normalized costs. Figures 4.6(b,d,f) show that the trend for graph coloring problems carries over to sensor network, meeting scheduling and combinatorial auction problems as well.

We now compare ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ to MGM-k for $k \leq 3$. We do not use larger values of k because efficient implementations are so far known only for $k \leq 3$ (Maheswaran et al., 2004). Figure 4.7 shows the results. As k increases, the normalized solution costs of MGM-k increase and the runtimes of MGM-k decrease. The performance of the DCOP algorithms appears to be domain dependent. On graph

coloring problems, BnB-ADOPT$_{WHM}$ performs better than ADOPT$_{WHM}$ and MGM-k. ADOPT$_{WHM}$ performs better than MGM-k when they find solutions with normalized costs no larger than 1.03 and performs similarly otherwise. On sensor network problems, all three DCOP algorithms perform similarly. On meeting scheduling problems, ADOPT$_{WHM}$ performs better than MGM-k and BnB-ADOPT$_{WHM}$. On combinatorial auction problems, BnB-ADOPT$_{WHM}$ performs better than ADOPT$_{WHM}$. ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ found solutions with normalized costs no larger than 1.05, and MGM-k found solutions with normalized costs no smaller than 1.05. Overall, ADOPT and BnB-ADOPT with the Weighted Heuristics mechanism have smoother tradeoffs between solution costs and runtimes than MGM-k.

Overall, the Weighted Heuristics mechanism experimentally performs better than the other tested approximation mechanisms when used by BnB-ADOPT. Therefore, it is a good complement to the existing Absolute Error mechanism that only provides absolute error bounds. In general, we expect our approximation mechanisms to apply to other DCOP search algorithms as well since all of them perform search and thus benefit from using heuristic values to focus their searches.

4.5 Summary

This chapter introduced two approximation mechanisms, the Relative Error mechanism and the Weighted Heuristics mechanism, that trade off solution costs for smaller runtimes. These two approximation mechanisms provide relative error bounds and thus complement the existing Absolute Error mechanism that only provides absolute error bounds. The

Relative Error mechanism changes the early termination condition of the Absolute Error mechanism such that it provides relative error bounds. The Weighted Heuristics mechanism uses weighted heuristic values to find suboptimal solutions and guarantees that the costs of the solutions found are at most a constant factor larger than the minimal costs, where the constant is the largest weight used. Our experimental results show that, when ADOPT and BnB-ADOPT use the Weighted Heuristics mechanism, they terminate faster with larger weights, validating the hypothesis that DCOP search algorithms that use weighted heuristic values can have runtimes that decrease as larger weights are used.

Chapter 5

Speeding Up via Caching Schemes

This chapter frames the caching problem as an optimization problem and introduces three new DCOP-specific caching schemes called the MaxPriority, MaxEffort and MaxUtility schemes. They allow the agents in ADOPT and BnB-ADOPT to determine which information to purge from memory when their memory is full and new information needs to be stored in memory. Our experimental results show that the MaxEffort and MaxUtility schemes speed up ADOPT more than the currently used generic caching schemes, and the MaxPriority scheme speeds up BnB-ADOPT at least as much as the currently used generic caching schemes. Therefore, these results validate the hypothesis that DCOP-specific caching schemes can reduce the runtime of DCOP search algorithms at least as much as the currently used generic caching schemes.

This chapter is organized as follows: We first describe the motivation for our work in Section 5.1. In Section 5.2, we provide a detailed description of the MaxPriority, MaxEffort and MaxUtility schemes. We then prove their correctness and completeness and describe their space and message complexities in Section 5.3 before presenting our experimental results in Section 5.4 and our summary in Section 5.5.

5.1 Motivation

Researchers have developed centralized search algorithms, such as MA* (Chakrabarti et al., 1989) and SMA* (Russell, 1992), that employ any-space best-first search, which is a version of memory-bounded best-first search that uses more memory than the minimal amount. These algorithms cache information as long as memory is available. Typically, the runtime of these algorithms decreases as more memory is available (Chakrabarti et al., 1989; Russell, 1992).

Motivated by these results, researchers have developed any-space versions of DCOP search algorithms, such as any-space ADOPT (Matsui et al., 2005) and any-space NCBB (Chechetka & Sycara, 2006a), that cache information as long as memory is available. However, unlike MA* and SMA*, memory is distributed among the agents in DCOP search algorithms. Therefore, the caching schemes of centralized search algorithms cannot be applied directly to DCOP search algorithms. As a result, any-space ADOPT and any-space NCBB use generic caching schemes that are similar to popular page replacement schemes used in operating systems. For example, any-space ADOPT uses the Least-Recently-Used (LRU) caching scheme and any-space NCBB uses the First-In-First-Out (FIFO) caching scheme, which allow the agents to determine which information to purge from memory when their memory is full and new information needs to be stored in memory. Despite the use of generic caching schemes, similar to the results of any-space centralized search algorithms, the runtime of these any-space DCOP search algorithms also decreases as more memory is available (Matsui et al., 2005; Chechetka & Sycara, 2006a).

I hypothesize that DCOP-specific caching schemes can reduce the runtime of DCOP search algorithms at least as much as the currently used generic caching schemes. Therefore, I introduce the MaxPriority, MaxEffort and MaxUtility schemes, three caching schemes that exploit the cached information in a DCOP-specific way. Thus, the comparison of the runtimes of ADOPT and BnB-ADOPT with the different caching schemes will experimentally assess my hypothesis. This work is non-trivial since agents ADOPT and BnB-ADOPT need to predict future information access while having only local views of the problem or, in other words, knowing only the agents that they share constraints with and the costs of those constraints.

5.2 Caching

Each agent in ADOPT and BnB-ADOPT requires only a linear (in the number of the agents) amount of memory to find a cost-minimal solution. However, if additional memory is available, it can use the additional memory to cache additional lower and upper bounds, which avoids search effort. For example, using the same definitions described in the context of Figure 2.7, Figure 5.1 shows the search strategy of ADOPT when all agents have a sufficient amount of memory to maintain all lower and upper bounds. ADOPT takes seven steps fewer since it no longer needs to re-expand nodes. We describe the information that is cached in Section 5.2.1, formulate the caching problem in Section 5.2.2 and introduce the caching schemes in Section 5.2.3.

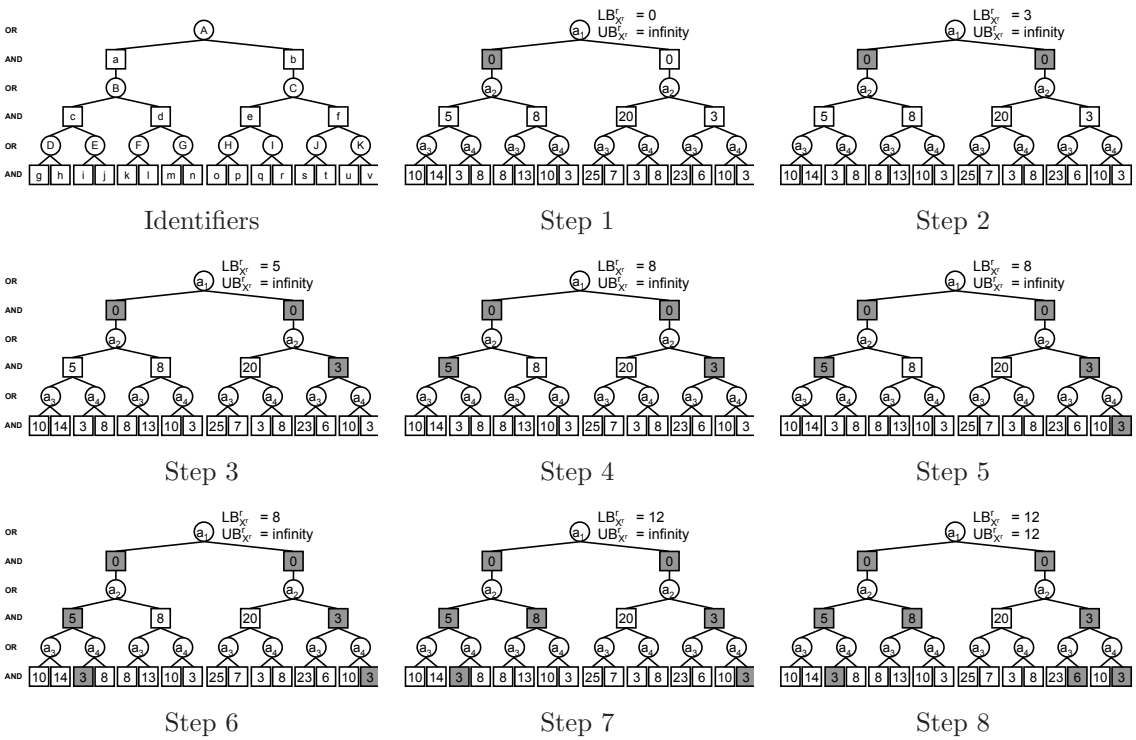


Figure 5.1: Trace of Simplified Best-First Search (Centralized ADOPT without Memory Limitations)

5.2.1 Cache Design

An information unit I consists of one context ${}^{I}X^{a}$ of agent a and the lower and upper bounds ${}^{I}lb^{a,c}_{I_{X^a}}(d)$, ${}^{I}LB^{a}_{I_{X^a}}(d)$, ${}^{I}ub^{a,c}_{I_{X^a}}(d)$ and ${}^{I}UB^{a}_{I_{X^a}}(d)$ for that context, all values $d \in Dom(a)$ and all child agents $c \in C(a)$. Each agent in ADOPT and BnB-ADOPT caches only one information unit, namely for its current context.

We now generalize ADOPT and BnB-ADOPT by allowing agents to cache more than one information unit. An agent always uses the information unit with its current context. We assume that an agent can cache a given number of information units. If the agent receives lower and upper bounds via COST messages for a context that is compatible with the context of a cached information unit, then it updates the bounds of that information

unit according to update equations 3.8 to 3.13. If the agent receives lower and upper bounds for a context that is incompatible with the context of any cached information unit and its cache is not yet full, then it creates a new information unit and caches the context and bounds in that information unit. If the agent receives lower and upper bounds for a context that is incompatible with the context of any cached information unit and the cache is full, then it ignores the bounds, with one exception: If the agent receives lower and upper bounds for a context and then switches its current context to that context, then it needs to cache them since an agent always has to cache the information unit with its current context. A caching scheme then decides which information unit to purge from the cache.

5.2.2 Caching Problem

We frame the problem of which information unit to purge from the cache as an optimization problem. We calculate the utility

$$U(I) := P(I) \cdot E(I) \tag{5.1}$$

of an information unit I based on its likelihood of future use $P(I)$ and the invested search effort $E(I)$. Each agent then greedily maximizes the sum of the utilities of all cached information units by purging an information unit with the smallest utility.

5.2.2.1 Likelihood of Future Use: P(I)

The likelihood of future use $P(I)$ measures the probability that the context of information unit I will again become the current context. It is important to use the likelihood of future use as part of a measure of the utility of an information unit because it is pointless to cache an information unit whose context will never again become the current context. It is affected by two factors:

- **Distributed Execution:** The agents in ADOPT and BnB-ADOPT can expand intermediate nodes or, synonymously, explore intermediate contexts for a short period of time when their current context changes because they operate in a distributed fashion. For example, assume that the context of an agent is $\{(a_1, 0), (a_2, 0)\}$ and the next context of a centralized variant of ADOPT and BnB-ADOPT would be $\{(a_1, 1), (a_2, 1)\}$ (where the IDs are omitted for simplicity). The agent updates its context to $\{(a_1, 1), (a_2, 0)\}$ when it receives the message from agent a_1 that it takes on value 1. The agent then updates its context to $\{(a_1, 1), (a_2, 1)\}$ when it receives the message from agent a_2 that it takes on value 1. Thus, the agent explores the intermediate context $\{(a_1, 1), (a_2, 0)\}$ for a short period of time. The more agent-value assignments a context has in common with the current context of the agent, the larger the likelihood of future use is.

- **Search Strategy:** As described in Sections 2.2.2 and 2.3.1, the depth-first branch-and-bound search strategy of BnB-ADOPT does not require agents to re-expand nodes but the memory-bounded best-first search strategy of ADOPT does require agents to re-expand nodes.

5.2.2.2 Invested Search Effort: E(I)

Existing caching schemes for DCOP search algorithms use only the likelihood of future use to measure the utility of an information unit. It is also important to use the invested search effort (= search effort that has been spent to update the lower and upper bounds of the information unit and thus will likely have to be spent again if it is purged from the cache but its context gets re-explored) as part of a measure of the utility of an information unit because it might be better to purge an information unit from the cache whose likelihood of reuse is 100 percent but whose invested search effort is almost zero than an information unit whose likelihood of future use is only 50 percent but whose invested search effort is large.

5.2.3 Caching Schemes

We classify caching schemes into three categories:

- **Category 1**: Caching schemes that purge an information unit I with the smallest likelihood of future use $P(I)$. We introduce a new MaxPriority scheme for this category.

- **Category 2**: Caching schemes that purge an information unit I with the smallest invested search effort $E(I)$. We introduce a new MaxEffort scheme for this category.

- **Category 3**: Caching schemes that purge an information unit I with the smallest utility $U(I) = P(I) \cdot E(I)$. We introduce a new MaxUtility scheme for this category.

5.2.3.1 Benchmark Schemes

We use page replacement schemes for virtual memory management from the operating systems literature as benchmark schemes. In particular, we use First-In-First-Out (FIFO) and Least-Recently-Used (LRU) as benchmark schemes of Category 1, which are similar to existing caching schemes for DCOP search algorithms. For example, any-space NCBB uses a version of the FIFO scheme, and any-space ADOPT uses a version of the LRU scheme. The FIFO scheme purges the information unit that has been in the cache for the longest time, and the LRU scheme purges the information unit that has not been used or updated for the longest time. Both caching schemes use the intuition that an information unit that has been cached, used or updated recently will likely be used again.

Similarly, we use Last-In-First-Out (LIFO) and Least-Frequently-Used (LFU) as benchmark schemes of Category 2. The LIFO scheme purges the information unit that has been in the cache for the shortest time, and the LFU scheme purges the information unit that has been used or updated the least number of times. Both caching schemes use the intuition that a large search effort has been invested in an information unit that has been in the cache for a long time (which assumes similar update frequencies for all information units) or that has been used or updated frequently (which assumes a similar ratio of use and update frequencies for all information units).

5.2.3.2 MaxPriority Scheme

The MaxPriority scheme attempts to purge the information unit I with the smallest likelihood of future use $P(I)$. The likelihood of future use of an information unit is affected by both the distributed execution and the search strategy of a DCOP search

algorithm. It is currently unknown how to best estimate the likelihood of future use due to the search strategy. The MaxPriority scheme thus estimates only the likelihood of future use due to the distributed execution. The more agent-value assignments the context of the information unit has in common with the current context of the agent, the larger the likelihood of future use due to the distributed execution is.

The MaxPriority scheme, however, uses additional knowledge of the operation of ADOPT and BnB-ADOPT in the form of the lower bounds ${}^{I}LB^{a}_{{}^{I}X^{a}}(d)$ that every agent a maintains for all values $d \in Dom(a)$ in the information unit I with its current context ${}^{I}X^{a}$. The MaxPriority scheme uses the property that every agent a takes on its best value $\arg\min_{d \in Dom(a)} \{{}^{I}LB^{a}_{{}^{I}X^{a}}(d)\}$ when it changes its value or, synonymously, the value with the smallest lower bound in the information unit with its current context.

We now discuss how the MaxPriority scheme estimates the likelihood of future use of an information unit I of agent a: Let $a^1 \ldots a^k$ be the ancestor agents of agent a in the pseudo-tree, ordered in increasing order of their depth in the pseudo-tree. Consider any ancestor agent a^l and assume that estimates of the lower bounds in the information unit of agent a^l with the current context of agent a^l are available to agent a. Let $I(a^l)$ be the index of the lower bound of the value of agent a^l in the context of information unit I in decreasing order of all lower bounds, with one exception: $I(a^l)$ is infinity if the value of agent a^l in the context of information unit I is equal to the value of agent a^l in the current context of agent a. For example, assume that agent a^l can take on four values, namely 0, 1, 2 and 3. Assume that the following estimates of the lower bounds in information unit I of agent a^l with the current context of the agent are available: ${}^{I}LB^{a^l}_{{}^{I}X^{a^l}}(0) = 8$, ${}^{I}LB^{a^l}_{{}^{I}X^{a^l}}(1) = 12$, ${}^{I}LB^{a^l}_{{}^{I}X^{a^l}}(2) = 10$ and ${}^{I}LB^{a^l}_{{}^{I}X^{a^l}}(3) = 6$. If the value of

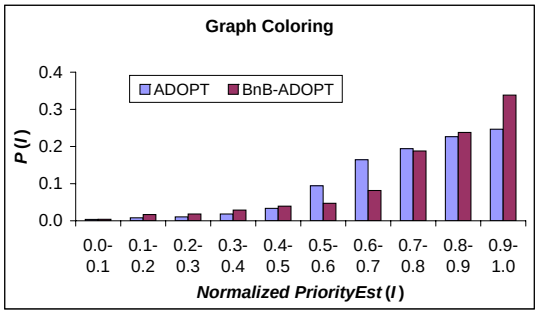


Figure 5.2: Correlation of $\hat{P}(I)$ and $P(I)$

agent a^l in the current context of agent a is 0, then it is most likely that agent a^l still takes on this value. Since each agent takes on the value with the smallest lower bound in the information unit with its current context, if it changes its value, the value that agent a^l currently takes on is in increasing order of likelihood: 1, 2, 3 and 0. The lower bounds are in decreasing order: ${}^{I}LB^{a^l}_{I_{X^{a^l}}}(1) = 12$ (index 0), ${}^{I}LB^{a^l}_{I_{X^{a^l}}}(2) = 10$ (index 1), ${}^{I}LB^{a^l}_{I_{X^{a^l}}}(0) = 8$ (index 2) and ${}^{I}LB^{a^l}_{I_{X^{a^l}}}(3) = 6$ (index 3). Thus, the index of the lower bound of value 3 is 3, the index of the lower bound of value 2 is 1, and the index of the lower bound of value 1 is 0. The index of the lower bound of value 0 is changed to infinity because it is the value of agent a^l in the current context of agent a. Thus, the larger the index is, the more likely it is that agent a^l currently takes on this value. Now consider the tuple $(I(a^1), \ldots, I(a^k))$ for each information unit I cached by agent a. The MaxPriority scheme purges the information unit whose tuple is lexicographically smallest among these information units. More generally, it uses the index of the tuple of an information unit I in the increasing lexicographic order of the tuples of all information units cached by the agent as estimate $\hat{P}(I)$ of $P(I)$. $\hat{P}(I)$ is not meant to approximate $P(I)$ but be proportional to it.

To evaluate how well $\hat{P}(I)$ and $P(I)$ are correlated for ADOPT and BnB-ADOPT, we conduct experiments on the graph coloring problems described in Section 5.4.2. The caches of all agents are sufficiently large to store all information units. For each information unit I of each agent a directly before the agent changes its current context, we divide $\hat{P}(I)$ by the number of cached information units at that point in time minus one to normalize it into the interval from 0 to 1, shown as Normalized PriorityEst(I) in Figure 5.2, and then classify it into one of 10 buckets that cover the interval evenly. We then calculate the frequency for each bucket of the event that the contexts of its information units became the current contexts of their agents after the context switch, shown as $P(I)$ in Figure 5.2. The Pearson's coefficient shows indeed a strong correlation between $\hat{P}(I)$ and $P(I)$ with $\rho > 0.85$.

5.2.3.3 MaxEffort Scheme

The MaxEffort scheme attempts to purge the information unit I with the smallest invested search effort $E(I)$. It estimates the invested search effort of an information unit by using knowledge of the operation of ADOPT and BnB-ADOPT in the form of the lower and upper bounds ${}^{I}LB^{a}_{{}^{I}X^{a}}(d)$ and ${}^{I}UB^{a}_{{}^{I}X^{a}}(d)$, respectively, that every agent a maintains for all values $d \in Dom(a)$ in the information unit I with its current context ${}^{I}X^{a}$. The MaxEffort scheme uses the property that the lower bounds ${}^{I}LB^{a}_{{}^{I}X^{a}}(d)$ are monotonically non-decreasing and the upper bounds ${}^{I}UB^{a}_{{}^{I}X^{a}}(d)$ are monotonically non-increasing if the context ${}^{I}X^{a}$ does not change. Thus, the difference between the lower and upper bounds of an information unit decreases as more search effort is invested in it.

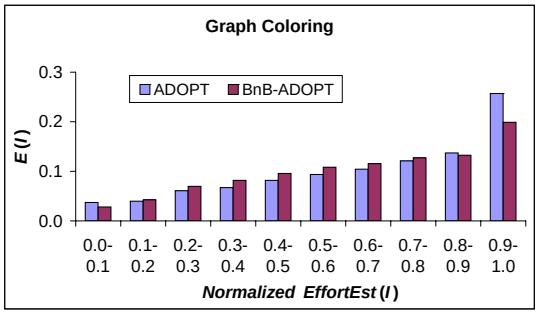


Figure 5.3: Correlation of $\hat{E}(I)$ and $E(I)$

We now discuss how the MaxEffort scheme estimates the invested search effort of an information unit I of agent a: The MaxEffort scheme calculates the average difference $AD(I)$ between the upper bounds ${}^{I}UB^{a}_{I_{X^a}}(d)$ and lower bounds ${}^{I}LB^{a}_{I_{X^a}}(d)$ of the information unit I over all values $d \in Dom(a_i)$:

$$AD(I) := \frac{\sum_{d \in Dom(a)} \{ {}^{I}UB^{a}_{I_{X^a}}(d) - {}^{I}LB^{a}_{I_{X^a}}(d) \}}{|Dom(a)|} \tag{5.2}$$

The MaxEffort scheme purges the information unit whose average difference is largest among all cached information units. More generally, it uses

$$\hat{E}(I) := AD(I') - AD(I) \tag{5.3}$$

as estimate $\hat{E}(I)$ of $E(I)$, where I' is the information unit of agent a_i with the largest average difference. $\hat{E}(I)$ is not meant to approximate $E(I)$ but be proportional to it.

To evaluate how well $\hat{E}(I)$ and $E(I)$ are correlated for ADOPT and BnB-ADOPT, we use the same experimental formulation and setup as described in the context of the MaxPriority scheme. For the information unit I with the current context of each agent a directly after the agent changed its current context, we divide $\hat{E}(I)$ by the largest such estimate over all information units cached by agent a at that point in time to normalize it into the interval from 0 to 1, shown as Normalized EffortEst(I) in Figure 5.3, and then classify it into one of 10 buckets that cover the interval evenly. We then calculate for each bucket the average number of cycles that the contexts of its information units had already been the current contexts of their agents before the context switches and divide each average by the largest average over all buckets to normalize it into the interval from 0 to 1, shown as $E(I)$ in Figure 5.3. The Pearson's coefficient shows indeed a strong correlation between $\hat{E}(I)$ and $E(I)$ with $\rho > 0.85$.

5.2.3.4 MaxUtility Scheme

The MaxUtility scheme attempts to purge the information unit I with the smallest value of $U(I) = P(I) \cdot E(I)$. It uses $\hat{U}(I) := \hat{P}(I) \cdot \hat{E}(I)$ as estimate $\hat{U}(I)$ of $U(I)$. It calculates $\hat{P}(I)$ like the MaxPriority scheme and $\hat{E}(I)$ like the MaxEffort scheme.

5.3 Correctness, Completeness and Complexity

In this section, we prove the correctness and completeness of the caching schemes when used by BnB-ADOPT. Their correctness and completeness when used by ADOPT can be proven in a similar way. We also describe their space and message complexities.

5.3.1 Correctness and Completeness

We follow the assumptions in Section 3.3.1 and assume that each agent a uses the following equations for all values d, all child agents c and all contexts X^a to initialize its bounds.

$$lb^{a,c}_{X^a}(d) := w \cdot h^{a,c}_{X^a}(d) \tag{3.20}$$

$$ub^{a,c}_{X^a}(d) := \infty \tag{3.21}$$

where the weight w is a floating point number that satisfies $1 \leq w < \infty$ and the heuristic values $h^{a,c}_{X^a}(d)$ are floating point numbers that satisfy

$$0 \leq h^{a,c}_{X^a}(d) \leq \gamma^{c}_{X^a \cup (a,d)} \tag{3.22}$$

The only exception is the case where X^a is compatible with the context ${}^{I}X^a$ of a cached information unit I, in which case agent a uses the following equations.

$$lb^{a,c}_{X^a}(d) := {}^{I}lb^{a,c}_{{}^{I}X^a}(d) \tag{5.4}$$

$$ub^{a,c}_{X^a}(d) := {}^{I}ub^{a,c}_{{}^{I}X^a}(d) \tag{5.5}$$

where ${}^{I}lb^{a,c}_{{}^{I}X^a}(d)$ and ${}^{I}lb^{a,c}_{{}^{I}X^a}(d)$ are the bounds in information unit I. Due to this exception, the proof of Lemma 4 is no longer correct. However, we provide a more general proof

of Lemma 4 below that shows that the lemma still holds. All other definitions, lemmas and theorems of Sections 3.3.1 and 4.3.1 continue to hold when BnB-ADOPT uses the caching schemes.

Proof of Lemma 4 by induction on the number of times that agent a changes its context or updates its bounds $lb^{a,c}_{X^a}(d)$ and $ub^{a,c}_{X^a}(d)$ for an arbitrary value d and an arbitrary child agent c after agent a initializes its bounds: The lemma holds after agent a with context X^a initializes its bounds for the first time since

$$\begin{aligned} lb^{a,c}_{X^a}(d) &= w \cdot h^{a,c}_{X^a}(d) && \text{(Eq. 3.20)} \\ &\leq w \cdot \gamma^{c}_{X^a \cup (a,d)} && \text{(Eq. 3.22)} \\ &\leq \infty \\ &= w \cdot ub^{a,c}_{X^a}(d) && \text{(Eq. 3.7)} \end{aligned}$$

(induction basis). Now assume that the lemma holds after agent a changed its context (and either cached or purged its bounds) or updated its bounds a number of times (induction assumption). We show that it then also holds after agent a changes its context or updates its bounds one more time (induction step). There are the following three cases (where we split the operations after receiving a COST message into three parts).

- Case 1: The lemma holds when agent a changes its context from X^a to $\hat{X}^a$ after receiving a VALUE or COST message and the two contexts are compatible since agent a then does not change its bounds and thus

$$
\begin{aligned}
lb^{a,c}_{\hat{X}^a}(d) &= lb^{a,c}_{X^a}(d) && \text{(premise of case)} \\
&\leq w \cdot \gamma^c_{X^a \cup (a,d)} && \text{(induction assumption)} \\
&= w \cdot \gamma^c_{\hat{X}^a \cup (a,d)} && \text{(Lemma 1)} \\
ub^{a,c}_{\hat{X}^a}(d) &= ub^{a,c}_{X^a}(d) && \text{(premise of case)} \\
&\geq \gamma^c_{X^a \cup (a,d)} && \text{(induction assumption)} \\
&= \gamma^c_{\hat{X}^a \cup (a,d)} && \text{(Lemma 1)}
\end{aligned}
$$

after receiving the VALUE or COST message.

- Case 2: The lemma holds when agent a updates its bounds from $lb^{a,c}_{X^a}(d)$ and $ub^{a,c}_{X^a}(d)$ to $\hat{lb}^{a,c}_{X^a}(d)$ and $\hat{ub}^{a,c}_{X^a}(d)$, respectively, after receiving a COST message from some child agent c with bounds $LB^c_{X^c}$ and $UB^c_{X^c}$ and context X^c that is compatible with its context X^a and in which agent a has value d since

$$
\begin{aligned}
\hat{lb}^{a,c}_{X^a}(d) &= \max\{lb^{a,c}_{X^a}(d), LB^c_{X^c}\} && \text{(Eq. 3.8)} \\
&\leq \max\{w \cdot \gamma^c_{X^a \cup (a,d)}, w \cdot \gamma^c_{X^c}\} \\
& && \text{(induction assumption and premise of lemma)} \\
&= \max\{w \cdot \gamma^c_{X^a \cup (a,d)}, w \cdot \gamma^c_{X^a \cup (a,d)}\} && \text{(Lemma 1)} \\
&= w \cdot \gamma^c_{X^a \cup (a,d)}
\end{aligned}
$$

$$\hat{ub}^{a,c}_{X^a}(d) = \min\{ub^{a,c}_{X^a}(d), UB^c_{X^c}\} \quad \text{(Eq. 3.11)}$$

$$\geq \min\{\gamma^c_{X^a \cup (a,d)}, \gamma^c_{X^c}\} \quad \text{(induction assumption and premise of lemma)}$$

$$= \min\{\gamma^c_{X^a \cup (a,d)}, \gamma^c_{X^a \cup (a,d)}\} \quad \text{(Lemma 1)}$$

$$= \gamma^c_{X^a \cup (a,d)}$$

after receiving the COST message.

- Case 3: The lemma holds when agent a changes its context from X^a to $\hat{X}^a$ after receiving a VALUE or COST message and the two contexts are incompatible. There are the following two cases.

  - Case 3a: If context $\hat{X}^a$ is incompatible with the context of any cached information unit, then agent a reinitializes its bounds according to Equations 3.20 and 3.7, and this case is thus identical to the induction basis.

  - Case 3b: If context $\hat{X}^a$ is compatible with the context ${}^I X^a$ of a cached information unit I, then agent a updates its bounds from $lb^{a,c}_{X^a}(d)$ and $ub^{a,c}_{X^a}(d)$ to the bounds ${}^I lb^{a,c}_{I_{X^a}}(d)$ and ${}^I ub^{a,c}_{I_{X^a}}(d)$ in information unit I, respectively. The lemma holds since

$$lb^{a,c}_{\hat{X}^a}(d) = {}^I lb^{a,c}_{I_{X^a}}(d) \quad \text{(Eq. 5.4)}$$

$$\leq w \cdot \gamma^c_{X^a \cup (a,d)} \quad \text{(induction assumption)}$$

$$= w \cdot \gamma^c_{\hat{X}^a \cup (a,d)} \quad \text{(Lemma 1)}$$

$$
\begin{aligned}
ub^{a,c}_{\hat{X}^a}(d) &= {}^{I}ub^{a,c}_{{}^{I}X^a}(d) && \text{(Eq. 5.5)} \\
&\geq \gamma^{c}_{X^a \cup (a,d)} && \text{(induction assumption)} \\
&= \gamma^{c}_{\hat{X}^a \cup (a,d)} && \text{(Lemma 1)}
\end{aligned}
$$

Thus, $lb^{a,c}_{X^a}(d) \leq w \cdot \gamma^{c}_{X^a \cup (a,d)} \leq w \cdot ub^{a,c}_{X^a}(d)$ at all times for all values $d \in Dom(a)$ and all child agents $c \in C(a)$. ■

5.3.2 Complexity

We measure the space complexity of ADOPT and BnB-ADOPT in the number of floating point numbers. The memory size of a context is $O(|A|)$. Every agent a needs to store ${}^{I}lb^{a,c}_{{}^{I}X^a}(d)$ and ${}^{I}ub^{a,c}_{{}^{I}X^a}(d)$ for all child agents $c \in C(a)$ and values $d \in Dom(a)$ for each information unit I. Thus, the memory size of an information unit is $O(|A| \cdot maxDom)$, where $maxDom := \max_{a \in A} |Dom(a)|$ is the maximum domain cardinality over all agents $a \in A$. An agent can cache $O(maxDom^{|A|})$ information units at the same time, namely one for each of the $O(maxDom^{|A|})$ possible contexts. Thus, the maximum cache size and space complexity of every agent in ADOPT and BnB-ADOPT is $O(|A| \cdot maxDom \cdot maxDom^{|A|}) = O(|A| \cdot maxDom^{|A|+1})$.

We measure the message complexity of ADOPT and BnB-ADOPT in the number of floating point numbers as well. The message complexity of ADOPT and BnB-ADOPT is $O(|A|)$ as described in Section 3.3.2. None of the caching schemes increase this message complexity, except for the MaxPriority and MaxUtility schemes. An agent a that uses these caching schemes needs to know, for all of its ancestor agents $p \in SCP(a)$, the indices

of the lower bounds for all of their values in their information units with their current contexts. VALUE messages therefore need to include these indices for the sending agent, and COST messages need to include these indices for all ancestor agents $p \in SCP(a)$ of the sending agent a. There are $O(|A|)$ ancestor agents, each one of which has $O(maxDom)$ values and thus indices. The message complexity thus increases to $O(|A|+maxDom\cdot|A|) = O(maxDom \cdot |A|)$.

5.4 Experimental Evaluation

We now compare ADOPT and BnB-ADOPT with the different caching schemes and the DP2 pre-processing framework described in Section 3.4. We also investigate the effects of caching on ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$.

5.4.1 Metrics

We measure the runtimes in cycles. We do not measure the runtimes in NCCCs because the number of cycles reflects the number of NCCCs; the number of constraint checks and the number of messages sent in each cycle are very similar for both ADOPT and BnB-ADOPT. We also measure the solution costs found by the algorithms. We report normalized solution costs, that is, the solution costs divided by the minimal solution costs. Thus, the normalized solution cost 2.5 refers to a solution cost that is two and a half times larger than the minimal solution cost. Lastly, we also measure the number of unique and repeated contexts explored per agent in ADOPT and BnB-ADOPT to better understand the reason for the speedups. The sum of both numbers is correlated with the runtime of ADOPT and BnB-ADOPT.

We vary the number of information units that the agents can cache using the cache factor metric (Chechetka & Sycara, 2006a). The cache factor of an agent is the ratio of (the number of information units that fit into its cache - 1) and (the number of its possible contexts - 1). If an agent has only one possible context, then it can only cache one information unit independent of the cache factor. Otherwise, agents that can cache only one information unit thus have cache factor zero, and agents that can cache all information units have cache factor one or larger. We vary the cache factor from 0.0 to 1.0. All agents always have the same cache factor.[1]

5.4.2 Problem Types

As described in Section 2.1.5, we run our experiments on four problem types, namely graph coloring, sensor network, meeting scheduling and combinatorial auction problems.

- **Graph coloring problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= vertices to color) to 10 and the density to 2. Each agent always has five possible values (= colors). All costs are randomly generated from 0 to 10,000. We average the experimental results over 50 DCOP problem instances.

- **Sensor network problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= targets to track) to 12. Each agent

[1] We also performed experiments with the cache size metric on graph coloring problems. The cache size of an agent is the number of information units that fit into its cache. The trend for graph coloring problems with the cache factor metric carries over to graph coloring problems with the cache size metric, except that the runtime difference between caching schemes is smaller.

always has five values (= time slots). We average the experimental results over 50 DCOP problem instances.

- **Meeting scheduling problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= meetings to schedule) to 10. Each agent always has five values (= time slots). We average the experimental results over 50 DCOP problem instances.

- **Combinatorial auction problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= bids to consider) to 25. Each agent always has two values (= bid results). We average the experimental results over 50 DCOP problem instances.

5.4.3 Experimental Results

We show the experimental results comparing ADOPT and BnB-ADOPT with the different caching schemes in Section 5.4.3.1 and the experimental results comparing ADOPT$_{WHM}$ with the MaxEffort scheme and BnB-ADOPT$_{WHM}$ with the MaxPriority scheme in Section 5.4.3.2.

5.4.3.1 Caching Schemes

Figure 5.4 shows our experimental results for ADOPT and BnB-ADOPT with the caching schemes on graph coloring problems. Figures in the left and right column show our results for ADOPT and BnB-ADOPT, respectively. We do not report the solution costs found by the algorithms since they all find cost-minimal solutions. We make the following observations:

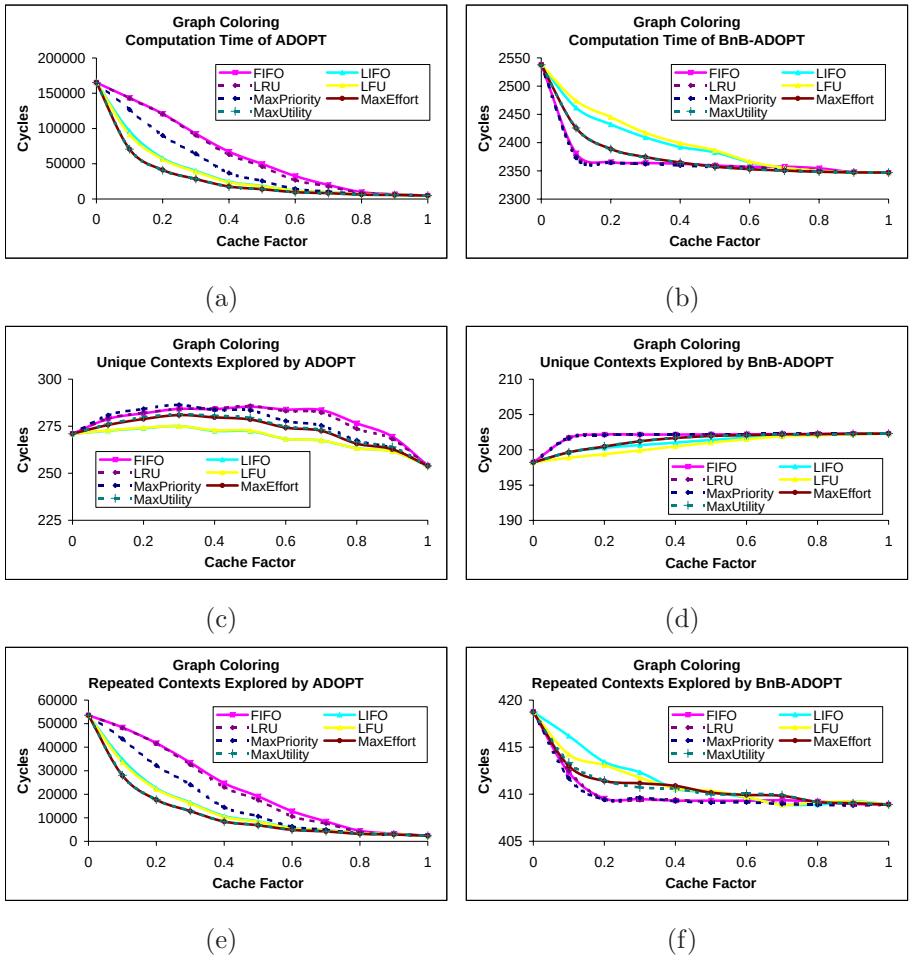


Figure 5.4: Experimental Results Comparing ADOPT and BnB-ADOPT with the Caching Schemes on Graph Coloring Problems

- Figures 5.4(a,b) show that the runtimes of ADOPT and BnB-ADOPT with the caching schemes decrease as the cache factor increases, as expected. The runtime of all caching schemes is identical for each DCOP search algorithm if the cache factor is 0 (because all information units need to be purged) or 1 (because no information units need to be purged). The speedup from caching is much larger for ADOPT than BnB-ADOPT.

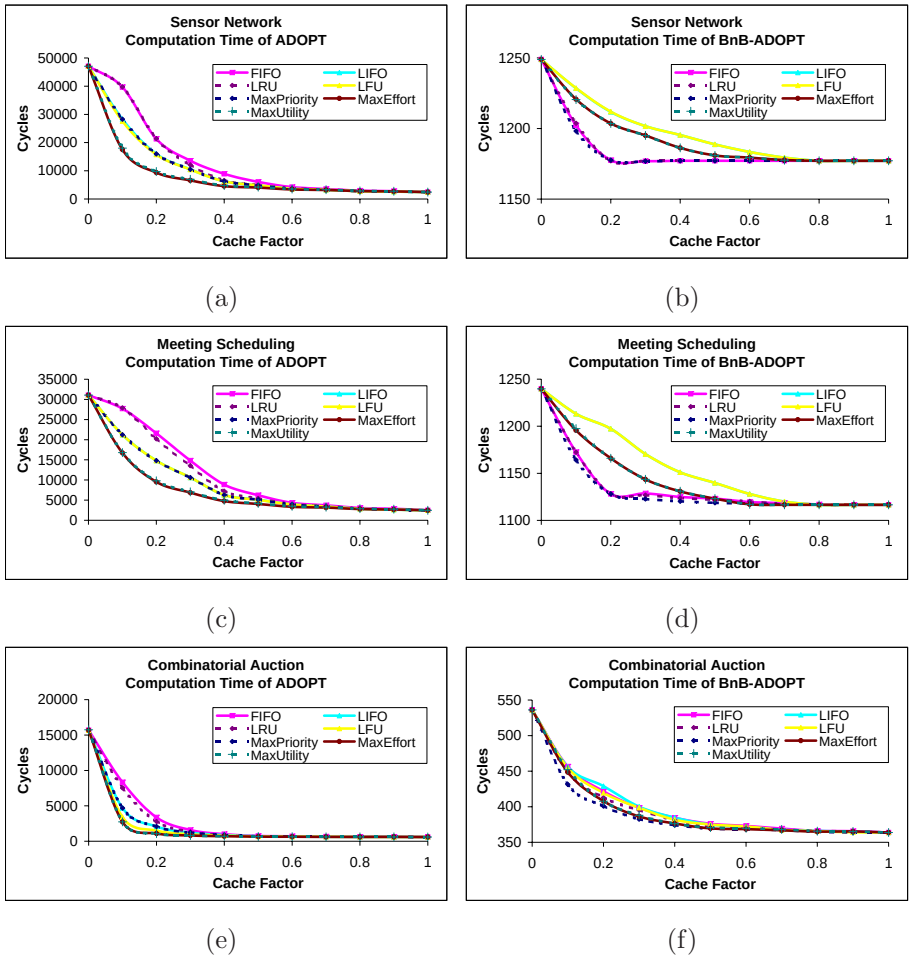


Figure 5.5: Experimental Results Comparing ADOPT and BnB-ADOPT with the Caching Schemes on Sensor Network, Meeting Scheduling and Combinatorial Auction Problems

- Figures 5.4(c-f) give insight into the reason for the decrease in runtime. They show that the number of unique contexts is about the same for all cache factors, indicating that the size of the search space explored by ADOPT and BnB-ADOPT is about the same for all cache factors. The figures also show that the number of repeated contexts per agent decreases as the cache factor increases, indicating that the size of the search space that they need to reconstruct decreases. The size of the search space that ADOPT needs to reconstruct is much larger than the one that

BnB-ADOPT needs to reconstruct since ADOPT needs to reconstruct abandoned partial solutions repeatedly. This difference is also reflected in the runtimes.

- The caching schemes of Category 2 (LIFO, LFU, MaxEffort) result in a smaller runtime than the ones of Category 1 (FIFO, LRU, MaxPriority) for ADOPT and vice versa for BnB-ADOPT. However, the relationships within each category are similar for ADOPT and BnB-ADOPT. For Category 1, the MaxPriority scheme is generally faster than the LRU scheme, which in turn is generally faster than the FIFO scheme. Thus, the MaxPriority scheme is a good caching scheme for Category 1. For Category 2, the MaxEffort scheme is generally faster than the LIFO scheme, which in turn is generally faster than the LFU scheme. Thus, the MaxEffort scheme is a good caching scheme for Category 2. The runtime is about the same for the MaxUtility and MaxEffort schemes. Overall, the MaxEffort and MaxUtility schemes experimentally speed up ADOPT more than the other tested caching schemes, while our MaxPriority scheme experimentally speeds up BnB-ADOPT at least as much as the other tested caching schemes.

  The depth-first branch-and-bound search strategy of BnB-ADOPT does not require its agents to re-expand nodes or, synonymously, re-explore contexts. Due to their distributed execution, agents can explore intermediate contexts for a short period of time when their current context changes. The search effort invested in the information units with the intermediate contexts is thus small. The search effort invested in the other information units is often large but they do not need to be cached because their contexts do not need to be re-explored. On the other hand,

the memory-bounded best-first search strategy of ADOPT does require its agents to re-expand nodes or, synonymously, re-explore contexts. The search effort invested in the information units with these contexts varies. It is therefore important to select a good caching scheme carefully, as the runtimes show. Caching schemes of Category 1 are not well suited for ADOPT, as the runtimes show, since their estimates of the likelihood of future use take into account only that the agents re-explore contexts due to the distributed execution of ADOPT but not that they also re-explore contexts due to its memory-bounded best-first search strategy. Caching schemes of Category 2 are not well suited for BnB-ADOPT, as the runtimes show, since the contexts of the information units with the largest invested search effort do not need need to be re-explored. Ideally, the MaxUtility scheme should result in smaller runtimes than the other caching schemes since an ideal caching scheme minimizes the search effort of a DCOP search algorithm by accurately estimating both the likelihood of future use $P(I)$ of an information unit I and the invested search effort $E(I)$. However, while our estimate $\hat{P}(I)$ is correlated with $P(I)$ and our estimate $\hat{E}(I)$ is correlated with $E(I)$, these correlations are not linear, as shown in Figures 5.2 and 5.3. Thus, the information unit with the smallest value of $\hat{P}(I)$ (which gets purged by the MaxPriority scheme) or $\hat{E}(I)$ (which gets purged by the MaxEffort scheme) is often also the information unit with the smallest value of $P(I)$ or $E(I)$, respectively. However, the information unit with the smallest value of $\hat{U}(I) = \hat{P}(I) \cdot \hat{E}(I)$ (which gets purged by the MaxUtility scheme) is often not the information unit with the smallest value of $U(I) = P(I) \cdot E(I)$.

- Figure 5.5 shows that the trend for graph coloring problems carries over to sensor network, meeting scheduling and combinatorial auction problems as well.

Overall, the MaxEffort and MaxUtility schemes experimentally speed up ADOPT more than the other tested caching schemes, while the MaxPriority scheme experimentally speeds up BnB-ADOPT at least as much as the other tested caching schemes. In general, we also expect our caching schemes to apply to other DCOP search algorithms since they also maintain lower and upper bounds on the solution costs.

5.4.3.2 Caching Schemes with Approximation Mechanisms

Figure 5.6 shows our experimental results for ADOPT$_{WHM}$ with the MaxEffort scheme and BnB-ADOPT$_{WHM}$ with the MaxPriority scheme on graph coloring problems. We performed experiments with ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ because we showed in Section 4.4.3 that the Weighted Heuristics mechanism performs better than the other tested approximation mechanisms when used by BnB-ADOPT.[2] We let ADOPT$_{WHM}$ use the MaxEffort scheme and BnB-ADOPT$_{WHM}$ use the MaxPriority scheme because we showed in Section 5.4.3.1 that the MaxEffort scheme performs better than the other tested schemes when used by ADOPT and the MaxPriority scheme is no worse than all other tested schemes when used by BnB-ADOPT. We make the following observations:

- Figures 5.6(a,b) show that the normalized solution costs of ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ are very similar for every cache factor.

[2]We also performed experiments with ADOPT$_{AEM}$, ADOPT$_{REM}$, BnB-ADOPT$_{AEM}$ and BnB-ADOPT$_{REM}$ on graph coloring problems. The trend for ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ carries over to the other ADOPT and BnB-ADOPT suboptimal variants, respectively, except that the speedups from caching are smaller.

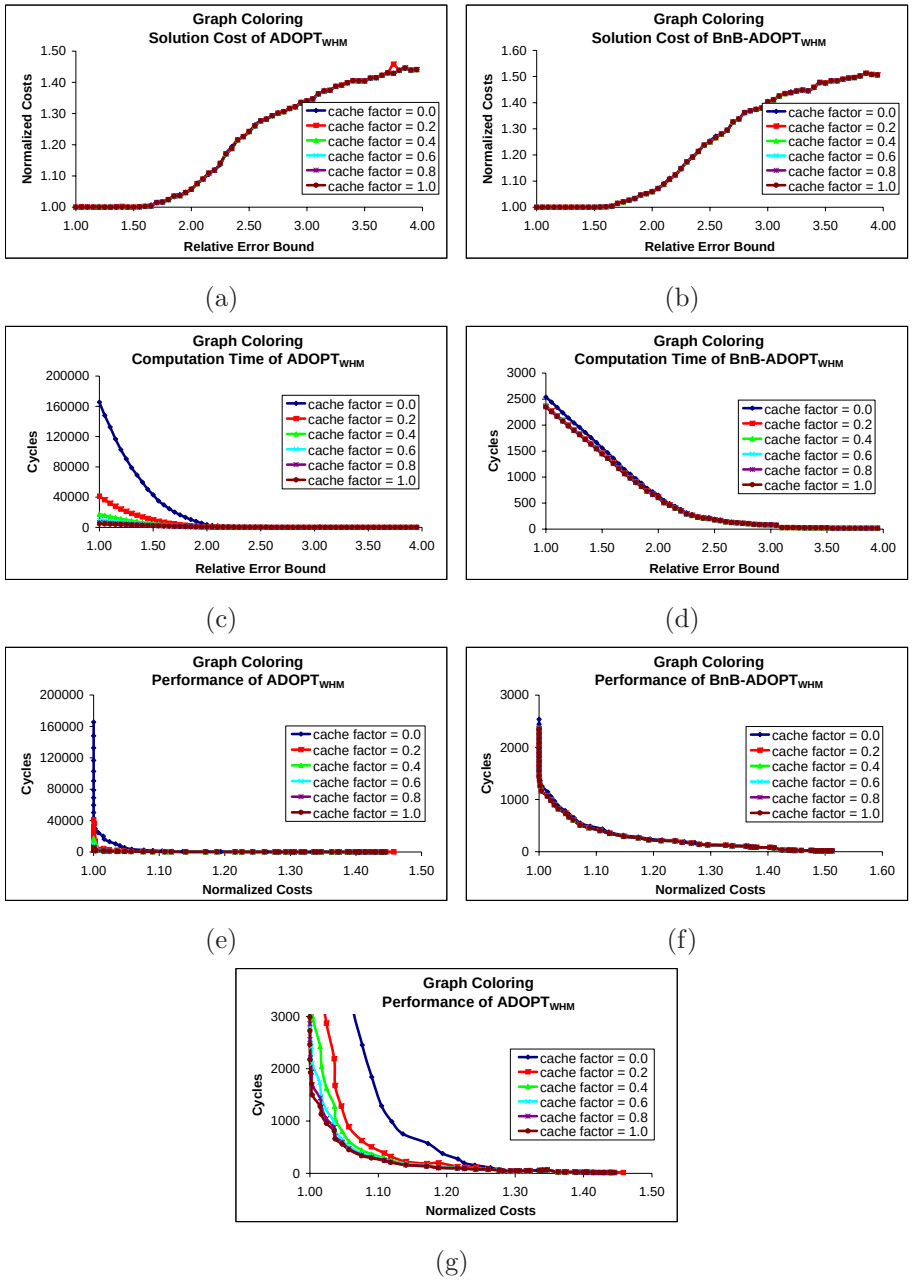


Figure 5.6: Experimental Results of ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ with the Caching Schemes on Graph Coloring Problems

- Figures 5.6(c,d) show that the runtimes of ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ decrease as the cache factor increases. The speedup from caching for ADOPT$_{WHM}$

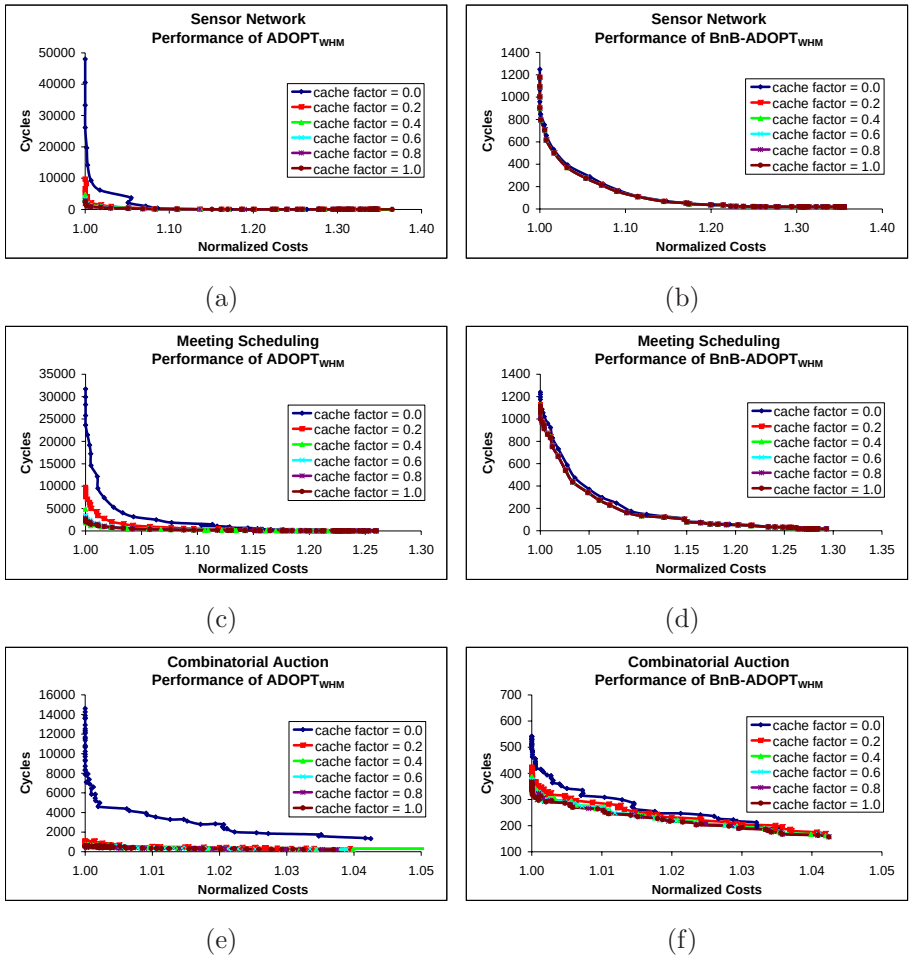


Figure 5.7: Experimental Results of ADOPT$_{WHM}$ and BnB-ADOPT$_{WHM}$ with the Caching Schemes on Sensor Network, Meeting Scheduling and Combinatorial Auction Problems

is much larger than for BnB-ADOPT$_{WHM}$. The reason for this behavior is described in Section 5.4.3.1.

- Figures 5.6(d-f) compare the results for the different cache factors by plotting the runtime needed to achieve a given normalized solution cost. (Figures 5.6(d,f) are identical except for the range of the runtimes. Figure 5.6(f) plots the runtimes from 0 to 3,000 only to better show the difference between the different cache factors.) For ADOPT$_{WHM}$, the runtime needed to achieve a given normalized

solution cost decreases as the cache factor increases. We expect this result since the normalized solution costs of ADOPT$_{WHM}$ are very similar for every cache factor but the runtime of ADOPT$_{WHM}$ decreases as the cache factor increase. For BnB-ADOPT$_{WHM}$, there does not seem to be a significant difference between the different cache factors. We expect this result since the normalized solution costs and runtimes of BnB-ADOPT$_{WHM}$ are very similar for every cache factor.

- Figure 5.7 shows that the trend for graph coloring problems carries over to sensor network, meeting scheduling and combinatorial auction problems as well.

Therefore, one can combine the MaxEffort scheme and the Weighted Heuristics mechanism to further speed up ADOPT. The combination of the MaxPriority scheme and the Weighted Heuristics mechanism do not further speed up BnB-ADOPT. However, we expected this result since caching speeds up BnB-ADOPT only very slightly as seen in Figures 5.4 and 5.5.

5.5 Summary

This chapter framed the caching problem as an optimization problem, where each agent greedily maximizes the sum of the utilities of all cached information units by purging an information unit with the smallest utility from the cache. This chapter also introduced three new DCOP-specific caching schemes called the MaxPriority, MaxEffort and MaxUtility schemes. These three caching schemes use additional knowledge of the operation of ADOPT and BnB-ADOPT in the form of the lower and upper bounds that every agent maintains. The MaxPriority scheme uses the lower bounds to estimate the likelihood

of future use $P(I)$ of an information unit I. The MaxEffort scheme uses the lower and upper bounds to estimate the invested search effort $E(I)$ of an information unit I. The MaxUtility scheme combines the MaxPriority and MaxEffort schemes to estimate the utility $U(I) = P(I) \cdot E(I)$ of an information unit I. Our experimental results show that the MaxEffort and MaxUtility schemes speed up ADOPT more than the currently used generic caching schemes, and the MaxPriority scheme speeds up BnB-ADOPT at least as much as the currently used generic caching schemes. Therefore, these results validate the hypothesis that DCOP-specific caching schemes can reduce the runtime of DCOP search algorithms at least as much as the currently used generic caching schemes.

Chapter 6

Speeding Up via Incremental Search

This chapter models dynamic DCOP problems as sequences of static DCOP problems and introduces an incremental procedure and an incremental pseudo-tree reconstruction algorithm that allow ADOPT and BnB-ADOPT to reuse information from searches of similar static DCOP problems to guide their search to potentially solve the current static DCOP problem faster. The ReuseBounds procedure is an incremental procedure that allows some agents in ADOPT and BnB-ADOPT to reuse the lower and upper bounds from the previous static DCOP problem to solve the current static DCOP problem. The Hybrid Algorithm for Reconstructing Pseudo-trees (HARP) is an incremental pseudo-tree reconstruction algorithm that reuses parts of the pseudo-tree of the previous static DCOP problem to construct the pseudo-tree of the current static DCOP problem. Our experimental results show that ADOPT and BnB-ADOPT with the ReuseBounds procedure and the HARP algorithm terminate faster when they reuse more information. Therefore, these results validate the hypothesis that DCOP search algorithms that reuse information from searches of similar static DCOP problems to guide their search can have runtimes that decrease as they reuse more information.

This chapter is organized as follows: We first describe the motivation for our work in Section 6.1. In Section 6.2, we provide a detailed description of the ReuseBounds procedure and the HARP algorithm. We then prove their correctness and completeness and describe their space and message complexities in Section 6.3 before presenting our experimental results in Section 6.4 and our summary in Section 6.5.

6.1 Motivation

Researchers have developed a class of centralized search algorithms, called incremental search algorithms, that use information from solving problems similar to the given problem to help solve the given problem faster (Koenig et al., 2004b). These algorithms find solutions to a series of problems potentially faster than is possible by solving each problem from scratch, while guaranteeing that they find a cost-minimal solution for each problem (Koenig et al., 2004b). Typically, the runtime of these algorithms decreases as they reuse more information.

Motivated by these results, I hypothesize that DCOP search algorithms that reuse information from searches of similar DCOP static problems to guide their search can have runtimes that decrease as they reuse more information. As described in Section 2.3.4, there are three classes of incremental search algorithms. As a case study, I apply only techniques from the third class of incremental search algorithms to DCOP search algorithms. Specifically, I model dynamic DCOP problems as sequences of static DCOP problems and introduce an incremental procedure, called the ReuseBounds procedure, and an incremental pseudo-tree reconstruction algorithm, called Hybrid Algorithm for

Reconstructing Pseudo-trees (HARP), that allow ADOPT and BnB-ADOPT to reuse information from searches of similar static DCOP problems to guide their search to potentially solve the current static DCOP problem faster. The comparison of the runtimes of ADOPT and BnB-ADOPT with and without the ReuseBounds procedure and the HARP algorithm on dynamic DCOP problems will experimentally assess my hypothesis. This work is non-trivial since agents in ADOPT and BnB-ADOPT need to know the changes to the DCOP problem and identify information that they can reuse while having only local views of the problem or, in other words, knowing only the agents that they share constraints with and the costs of those constraints.

6.2 Incremental Approaches

Since many multi-agent problems change dynamically over time, it is desirable to model such problems with the DCOP model and to design DCOP search algorithms that solve them. We first describe our model of dynamic DCOP problems in Section 6.2.1. We then describe the ReuseBounds procedure in Section 6.2.2 and the Hybrid Algorithm for Reconstructing Pseudo-trees (HARP) in Section 6.2.3.

6.2.1 Dynamic DCOP Problems

There are generally two approaches one can take to model dynamic DCOP problems.

- **Online approach:** One can model dynamic DCOP problems as sequences of static DCOP problems with changes between subsequent static DCOP problems. Each

static DCOP problem in the sequence is a snapshot of the dynamic DCOP problem. Solving a dynamic DCOP problem optimally means finding a cost-minimal solution for each static DCOP problem in the sequence. Therefore, this approach is a myopic approach that does not consider future changes to the dynamic DCOP problem. Researchers have used this approach to model dynamic constraint satisfaction problems (Dechter & Dechter, 1988; Schiex & Verfaillie, 1994) and dynamic path-planning problems (Stentz, 1995; Koenig et al., 2004b). The advantage of this approach is that solving dynamic DCOP problems is no harder than solving static DCOP problems. The disadvantage of this approach is that it can become impossible to solve dynamic DCOP problems if they change sufficiently frequently.

- **Offline approach:** One can model dynamic DCOP problems to take into account all possible future changes to the problem. Solving a dynamic DCOP problem optimally means finding a mapping that maps each possible static DCOP problem to a cost-minimal solution for that static DCOP problem. This model is thus similar to the decentralized Markov decision process (Dec-MDP) model (Pynadath & Tambe, 2002; Goldman & Zilberstein, 2004). The advantage of this approach is that it is able to solve dynamic DCOP problems that change arbitrarily fast since no computation is required at runtime. The disadvantage of this approach is that the dynamic DCOP problem can become intractable when the number of possible changes is large.

Researchers have shown that (centralized) incremental search algorithms can be faster than (centralized) non-incremental search algorithms, such as repeated A*

searches (Koenig et al., 2004b). Thus, DCOP search algorithms that use similar incremental approaches might be able to solve static DCOP problems sufficiently fast to avoid the disadvantages of the online approach. Therefore, we model dynamic DCOP problems using the online approach in this dissertation. Additionally, we assume that dynamic DCOP problems can change in the following five ways: (1) the costs of a constraint can change, (2) a constraint can be removed, (3) a constraint can be added, (4) an agent and its constraints can be removed, and (5) an agent and its constraints can be added.

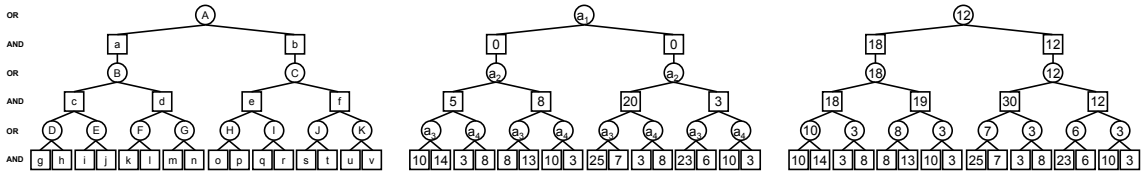
6.2.2 Incremental Procedure

Instead of solving each static DCOP problem from scratch, DCOP search algorithms can reuse information from searches of similar static DCOP problems to guide their search to potentially solve the current static DCOP problem faster. Therefore, we introduce the ReuseBounds procedure, an incremental procedure that ADOPT and BnB-ADOPT can use to identify lower and upper bounds from the previous static DCOP problem that can be reused for the current static DCOP problem. After each change in the dynamic DCOP problem, ADOPT and BnB-ADOPT first reconstruct their pseudo-trees using one of the algorithms described in Section 6.2.3. They then call the ReuseBounds procedure in a pre-processing step before solving the current static DCOP problem.

The principle behind the ReuseBounds procedure is as follows. It identifies potentially affected and unaffected agents in the DCOP problem. The potentially affected agents are those agents whose lower and upper bounds that they maintain from the previous static DCOP problem might no longer be correct bounds for the current static DCOP problem.

The unaffected agents are the other agents. Thus, each potentially affected agent purges all its cached information units, creates a new information unit and caches the current context and the lower and upper bounds initialized to their default values for that context before solving the current static DCOP problem. Unaffected agents reuse the lower and upper bounds in all their cached information units. An agent a is a potentially affected agent iff the gamma costs $\gamma^a_{X^a}$ or $\gamma^a_{X^a}(d)$ for some context X^a and value d of agent a can change between the previous and current static DCOP problems. To be more precise, a gamma cost can change iff one can assign some cost to each constraint that is added, removed or whose costs changed in such a way that the gamma cost indeed changes. If $\gamma^a_{X^a}$ changes, then the lower and upper bounds $LB^a_{X^a}$ and $UB^a_{X^a}$, which were correct bounds $LB^a_{X^a} \leq w \cdot \gamma^a_{X^a} \leq w \cdot UB^a_{X^a}$ for the previous static DCOP problem, might no longer be correct bounds for the current static DCOP problem. Similarly, if $\gamma^a_{X^a}(d)$ changes, then the lower and upper bounds $LB^a_{X^a}(d)$ and $UB^a_{X^a}(d)$, which were correct bounds $LB^a_{X^a}(d) \leq w \cdot \gamma^a_{X^a}(d) \leq w \cdot UB^a_{X^a}(d)$ for the previous static DCOP problem, might no longer be correct bounds for the current static DCOP problem. Unfortunately, an agent cannot directly discern if the gamma costs can change because it needs to solve the current static DCOP problem to do so. However, an agent a can determine if it is a potentially affected agent by checking if it has one or more of the following properties.

- **Property 1:** Agent a shares an added constraint, deleted constraint or constraint with changed constraint costs with another agent. If agent a shares the constraint with a descendant agent, then it is a potentially affected agent (see Property 3). If agent a shares the constraint with an ancestor agent, then the changes in the



(a) Identifiers (b) Delta Costs (c) Gamma Costs

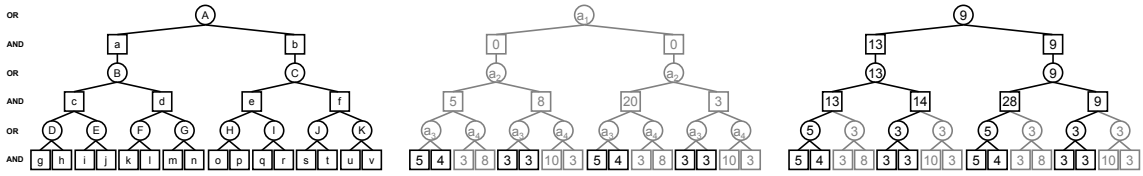
Figure 6.1: Delta and Gamma Costs of the Example DCOP Problem

constraints can change the delta cost $\delta^a_{X^a}(d)$ for some context X^a and value d of agent a, which in turn can change the gamma cost $\gamma^a_{X^a}(d)$ (see Equation 3.1), which in turn can change the gamma cost $\gamma^a_{X^a}$ (see Equation 3.2). Agent a is thus a potentially affected agent.

- **Property 2:** The set of child agents $C(a)$ of agent a changes between the previous and current static DCOP problems. This difference can change the sum $\sum_{c \in C(a)} \gamma^c_{X^a \cup (a,d)}$ for some context X^a and value d, which in turn can change $\gamma^a_{X^a}(d)$ (see Equation 3.1), which in turn can change $\gamma^a_{X^a}$ (see Equation 3.2). Agent a is thus a potentially affected agent.

- **Property 3:** Agent a has a descendant agent c that is a potentially affected agent. If $\gamma^c_{X^c}$ for some context X^c of agent c changes, then the gamma cost $\gamma^a_{X^a}(d)$ can change for some context X^a and value d of agent a (see Equation 3.1), which in turn can change the gamma cost $\gamma^a_{X^a}$ (see Equation 3.2). Agent a is thus a potentially affected agent.

For example, imagine that the constraint between agents a_1 and a_3 is removed from our example DCOP problem of Figure 1.1. Figures 6.1 and 6.2 show the delta and gamma costs for the example DCOP problem before and after the removal of the constraint,



(a) Identifiers (b) Delta Costs (c) Gamma Costs

Figure 6.2: Delta and Gamma Costs after the Removal of the Constraint between Agents a_1 and a_3

respectively, assuming that the ordering of agents in the pseudo-tree remains unchanged. The delta and gamma costs that change are shown in bold. The removal of the constraint changes the delta costs of nodes g, h, k, l, o, p, s, t, which in turn changes the gamma costs of those nodes and nodes D, F, H and J. Thus, agent a_3 is a potentially affected agent since it maintains lower and upper bounds of those nodes (Property 1). The change in the gamma costs of nodes D, F, H and J changes the gamma costs of nodes c, d, e and f, which in turn changes the gamma costs of nodes B and C. Thus, agent a_2 is a potentially affected agent since it maintains lower and upper bounds of those nodes (Property 3). The change in the gamma costs of nodes B and C changes the gamma costs of nodes a and b, which in turn changes the gamma cost of node A. Thus, agent a_1 is a potentially affected agent since it maintains lower and upper bounds of those nodes (Property 3).

Each agent can directly discern if it has Properties 1 or 2. However, it cannot directly discern if it has Property 3 since it knows neither the constraints nor the set of child agents of each one of its descendant agents. Therefore, each agent in ADOPT and BnB-ADOPT runs the ReuseBounds procedure to discern if it has Property 3 after it reconstructs the pseudo-tree for the current static DCOP problem. The idea behind the ReuseBounds procedure is as follows. The root agent starts the propagation of QUERY messages down

```
procedure ReuseBounds()
[01]  sentSTART := amAffected := false;
[02]  loop forever
[03]    if(message queue is not empty)
[04]      pop msg off message queue;
[05]      When Received(msg);
[06]    if(!sentSTART and detected changes in its constraints or C(a))
[07]      sentSTART := true;
[08]      Send(START) to pa(a) if a is not root;
[09]      Send(QUERY) to each c ∈ C(a) if a is root;

procedure When Received(START)
[10]  if(!sentSTART)
[11]    sentSTART := true;
[12]    Send(START) to pa(a) if a is not root;
[13]    Send(QUERY) to each c ∈ C(a) if a is root;

procedure When Received(QUERY)
[14]  Send(QUERY) to each c ∈ C(a);
[15]  if(a is a leaf)
[16]    if(detected changes in its constraints or C(a))
[17]      amAffected := true;
[18]    Send(RESPONSE, a, amAffected) to pa(a);

procedure When Received(RESPONSE, c, amAffected^c)
[19]  if(amAffected^c or detected changes in its constraints or C(a))
[20]    amAffected := true;
[21]  if(received a RESPONSE message from each c' ∈ C(a))
[22]    Send(RESPONSE, a, amAffected) to pa(a) if a is not root;
[23]    if(a is root)
[24]      if(amAffected)
[25]        purge all information units and reinitialize information unit for current context;
[26]      Send(STOP) to each c' ∈ C(a);
[27]      restart DCOP algorithm to solve the current static DCOP problem;

procedure When Received(STOP)
[28]  if(amAffected)
[29]    purge all information units and reinitialize information unit for current context;
[30]  Send(STOP) to each c ∈ C(a);
[31]  restart DCOP algorithm to solve the current static DCOP problem;
```

Figure 6.3: Pseudocode of ReuseBounds

the pseudo-tree. When a leaf agent receives a QUERY message, it identifies itself as a potentially affected agent if it has Properties 1 or 2. It then sends a RESPONSE message to its parent agent with the information whether it is a potentially affected agent. When an agent receives a RESPONSE message, it identifies itself as a potentially affected agent if it has Properties 1 or 2 or its child is a potentially affected agent. After it has received a RESPONSE message from each one of its child agents, it sends a RESPONSE message to its parent agent with the information whether it is a potentially affected agent. Therefore, when the root agent receives a RESPONSE message from each one of its child agents,

each agent must have identified itself as a potentially affected agent if it has Properties 1, 2 or 3.

Figure 6.3 shows the pseudocode. The code is identical for every agent except that the variable a is a "self" variable that points to the agent itself. Each agent by default identifies itself as an unaffected agent by setting its *amAffected* flag to false [Line 1]. The ReuseBounds procedure uses four types of messages:

- **START messages:** START messages propagate up the pseudo-tree until they reach the root agent. The purpose of START messages is to start the procedure and to trigger the root agent to send QUERY messages.

  Each agent with Properties 1 or 2 sends a START message to its parent agent [Lines 6-8]. When an agent receives a START message, if it did not send a START message earlier (on Line 8), then it sends a START message to its parent agent [Lines 10-12].

- **QUERY messages:** QUERY messages propagate down the pseudo-tree until they reach all leaf agents. The purpose of QUERY messages is to trigger leaf agents to send RESPONSE messages.

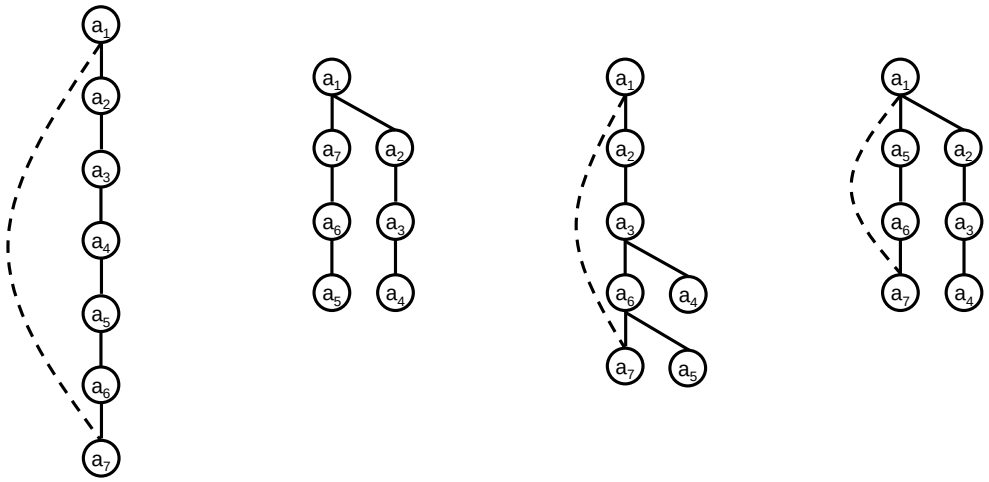
  If the root agent has Properties 1 or 2, it sends a QUERY message to each one of its child agents [Line 9]. When the root agent receives its first START message, if it did not send QUERY messages earlier (on Line 9), then it sends a QUERY message to each one of its child agents [Line 13]. When an agent receives a QUERY message, it sends a QUERY message to each one of its child agents [Line 14].

- **RESPONSE messages:** RESPONSE messages propagate up the pseudo-tree until they reach the root agent. The purpose of RESPONSE messages is to enable the agents to identify themselves as potentially affected agents if they have Properties 1, 2 or 3 and to trigger the root agent to send STOP messages.

  When a leaf agent receives a QUERY message, it identifies itself as a potentially affected agent if it has Properties 1 or 2 by setting its *amAffected* flag to true [Lines 15-17]. It then sends a RESPONSE message to its parent agent with the information whether it is a potentially affected agent [Line 18]. When an agent receives a RESPONSE message from one of its child agents, if it has Properties 1 or 2 or that child agent is a potentially affected agent (and it thus has Property 3), then it identifies itself as a potentially affected agent [Lines 19-20]. After an agent receives a RESPONSE message from each one of its child agents, it sends a RESPONSE message to its parent agent with the information whether it is a potentially affected agent [Lines 21-22].

- **STOP messages:** STOP messages propagate down the pseudo-tree until they reach the leaf agents. The purpose of the STOP messages is to end the procedure.

  After the root agent receives a RESPONSE message from each one of its child agents, every agent must have identified itself as a potentially affected agent if it has Properties 1, 2 or 3. The root agent then purges all its cached information units, creates a new information unit and caches the current context and the lower and upper bounds initialized to their default values if it is a potentially affected agent, sends a STOP message to each one of its child agents and restarts the DCOP



(a) Initial pseudo-tree (b) DFS pseudo-tree (c) Mobed pseudo-tree (d) HARP pseudo-tree

Figure 6.4: DFS, Mobed and HARP Pseudo-trees after Removal of the Constraint between Agents a_4 and a_5

algorithm to solve the current static DCOP problem [Lines 23-27]. When an agent receives a STOP message, it does the same [Lines 28-31].

6.2.3 Incremental Pseudo-tree Reconstruction Algorithms

Before solving the current static DCOP problem, DCOP search algorithms may need to reconstruct their pseudo-trees to reflect the changes from the previous static DCOP problem to the current static DCOP problem. For example, if a new agent is added to the DCOP problem, then a DCOP search algorithm needs to add that agent to its pseudo-tree.

6.2.3.1 Existing Pseudo-tree Reconstruction Algorithms

Finding optimal pseudo-trees is NP-hard (Arnborg, Corneil, & Proskurowski, 1987). As a result, researchers have developed greedy distributed algorithms to reconstruct pseudo-trees.

- **Distributed Depth-First Search (DFS) Algorithm:** The distributed DFS algorithm constructs the pseudo-tree for each static DCOP problem from scratch. It is a distributed algorithm that constructs the pseudo-tree by traversing the constraint graph using DFS with the max-degree heuristic (Collin & Dolev, 1994; Hamadi, Bessière, & Quinqueton, 1998).[1] It chooses the agent with the largest number of constraints and makes it the root agent of the pseudo-tree. The root agent is now the current agent. The DFS algorithm then repeatedly performs the following operations: It chooses the agent with the largest number of constraints among all agents not in the pseudo-tree that are constrained with the current agent and makes this agent a child agent of the current agent. This agent is now the current agent. The algorithm backtracks if no such agent exists and terminates when the pseudo-tree contains all agents. For example, Figure 6.4(a) shows an example initial pseudo-tree, and Figure 6.4(b) shows the pseudo-tree reconstructed by the distributed DFS algorithm after the constraint between agents a_4 and a_5 was removed.

- **Multiagent Organization with Bounded Edit Distance (Mobed) Algorithm:** The Mobed algorithm constructs pseudo-trees with small edit distances

[1]The distributed DFS algorithm can use other heuristics to construct pseudo-trees as well, but we describe the max-degree heuristic because it is commonly used by many DCOP search algorithms including ADOPT and NCBB.

between subsequent static DCOP problems (Sultanik, Lass, & Regli, 2009). The edit distance between two pseudo-trees is the smallest number of parent-child relationships that must be re-assigned, added or deleted in order for both pseudo-trees to become isomorphic. It requires another algorithm such as the distributed DFS algorithm to construct the pseudo-tree of the first static DCOP problem. For each new agent that is added to the DCOP problem, Mobed identifies an insertion point in the pseudo-tree of the previous static DCOP problem and adds the new agent to the pseudo-tree at that insertion point. For each agent that is removed from the DCOP problem, Mobed removes that agent from the pseudo-tree and makes all child agents of the removed agent the child agents of the parent agent of the removed agent. For each constraint that is added or removed, Mobed removes and adds all agents that share that constraint. For example, Figure 6.4(c) shows the pseudo-tree reconstructed by the Mobed algorithm after the constraint between agents a_4 and a_5 was removed.

6.2.3.2 HARP Pseudo-tree Reconstruction Algorithm

The Mobed algorithm constructs pseudo-trees with small edit distances. On the other hand, the distributed DFS algorithm constructs its pseudo-trees from scratch, which can have large edit distances. Therefore, there is usually a larger number of unaffected agents in the Mobed pseudo-trees than in the DFS pseudo-trees. For example, in our example pseudo-trees of Figure 6.4, there is one unaffected agent (= agent a_7) in the Mobed pseudo-tree and there are no unaffected agents in the DFS pseudo-tree. On the other hand, DFS pseudo-trees can have smaller depths than Mobed pseudo-trees. For

procedure HARP()
[01] $sentSTART := amAffected := false$;
[02] $pseudoID := agentID$;
[03] **loop forever**
[04] **if**(message queue is not empty)
[05] pop msg off message queue;
[06] **When Received**(msg);
[07] **if**(!$sentSTART$ and detected changes in its constraints)
[08] $sentSTART := true$;
[09] **Send**(START) to $pa(a)$ if a is not root;
[10] **Send**(QUERY) to each $c \in C(a)$ if a is root;

procedure When Received(START)
[11] **if**(!$sentSTART$)
[12] $sentSTART := true$;
[13] **Send**(START) to $pa(a)$ if a is not root;
[14] **Send**(QUERY) to each $c \in C(a)$ if a is root;

procedure When Received(QUERY)
[15] **Send**(QUERY) to each $c \in C(a)$;
[16] **if**(a is a leaf)
[17] **if**(detected changes in its constraints)
[18] $amAffected := true$;
[19] **Send**(RESPONSE, a, $amAffected$) to $pa(a)$;

procedure When Received(RESPONSE, c, $amAffected^c$)
[20] **if**($amAffected^c$ or detected changes in its constraints)
[21] $amAffected := true$;
[22] **if**(received a RESPONSE message from each $c' \in C(a)$)
[23] **Send**(RESPONSE, a, $amAffected$) to $pa(a)$ if a is not root;
[24] **Send**(PSEUDOID, $pseudoID$, $amAffected$) to each $c' \in C(a)$ if a is root;

procedure When Received(PSEUDOID, $pseudoID^p$, $amAffected^p$)
[25] **if**(!$amAffected$ and $amAffected^p$)
[26] **Send**(CONSTRAINT, a, $SCP(a)$) to each $p \in SCP(a)$;
[27] **else**
[28] **if**(!$amAffected^p$)
[29] $pseudoID := pseudoID^p$;
[30] **Send**(PSEUDOID, $pseudoID$, $amAffected$) to each $c \in C(a)$ if a is not leaf;
[31] **Send**(PSEUDOID-ACK) to $pa(a)$ if a is leaf;

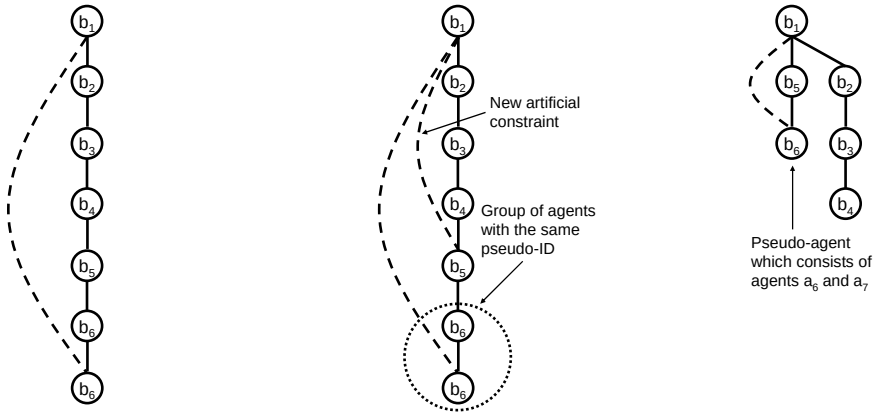
procedure When Received(PSEUDOID-ACK)
[32] **if**(received a PSEUDOID-ACK message from each $c \in C(a)$)
[33] **Send**(PSEUDOID-ACK) to $pa(a)$ if a is not root;
[34] **if**(a is root)
[35] **Send**(STOP) to each $c \in C(a)$;
[36] run distributed DFS algorithm to reconstruct the pseudo-tree for the current static DCOP problem;

procedure When Received(CONSTRAINT, c, SCP^c)
[37] artificially constrain a with each $p \in SCP^c$;
[38] **Send**(CONSTRAINT-ACK) to c;

procedure When Received(CONSTRAINT-ACK)
[39] **if**(received a CONSTRAINT-ACK message from each $p \in SCP(a)$)
[40] **Send**(PSEUDOID, $pseudoID$, $amAffected$) to each $c \in C(a)$ if a is not leaf;
[41] **Send**(PSEUDOID-ACK) to $pa(a)$ if a is leaf;

procedure When Received(STOP)
[42] **Send**(STOP) to each $c \in C(a)$;
[43] run distributed DFS algorithm to reconstruct the pseudo-tree for the current static DCOP problem;

Figure 6.5: Pseudocode of HARP



(a) Assigning pseudo-IDs (b) Artificially constraining agents b_1 and b_5 (c) Constructing DFS pseudo-tree

Figure 6.6: HARP Pseudo-tree Reconstruction Steps

example, in our example pseudo-trees of Figure 6.4, the depth of the DFS pseudo-tree is one smaller than the depth of the Mobed pseudo-tree. Pseudo-trees with smaller depths are desirable since they have a larger number of independent subtrees, or synonymously, a larger number of independent subproblems. Thus, there is a tradeoff between the depths and edit distances of pseudo-trees.

We therefore introduce the Hybrid Algorithm for Reconstructing Pseudo-trees (HARP), an incremental pseudo-tree reconstruction algorithm that reuses parts of the pseudo-tree of the previous static DCOP problem to construct the pseudo-tree of the current static DCOP problem. It combines the principles and strengths of the Mobed and distributed DFS algorithms. Like the Mobed algorithm, HARP aims to preserve the parent-child relationships of unaffected agents in the pseudo-tree, and like the distributed DFS algorithm, HARP reconstructs the part of the pseudo-tree with the affected agents from scratch. Affected agents are agents from the previous static DCOP problem that are *guaranteed* to be potentially affected agents in the current static DCOP

problem regardless of the choice of pseudo-tree reconstruction algorithm. They are (1) the agents that share an added constraint, deleted constraint or constraint with changed costs with another agent (Property 1 in Section 6.2.2) and (2) their ancestor agents in the pseudo-tree of the previous static DCOP problem (Property 3). (Other agents might become potentially affected agents as well but that depends on the choice of pseudo-tree reconstruction algorithm and is thus not guaranteed.)

HARP operates on the pseudo-tree of the previous static DCOP problem to identify the affected agents and calls the distributed DFS algorithm to construct the pseudo-tree of the current static DCOP problem in such a way that non-affected agents are unaffected agents in the current static DCOP problem. Recall that unaffected agents are agents that are not potentially affected agents. For example, imagine that the constraint between agents a_4 and a_5 of Figure 6.4(a) is removed. The affected agents are agents a_4 and a_5 since they have Property 1 and agents a_1, a_2 and a_3 since they have Property 3. Figure 6.6 shows the steps of HARP to reconstruct the pseudo-tree for the current static DCOP problem. HARP assigns a unique pseudo-ID to all agents in the problem with the exception that, all agents in an unaffected subtree are assigned the same pseudo-ID. An unaffected subtree is a subtree that has only unaffected agents. In our example, HARP assigns agents a_1, a_2, a_3, a_4 and a_5 the pseudo-IDs b_1, b_2, b_3, b_4 and b_5, respectively. HARP assigns agents a_6 and a_7 the same pseudo-ID b_6 since they are in an unaffected subtree. Figure 6.6(a) shows the pseudo-tree with the pseudo-ID of each agent. For each group of agents with the same pseudo-ID that is not in a larger group of agents with the same pseudo-ID, HARP artificially constrains all their parent and pseudo-parent agents that are not in the group to each other. In our example, HARP artificially constrains

agents a_1 (with pseudo-ID b_1) and a_5 (with pseudo-ID b_5) since agent a_1 is a pseudo-parent agent of agent a_7 and agent a_5 is a parent agent of agent a_6. Figure 6.6(b) shows the pseudo-tree with the new artificial constraint.

HARP then runs the distributed DFS algorithm except that it treats all agents with the same pseudo-ID as a single pseudo-agent and sets pseudo-agents as child agents of the current agent only if every affected agent that is constrained with the current agent is already in the pseudo-tree. Therefore, pseudo-agents are leaf agents in the new pseudo-tree because every agent that they are constrained with was chosen first due to the artificial constraints. Therefore, non-affected agents do not have Properties 2 and 3 because pseudo-agents are leaf agents and all agents in a pseudo-agent thus have the same descendant agents, which are non-affected agents themselves, in the new pseudo-tree. Non-affected agents do not have Property 1 either because they are affected agents otherwise. Therefore, all non-affected agents are unaffected agents in the current static DCOP problem. Figure 6.6(c) shows the resulting pseudo-tree with pseudo-agents in our example, and Figure 6.4(d) shows the pseudo-tree with actual agents. The HARP pseudo-tree has the same depth as the DFS pseudo-tree and has two unaffected agents (= agents a_6 and a_7) compared to one unaffected agent in the Mobed pseudo-tree and no unaffected agents in the DFS pseudo-tree.

There are two phases in the HARP algorithm. The first phase identifies affected agents and is very similar to how the ReuseBounds procedure identifies potentially affected agents. The second phase assigns pseudo-IDs to agents and imposes artificial constraints between agents before calling the distributed DFS algorithm to reconstruct the pseudo-tree. Figure 6.5 shows the pseudocode of the HARP algorithm. The code is identical for

every agent except that the variable a is a "self" variable that points to the agent itself. Each agent by default identifies itself as an unaffected agent by setting its *amAffected* flag to false [Line 1] and sets its pseudo-ID to its unique agent-ID [Line 2]. The HARP algorithm uses eight types of messages:

- **START messages:** START messages propagate up the pseudo-tree until they reach the root agent. The purpose of START messages is to start the algorithm and to trigger the root agent to send QUERY messages.

  Each agent with Property 1 sends a START message to its parent agent [Lines 7-9]. When an agent receives a START message, if it did not send a START message earlier (on Line 9), then it sends a START message to its parent agent [Lines 11-13].

- **QUERY messages:** QUERY messages propagate down the pseudo-tree until they reach all leaf agents. The purpose of QUERY messages is to trigger leaf agents to send RESPONSE messages.

  If the root agent has Property 1, it sends a QUERY message to each one of its child agents [Line 10]. When the root agent receives its first START message, if it did not send QUERY messages earlier (on Line 10), then it sends a QUERY message to each one of its child agents [Line 14].

- **RESPONSE messages:** RESPONSE messages propagate up the pseudo-tree until they reach the root agent. The purpose of RESPONSE messages is to enable the agents to identify themselves as affected agents if they have Properties 1 or 3 and to trigger the root agent to send PSEUDOID messages.

When a leaf agent receives a QUERY message, it identifies itself as an affected agent if it has Property 1 by setting its *amAffected* flag to true [Lines 16-18]. It then sends a RESPONSE message to its parent agent with the information whether it is an affected agent [Line 19]. When an agent receives a RESPONSE message from one its child agents, if it has Property 1 or that child agent is an affected agent (and it thus has Property 3), then it identifies itself as an affected agent [Lines 20-21]. After an agent receives a RESPONSE message from each one of its child agents, it sends a RESPONSE message to its parent agent with the information whether it is an affected agent [Lines 22-23].

- **PSEUDOID messages:** PSEUDOID messages propagate down the pseudo-tree until they reach all leaf agents. The purpose of the PSEUDOID messages is to enable the agents in an unaffected subtree to assign themselves the same pseudo-ID (= the pseudo-ID of the root of the unaffected subtree).

  After the root agent receives a RESPONSE message from each one of its child agents, every agent must have identified themselves as affected agents if they have Properties 1 or 3. The root agent then sends a PSEUDOID message to each one of its child agents with its pseudo-ID and the information whether it is an affected agent [Line 24]. When an agent receives a PSEUDOID message from its parent agent, there are the following three cases:

  - If it is an affected agent (and it is thus not in an unaffected subtree), then it sends a PSEUDOID message to each one of its child agents with its pseudo-ID and the information that it is an affected agent [Line 30].

- If it is an unaffected agent but its parent agent is an affected agent (and it is thus the root of an unaffected subtree), then it eventually sends a PSEUDOID message to each one of its child agents with its pseudo-ID and the information that it is an unaffected agent [Line 40]. (We describe the reasons in the CONSTRAINT and CONSTRAINT-ACK messages bullet.)
- If both its parent agent and itself are unaffected agents (and it is thus in an unaffected subtree but is not the root of the subtree), then it sets its pseudo-ID to the pseudo-ID in the message and sends a PSEUDOID message to each one of its child agents with its pseudo-ID and the information that it is an unaffected agent [Lines 27-30].

- **CONSTRAINT and CONSTRAINT-ACK messages:** The purpose of CONSTRAINT and CONSTRAINT-ACK messages is to impose artificial constraints between all ancestor agents $p \in SCP(a)$ of the root agent a of an unaffected subtree.

  When an agent receives a PSEUDOID message, if it is an unaffected agent but its parent agent is an affected agent (and it is thus the root of an unaffected subtree), then it sends a CONSTRAINT message, which contains its identity and the set $SCP(a)$, to each ancestor agent $p \in SCP(a)$ [Lines 25-26]. When an agent receives a CONSTRAINT message, it artificially constrains itself with all agents in the set in the message and sends a CONSTRAINT-ACK message to the sender of the CONSTRAINT message [Lines 37-38]. After an agent a receives a CONSTRAINT-ACK message from each ancestor agent $p \in SCP(a)$, it sends a PSEUDOID message

to each one of its child agents if it is not a leaf agent and a PSEUDOID-ACK message to its parent agent otherwise [Lines 39-41].

- **PSEUDOID-ACK messages:** PSEUDOID-ACK messages propagate up the pseudo-tree until they reach the root agent. The purpose of PSEUDOID-ACK messages is to inform the root agent that each agent in an unaffected subtree has assigned itself the pseudo-ID of the root agent of the subtree, to inform the root agent that all the parent and pseudo-parent agents of the agents in each unaffected subtree that are not in the subtree themselves are artificially constrained with each other and to trigger the root agent to send STOP messages.

  When a leaf agent receives a PSEUDOID message, it sends a PSEUDOID-ACK message to its parent agent [Lines 31 and 41]. After an agent receives a PSEUDOID-ACK message from each one of its child agents, it sends a PSEUDOID-ACK message to its parent agent [Lines 32-33].

- **STOP messages:** STOP messages propagate down the pseudo-tree until they reach the leaf agents. The purpose of the STOP messages is to stop this phase of the HARP algorithm and start the distributed DFS algorithm.

  After the root agent receives a PSEUDOID-ACK message from each one of its child agents, each agent in an unaffected subtree must have the same pseudo-ID and all their parent and pseudo-parent agents that are not in the unaffected subtree are artificially constrained with each other. The only exception is the case where the subtree is part of a larger unaffected subtree. The root agent then sends a STOP message to each one of its child agents and runs the distributed DFS algorithm

to reconstruct the pseudo-tree for the current static DCOP problem [Lines 34-36]. When an agent receives a STOP message, it does the same [Lines 42-43].

6.3 Correctness, Completeness and Complexity

In this section, we prove the correctness and completeness of the ReuseBounds procedure and the HARP algorithm when used by BnB-ADOPT. Their correctness and completeness when used by ADOPT can be proven very similarly. We also describe their space and message complexities.

6.3.1 Correctness and Completeness

We provide the proofs of the ReuseBounds procedure in Section 6.3.1.1 and the proofs of the HARP algorithm in Section 6.3.1.2. All the lemmas and theorems in this section hold only if there are changes in the dynamic DCOP problem since the ReuseBounds procedure and the HARP algorithm is used as pre-processing steps before solving each new (but not the first) static DCOP problem. We thus do not explicitly state them later.

6.3.1.1 ReuseBounds Procedure

We follow the assumptions in Sections 3.3.1 and 5.3.1 and assume that each agent a uses the following equations for all values d, all child agents c and all contexts X^a to initialize its bounds.

- If X^a is incompatible with the context of any cached information unit, then

$$lb^{a,c}_{X^a}(d) := w \cdot h^{a,c}_{X^a}(d) \tag{3.20}$$

$$ub^{a,c}_{X^a}(d) := \infty \tag{3.21}$$

where the weight w is a floating point number that satisfies $1 \leq w < \infty$ and the heuristic values $h^{a,c}_{X^a}(d)$ are floating point numbers that satisfy

$$0 \leq h^{a,c}_{X^a}(d) \leq \gamma^c_{X^a \cup (a,d)} \tag{3.22}$$

- If X^a is compatible with the context ${}^I X^a$ of a cached information unit I, then

$$lb^{a,c}_{X^a}(d) := {}^I lb^{a,c}_{I X^a}(d) \tag{5.4}$$

$$ub^{a,c}_{X^a}(d) := {}^I ub^{a,c}_{I X^a}(d) \tag{5.5}$$

where ${}^I lb^{a,c}_{I X^a}(d)$ and ${}^I lb^{a,c}_{I X^a}(d)$ are the bounds in information unit I.

The only exception is the case where agent a is about to start solving a new (but not the first) static DCOP problem, in which case agent a does the following before solving the static DCOP problem.

- If an agent a is a potentially affected agent, then it purges all its cached information units, creates a new information unit and caches the current context and the lower and upper bounds initialized to their default values for that context.

- If agent a is a non-potentially affected agent, then it uses the following equations for all values d, all child agents c, all contexts ${}^{I}X^{a}$ and all information units I to initialize its bounds.

$$ {}^{I}lb^{a,c}_{{}^{I}X^{a}}(d) := {}^{I}\hat{lb}^{a,c}_{{}^{I}X^{a}}(d) \tag{6.1} $$

$$ {}^{I}ub^{a,c}_{{}^{I}X^{a}}(d) := {}^{I}\hat{ub}^{a,c}_{{}^{I}X^{a}}(d) \tag{6.2} $$

  where ${}^{I}\hat{lb}^{a,c}_{{}^{I}X^{a}}(d)$ and ${}^{I}\hat{ub}^{a,c}_{{}^{I}X^{a}}(d)$ are the bounds in information unit I of the previous static DCOP problem.

Due to this exception, the proof of Lemma 4 is no longer correct. However, we provide a more general proof of Lemma 4 at the end of this subsection that shows that the lemma still holds. All other definitions, lemmas and theorems of Sections 3.3.1 and 4.3.1 continue to hold when potentially affected agents in BnB-ADOPT purge their cached information units and the other agents in BnB-ADOPT initialize the bounds in their cached information units with the bounds in the corresponding cached information unit from the previous static DCOP problem before solving the current static DCOP problem. All line numbers in this subsection refer to the line numbers in Figure 6.3.

Definition 3 *An agent a is a potentially affected agent iff the gamma costs $\gamma^a_{X^a}$ and $\gamma^a_{X^a}(d)$ for some context X^a and value d of agent a can change between the previous and current static DCOP problems.*

Lemma 9 *If one or more of the descendant agents of an arbitrary agent a are potentially affected agents, then agent a is a potentially affected agent as well.*

Proof by induction on the depth of the agent in the pseudo-tree: The lemma holds for leaf agents since they do not have descendant agents (induction basis). Now assume that the lemma holds for all agents at depth d in the pseudo-tree (induction assumption). We show that it then also holds for all agents a at depth $d-1$ in the pseudo-tree (induction step). If agent a is a leaf agent, then this case is identical to the induction basis. Otherwise, if one or more of the descendant agents of agent a are potentially affected agents (premise of the lemma), then there must exist at least one child agent $c \in C(a)$ that is a potentially affected agent according to our induction assumption. Thus, the gamma costs $\gamma^c_{X^c}$ for some context X^c of child agent c can change between the previous and current static DCOP problems. Therefore, the gamma costs $\gamma^a_{X^a}(d)$ and $\gamma^a_{X^a}$ can also change between the previous and current static DCOP problems according to Equations 3.1 and 3.2. Agent a is thus a potentially affected agent according to Definition 3. ■

Lemma 10 *If an arbitrary agent a shares an added constraint, deleted constraint or constraint with changed constraint costs with another agent in the current static DCOP problem, then agent a is a potentially affected agent.*

Proof: There are the following two cases.

- Case 1: Agent a shares an added constraint, deleted constraint or constraint with changed constraint costs with an ancestor agent. Thus, the delta costs $\delta^a_{X^a}(d)$ for some context X^a and value d of agent a can change between the previous and current static DCOP problems. (The delta cost $\delta^a_{X^a}(d)$ is the sum of the constraint costs of all constraints that involve both agent a and one of its ancestor agents $p \in SCP(a)$, under the assumption that agent a takes on value d and the agents $p \in SCP(a)$ take on the values in context X^a.) Therefore, the gamma costs $\gamma^a_{X^a}(d)$ and $\gamma^a_{X^a}$ can also change between the previous and current static DCOP problems according to Equations 3.1 and 3.2. Therefore, agent a is a potentially affected agent according to Definition 3.

- Case 2: Agent a shares an added constraint, deleted constraint or constraint with changed constraint costs with a descendant agent c. Thus, the descendant agent c is a potentially affected agent according to Case 1. Therefore, agent a is a potentially affected agent according to Lemma 9. ■

Lemma 11 *If the set of child agents $C(a)$ of an arbitrary agent a changes between the previous and current static DCOP problems, then agent a is a potentially affected agent.*

Proof: The gamma costs $\gamma^a_{X^a}(d)$ for some context X^a and value d of agent a can change between the previous and current static DCOP problems according to Equation 3.1 since the set of child agents $C(a)$ of agent a changes between the previous and current static DCOP problems according to the premise of the lemma. Therefore, the gamma costs

$\gamma^a_{X^a}$ for context X^a of agent a can also change between the previous and current static DCOP problems according to Equation 3.2. Agent a is thus a potentially affected agent according to Definition 3. ■

Lemma 12 *An arbitrary agent a is a potentially affected agent iff it shares an added constraint, deleted constraint or constraint with changed constraint costs with another agent in the current static DCOP problem; its set of child agents $C(a)$ changes between the previous and current static DCOP problems; or one or more of its descendant agents are potentially affected agents.*

Proof: We first prove the lemma for the "if" direction. If agent a shares an added constraint, deleted constraint or constraint with changed constraint costs with another agent in the current static DCOP problem, then agent a is a potentially affected agent according to Lemma 10. If the set of child agents $C(a)$ of agent a changes between the previous and current static DCOP problems, then agent a is a potentially affected agent according to Lemma 11. If one or more of the descendant agents of agent a are potentially affected agents, then agent a is a potentially affected agent according to Lemma 9.

We now prove by contradiction the lemma for the "only if" direction. Let agent a be a potentially affected agent that does not share an added constraint, deleted constraint or constraint with changed constraint costs with another agent in the current static DCOP problem; whose set of child agents remains unchanged between the previous and current static DCOP problems; and that does not have any descendant agents that are potentially affected agents. The gamma costs

$$\gamma^a_{X^a}(d) = \delta^a_{X^a}(d) + \sum_{c \in C(a)} \gamma^c_{X^a \cup (a,d)} \quad \text{(Eq. 3.1)}$$

$$= \delta^a_{X^a}(d) + \sum_{c \in C(a)} \gamma^c_{X^c} \quad \text{(Lemma 1)}$$

$$\gamma^a_{X^a} = \min_{d \in Dom(a)} \{\gamma^a_{X^a}(d)\} \quad \text{(Eq. 3.2)}$$

for context X^a and value d of agent a in the previous static DCOP problem can change to

$$\hat{\gamma}^a_{X^a}(d) = \hat{\delta}^a_{X^a}(d) + \sum_{c \in \hat{C}(a)} \hat{\gamma}^c_{X^a \cup (a,d)} \quad \text{(Eq. 3.1)}$$

$$= \hat{\delta}^a_{X^a}(d) + \sum_{c \in \hat{C}(a)} \hat{\gamma}^c_{X^c} \quad \text{(Lemma 1)}$$

$$\hat{\gamma}^a_{X^a} = \min_{d \in Dom(a)} \{\hat{\gamma}^a_{X^a}(d)\} \quad \text{(Eq. 3.2)}$$

in the current static DCOP problem, respectively, according to Definition 3. However, $\hat{\delta}^a_{X^a}(d) = \delta^a_{X^a}(d)$ for all contexts X^a and values d of agent a (because agent a does not share an added constraint, deleted constraint or constraint with changed constraint costs with another agent in the current static DCOP problem), $\hat{C}(a) = C(a)$ (because the set of child agents of agent a remains unchanged between the previous and current static DCOP problems) and $\hat{\gamma}^c_{X^c} = \gamma^c_{X^c}$ for all contexts X^c of each child agent $c \in C(a)$ according to Definition 3 (because all child agents are non-potentially affected agents). Therefore, the

gamma costs $\gamma^a_{X^a}(d)$ and $\gamma^a_{X^a}$ of agent a cannot change, which implies that agent a is an non-potentially affected agent and thus contradicts our assumption. ■

Lemma 13 *In the ReuseBounds procedure, each agent eventually sends a QUERY message to each one of its child agents unless it is a leaf agent.*

Proof: Let agent a be the agent with the smallest depth among all agents with Properties 1 or 2. (There must be at least one agent with Properties 1 or 2 for each static DCOP problem because the problem is identical to the previous static DCOP problem otherwise.) There are the following two cases.

- Case 1: Agent a is the root agent, in which case it sends a QUERY message to each one of its child agents [Line 9].
- Case 2: Agent a is not the root agent, in which case it sends a START message to its parent agent $pa(a)$ [Line 8], which sends a START message to its parent agent $pa(pa(a))$ when it receives a START message [Line 12], and so on until the root agent receives a START message. The root agent then sends a QUERY message to each one of its child agents [Line 13].

Therefore, the root agent sends QUERY messages in both cases. Every other agent sends a QUERY message to each one of its child agents when it receives a QUERY message [Line 14]. The only exception are leaf agents, which do not have child agents. ■

Lemma 14 *In the ReuseBounds procedure, each agent eventually sends a RESPONSE message to its parent agent unless it is the root agent.*

Proof: Each leaf agent eventually receives a QUERY message from its parent agent according to Lemma 13. When it receives that QUERY message, it sends a RESPONSE message to its parent agent [Line 18]. Every other agent sends a RESPONSE message to its parent agent after it has received a RESPONSE message from each one of its child agents [Lines 21-22]. The only exception is the root agent, which does not have a parent agent. ■

Lemma 15 *In the ReuseBounds procedure, each potentially affected agent eventually sets its* amAffected *flag to true.*

Proof by induction on the depth of the agent in the pseudo-tree: A potentially affected leaf agent a has changes in its constraints (= it shares an added constraint, deleted constraint or constraint with changed constraint costs with another agent) according to Lemma 12 since it does not have any descendant agents. It also eventually receives a QUERY message from its parent agent according to Lemma 13. When it receives that QUERY message, it sets its *amAffected* flag to true [Lines 16-17] (induction basis). Now assume that the lemma holds for all agents at depth d in the pseudo-tree (induction assumption).

We show that it then also holds for all agents at depth $d - 1$ in the pseudo-tree (induction step). If agent a is a leaf agent, then this case is identical to the induction basis. Otherwise, agent a eventually receives a RESPONSE message from each one of its child agents according to Lemma 14. If agent a has changes in its constraints or its set of child agents $C(a)$, then it sets its *amAffected* flag to true when it receives its first RESPONSE message [Lines 19-20]. If agent a has a potentially affected descendant agent,

then it has a potentially affected child agent c according to our induction assumption. Agent a sets its *amAffected* flag to true when it receives the response message from that child agent c [Lines 19-20]. ■

Lemma 16 *In the ReuseBounds procedure, an agent is a potentially affected agent if it sets its* amAffected *flag to true.*

Proof: If an agent a sets its *amAffected* flag to true, then it must have either (a) detected changes in its constraints (= it shares an added constraint, deleted constraint or constraint with changed constraint costs with another agent) or its set of child agents $C(a)$ or (b) received a RESPONSE message from a potentially affected child agent, which is equivalent to having a potentially affected descendant agent [Lines 17 and 20]. Agent a is thus a potentially affected agent according to Lemmas 10, 11 and 9. ■

Lemma 17 *In the ReuseBounds procedure, an agent is a potentially affected agent iff it sets its* amAffected *flag is to true.*

Proof: If an agent is a potentially affected agent, its *amAffected* flag is eventually set to true according to Lemma 15. If an agent sets its *amAffected* flag to true, then it is a potentially affected agent according to Lemma 16. ■

Lemma 18 *The ReuseBounds procedure eventually restarts the DCOP algorithm to solve the current static DCOP problem.*

Proof: The root agent eventually receives a RESPONSE message from each one of its child agents according to Lemma 14. After it has received all those RESPONSE messages,

if it is a potentially affected agent, then it purges all its cached information units, creates a new information unit and caches the current context and the lower and upper bounds initialized to their default values for that context. The root agent then sends a STOP message to each one of its child agents and restarts the DCOP algorithm to solve the current static DCOP problem (even if it is a non-potentially affected agent) [Lines 23-27]. When an agent receives a STOP message from its parent agent, it does the same [Lines 28-31]. Therefore, each agent eventually receives a STOP message. ■

We now provide the proof of Lemma 4, which uses Lemmas 17 and 18.

Proof of Lemma 4 by induction on the number of static DCOP problems solved: The lemma holds for the first static DCOP problem using the proof in Section 5.3.1 since the exception described above does not apply to the first static DCOP problem (induction basis). Now assume that the lemma holds after a number of static DCOP problems have been solved (induction assumption). We show that it then also holds for the current (= new) static DCOP problem by induction on the number of times that agent a changes its context or updates its bounds $lb^{a,c}_{X^a}(d)$ and $ub^{a,c}_{X^a}(d)$ for an arbitrary value d and an arbitrary child agent c after agent a initializes its bounds (induction step): There are the following two cases.

- If agent a is a potentially affected agent, then the lemma holds after agent a with context X^a initializes its bounds for the first time since

$$
\begin{aligned}
lb^{a,c}_{X^a}(d) &= w \cdot h^{a,c}_{X^a}(d) && \text{(Eq. 3.20)} \\
&\leq w \cdot \gamma^{c}_{X^a \cup (a,d)} && \text{(Eq. 3.22)} \\
&\leq \infty \\
&= w \cdot ub^{a,c}_{X^a}(d) && \text{(Eq. 3.7)}
\end{aligned}
$$

- If agent a is a non-potentially affected agent, then the lemma holds after agent a with context X^a initializes its bounds for the first time since

$$
\begin{aligned}
lb^{a,c}_{X^a}(d) &= {}^{I}lb^{a,c}_{{}^{I}X^a}(d) && \text{(Eq. 5.4)} \\
&= {}^{I}\hat{lb}^{a,c}_{{}^{I}X^a}(d) && \text{(Eq. 6.1)} \\
&\leq w \cdot \gamma^{c}_{X^a \cup (a,d)} && \text{(induction assumption and Lemma 17)} \\
ub^{a,c}_{X^a}(d) &= {}^{I}ub^{a,c}_{{}^{I}X^a}(d) && \text{(Eq. 5.5)} \\
&= {}^{I}\hat{ub}^{a,c}_{{}^{I}X^a}(d) && \text{(Eq. 6.2)} \\
&\geq \gamma^{c}_{X^a \cup (a,d)} && \text{(induction assumption and Lemma 17)}
\end{aligned}
$$

  where ${}^{I}lb^{a,c}_{{}^{I}X^a}(d)$ and ${}^{I}lb^{a,c}_{{}^{I}X^a}(d)$ are the bounds in information unit I whose context ${}^{I}X^a$ is compatible with context X^a.

Agent a eventually initializes its bounds for the first time since the ReuseBounds procedure eventually restarts the DCOP algorithm according to Lemma 18. (induction basis).

Now assume that the lemma holds after agent a changed its context (and either cached or purged its bounds) or updated its bounds a number of times (induction assumption). We show that it then also holds after agent a changes its context or updates its bounds one more time (induction step). There are the following three cases (where we split the operations after receiving a COST message into three parts).

- Case 1: The lemma holds when agent a changes its context from X^a to $\hat{X}^a$ after receiving a VALUE or COST message and the two contexts are compatible since agent a then does not change its bounds and thus

$$\begin{aligned}
lb^{a,c}_{\hat{X}^a}(d) &= lb^{a,c}_{X^a}(d) && \text{(premise of case)} \\
&\leq w \cdot \gamma^c_{X^a \cup (a,d)} && \text{(induction assumption)} \\
&= w \cdot \gamma^c_{\hat{X}^a \cup (a,d)} && \text{(Lemma 1)} \\
ub^{a,c}_{\hat{X}^a}(d) &= ub^{a,c}_{X^a}(d) && \text{(premise of case)} \\
&\geq \gamma^c_{X^a \cup (a,d)} && \text{(induction assumption)} \\
&= \gamma^c_{\hat{X}^a \cup (a,d)} && \text{(Lemma 1)}
\end{aligned}$$

  after receiving the VALUE or COST message.

- Case 2: The lemma holds when agent a updates its bounds from $lb^{a,c}_{X^a}(d)$ and $ub^{a,c}_{X^a}(d)$ to $\hat{lb}^{a,c}_{X^a}(d)$ and $\hat{ub}^{a,c}_{X^a}(d)$, respectively, after receiving a COST message from some child agent c with bounds $LB^c_{X^c}$ and $UB^c_{X^c}$ and context X^c that is compatible with its context X^a and in which agent a has value d since

$$
\begin{aligned}
\hat{lb}^{a,c}_{X^a}(d) &= \max\{lb^{a,c}_{X^a}(d), LB^c_{X^c}\} && \text{(Eq. 3.8)} \\
&\leq \max\{w \cdot \gamma^c_{X^a \cup (a,d)}, w \cdot \gamma^c_{X^c}\} \\
& && \text{(induction assumption and premise of lemma)} \\
&= \max\{w \cdot \gamma^c_{X^a \cup (a,d)}, w \cdot \gamma^c_{X^a \cup (a,d)}\} && \text{(Lemma 1)} \\
&= w \cdot \gamma^c_{X^a \cup (a,d)} \\
\hat{ub}^{a,c}_{X^a}(d) &= \min\{ub^{a,c}_{X^a}(d), UB^c_{X^c}\} && \text{(Eq. 3.11)} \\
&\geq \min\{\gamma^c_{X^a \cup (a,d)}, \gamma^c_{X^c}\} && \text{(induction assumption and premise of lemma)} \\
&= \min\{\gamma^c_{X^a \cup (a,d)}, \gamma^c_{X^a \cup (a,d)}\} && \text{(Lemma 1)} \\
&= \gamma^c_{X^a \cup (a,d)}
\end{aligned}
$$

after receiving the COST message.

- Case 3: The lemma holds when agent a changes its context from X^a to $\hat{X}^a$ after receiving a VALUE or COST message and the two contexts are incompatible. There are the following two cases.

  - Case 3a: If context $\hat{X}^a$ is incompatible with the context of any cached information unit, then agent a reinitializes its bounds according to Equations 3.20 and 3.7, and this case is thus identical to the induction basis.

  - Case 3b: If context $\hat{X}^a$ is compatible with the context ${}^I X^a$ of a cached information unit I, then agent a updates its bounds from $lb^{a,c}_{X^a}(d)$ and $ub^{a,c}_{X^a}(d)$ to

the bounds ${}^{I}lb^{a,c}_{I_{X^a}}(d)$ and ${}^{I}ub^{a,c}_{I_{X^a}}(d)$ in information unit I, respectively. The lemma holds since

$$
\begin{aligned}
lb^{a,c}_{\hat{X}^a}(d) &= {}^{I}lb^{a,c}_{I_{X^a}}(d) && \text{(Eq. 5.4)} \\
&\leq w \cdot \gamma^{c}_{X^a \cup (a,d)} && \text{(induction assumption)} \\
&= w \cdot \gamma^{c}_{\hat{X}^a \cup (a,d)} && \text{(Lemma 1)} \\
ub^{a,c}_{\hat{X}^a}(d) &= {}^{I}ub^{a,c}_{I_{X^a}}(d) && \text{(Eq. 5.5)} \\
&\geq \gamma^{c}_{X^a \cup (a,d)} && \text{(induction assumption)} \\
&= \gamma^{c}_{\hat{X}^a \cup (a,d)} && \text{(Lemma 1)}
\end{aligned}
$$

Thus, $lb^{a,c}_{X^a}(d) \leq w \cdot \gamma^{c}_{X^a \cup (a,d)} \leq w \cdot ub^{a,c}_{X^a}(d)$ at all times for all values $d \in Dom(a)$ and all child agents $c \in C(a)$, and the lemma thus holds for the current static DCOP problem. ■

6.3.1.2 HARP Pseudo-tree Reconstruction Algorithm

All line numbers in this subsection refer to the line numbers in Figure 6.5.

Definition 4 *An agent a from the previous static DCOP problem is an affected agent a iff it is guaranteed to be a potentially affected agent in the current static DCOP problem. An agent is an unaffected agent otherwise.*

Definition 5 *A subtree of a pseudo-tree is an unaffected subtree iff it is a strict subtree and it contains only unaffected agents.*

Corollary 2 *If one or more of the descendant agents of an arbitrary agent a are affected agents, then agent a is an affected agent as well.*

Corollary 3 *If an arbitrary agent a shares an added constraint, deleted constraint or constraint with changed constraint costs with another agent in the current static DCOP problem, then agent a is an affected agent.*

Corollary 4 *An arbitrary agent a is an affected agent iff it shares an added constraint, deleted constraint or constraint with changed constraint costs with another agent in the current static DCOP problem or one or more of its descendant agents are affected agents.*

Corollary 5 *In the HARP algorithm, each agent eventually sends a QUERY message to each one of its child agents unless it is a leaf agent.*

Corollary 6 *In the HARP algorithm, each agent eventually sends a RESPONSE message to its parent agent unless it is the root agent.*

Corollary 7 *In the HARP algorithm, each affected agent sets its* amAffected *flag to true.*

Corollary 8 *In the HARP algorithm, an agent is an affected agent if it sets its* amAffected *flag to true.*

Corollary 9 *In the HARP algorithm, an agent is an affected agent iff it sets its* amAffected *flag is to true.*

The proofs of Corollaries 2, 3, 4, 5, 6, 7, 8, and 9 follow the proofs of Lemmas 9, 10, 12, 13, 14, 15, 16 and 17, respectively, since agents send START, QUERY and RESPONSE messages and set their respective *amAffected* flags in the same way in both the HARP algorithm and the ReuseBounds procedure.

Lemma 19 *In the HARP algorithm, each agent eventually sends a PSEUDOID message to each one of its child agents unless it is a leaf agent.*

Proof: The root agent eventually receives a RESPONSE message from each one of its child agents according to Corollary 6. After it has received all those RESPONSE messages, it sends a PSEUDOID message to each one of its child agents [Line 24]. When a child agent c receives a PSEUDOID message, there are the following two cases.

- Case 1: If agent c is an affected agent or its parent agent is an unaffected agent, then it sends a PSEUDOID message to each one of its child agents [Line 30].
- Case 2: Otherwise, agent c sends a CONSTRAINT message to each ancestor agent $p \in SCP(c)$ [Line 26], which eventually replies with a CONSTRAINT-ACK message to agent c [Line 38]. Agent c sends a PSEUDOID message to each one of its child agents after it has received a CONSTRAINT-ACK message from each ancestor agent $p \in SCP(c)$ [Lines 39-40].

Therefore, agent c sends PSEUDOID messages in both cases. Every other agent sends a PSEUDOID message to each one of its child agents according to the conditions in the above two cases when it receives a PSEUDOID message. The only exception are the leaf agents, which do not have child agents. ■

Lemma 20 *In the HARP algorithm, each agent eventually sends a PSEUDOID-ACK message to its parent agent unless it is the root agent.*

Proof: Each leaf agent a eventually receives a PSEUDOID message from its parent agent according to Lemma 19. When it receives that PSEUDOID message, there are the following two cases.

- Case 1: If agent a is an affected agent or its parent agent is an unaffected agent, then it sends a PSEUDOID-ACK message to its parent agent [Line 31].
- Case 2: Otherwise, agent a sends a CONSTRAINT message to each ancestor agent $p \in SCP(a)$ [Line 26], which eventually replies with a CONSTRAINT-ACK message to agent a [Line 38]. Agent a sends a PSEUDOID-ACK message to its parent agent after it has received a CONSTRAINT-ACK message from each ancestor agent $p \in SCP(a)$ [Lines 39 and 41].

Therefore, agent a sends PSEUDOID-ACK messages in both cases. Every other agent sends a PSEUDOID-ACK message to its parent agent after it has received a PSEUDOID-ACK message from each one of its child agents [Lines 32-33]. The only exception is the root agent, which does not have a parent agent. ■

Lemma 21 *In the HARP algorithm, if an unaffected subtree S of a pseudo-tree is not part of a larger unaffected subtree, then the root agent of subtree S has a parent agent that is an affected agent.*

Proof by contradiction: Assume that the lemma does not hold and choose an agent a that is the root of an arbitrary unaffected subtree S that is not part of a larger unaffected subtree. The agent then cannot be the root of the pseudo-tree and thus has a parent agent. There are the following two cases.

- Case 1: Agent a does not have any sibling agents in the pseudo-tree or all the sibling agents of agent a are unaffected agents. Since the parent agent of agent a is an unaffected agent according to our assumption, the subtree S must be part of a larger unaffected subtree that is rooted at the parent agent of agent a, which contradicts our assumption.

- Case 2: Agent a has a sibling agent a' that is an affected agent. The parent agent of agent a is an unaffected agent according to our assumption, but it has a child agent a' that is an affected agent, which contradicts Corollary 2. ■

Lemma 22 *If an unaffected subtree S of a pseudo-tree is not part of a larger unaffected subtree, then the HARP algorithm eventually artificially constrains all ancestor agents $p \in SCP(a)$ of the root agent a of subtree S to each other.*

Proof: The root agent a of subtree S is an unaffected agent according to the premise of the lemma and Definition 5, and it has a parent agent that is an affected agent according to Lemma 21. Furthermore, the parent agent eventually sets its *amAffected* flag to true according to Corollary 7 and does not set it to false later, and agent a eventually receives a PSEUDOID message from its parent agent according to Lemma 19. When it receives that PSEUDOID message, it sends a CONSTRAINT message to each ancestor agent $p \in SCP(a)$ [Line 26]. Each ancestor agent artificially constraints itself with all agents in $SCP(a)$ when it receives a CONSTRAINT message from agent a [Line 37]. ■

Lemma 23 *In the HARP algorithm, each agent a sends at most $|C(a)|$ QUERY messages, one to each one of its child agents.*

Proof by induction on the depth of the agent in the pseudo-tree: The lemma holds for the root agent because it sets its *sentSTART* flag to false at the start of the HARP algorithm [Line 1] and sets it to true [Lines 8 and 12] before it sends a QUERY message to each one of its child agents [Lines 10 and 14] (induction basis). Now assume that the lemma holds for all agents at depth d in the pseudo-tree (induction assumption). We show that it then also holds for all agents at depth $d+1$ in the pseudo-tree (induction step). Each agent sends a QUERY message to each one of its child agents only when it receives a QUERY message [Line 15]. Therefore, the lemma holds since each agent receives at most one QUERY message from its parent agent according to our induction assumption. ■

Lemma 24 *In the HARP algorithm, each agent sends at most one RESPONSE message.*

Proof by induction on the depth of the agent in the pseudo-tree: Each leaf agent sends a RESPONSE message to its parent agent only when it receives a QUERY message [Line 19]. Therefore, the lemma holds for the leaf agents since each leaf agent receives at most one QUERY message from its parent agent according to Lemma 23 (induction basis). Now assume that the lemma holds for all agents at depth d in the pseudo-tree (induction assumption). We show that it then also holds for all agents a at depth $d-1$ in the pseudo-tree (induction step). If agent a is a leaf agent, then this case is identical to the induction basis. Otherwise, agent a sends a RESPONSE message to its parent agent only after it has received a RESPONSE message from each one of its child agents [Lines 22-23]. Therefore, the lemma holds since each agent receives at most one RESPONSE message from each one of its child agents according to our induction assumption. ■

Lemma 25 *In the HARP algorithm, if an agent a receives at most one PSEUDOID message from its parent agent, then it sends at most* $|SCP(a)|$ *CONSTRAINT messages, one to each ancestor agent* $p \in SCP(a)$*, and receives at most* $|SCP(a)|$ *CONSTRAINT-ACK messages, one from each ancestor agent* $p \in SCP(a)$*.*

Proof: Each agent a sends a CONSTRAINT message to each ancestor agent $p \in SCP(a)$ only when it receives a PSEUDOID message [Line 26], and each agent only sends a CONSTRAINT-ACK message when it receives a CONSTRAINT message [Line 38]. Therefore, the lemma holds since agent a receives at most one PSEUDOID message from its parent agent according to the premise of the lemma. ■

Lemma 26 *In the HARP algorithm, each agent a sends at most* $|C(a)|$ *PSEUDOID messages, one to each one of its child agents.*

Proof by induction on the depth of the agent in the pseudo-tree: The root agent sends a PSEUDOID message to each one of its child agents only after it has received a RESPONSE message from each one of its child agents [Line 24]. Therefore, the lemma holds for the root agent since it receives at most one RESPONSE message from each one of its child agents according to Lemma 24 (induction basis). Now assume that the lemma holds for all agents at depth d in the pseudo-tree (induction assumption). We show that it then also holds for all agents at depth $d+1$ in the pseudo-tree (induction step). There are the following two cases.

- Case 1: If an agent a is an affected agent or its parent agent is an unaffected agent, then agent a sends a PSEUDOID message to each one of its child agents only when

it receives a PSEUDOID message [Line 30]. Therefore the lemma holds for this case since agent a receives at most one PSEUDOID message from its parent agent according to our induction assumption.

- Case 2: Otherwise, agent a sends a PSEUDOID message to each one of its child agents only after it has received a CONSTRAINT-ACK message from each ancestor agent $p \in SCP(a)$ [Line 41]. Therefore, the lemma holds for this case since agent a receives at most one CONSTRAINT-ACK message from each ancestor agent $p \in SCP(a)$ according to Lemma 25. ■

Lemma 27 *In the HARP algorithm, each agent can set its pseudo-ID only before it sends its PSEUDOID messages and can set it at most twice, once on Line 2 and once on Line 29.*

Proof: Each agent sets its pseudo-ID to its unique agent-ID at the start of the HARP algorithm [Line 2] and sets it again only before it sends a PSEUDOID message to each one of its child agents if both its parent agent and itself are unaffected agents [Lines 27-29]. Therefore, the lemma holds because each agent sends at most one PSEUDOID message to each one of its child agents according to Lemma 26. ■

Lemma 28 *If an unaffected subtree S of a pseudo-tree is not part of a larger unaffected subtree, then the agents in subtree S have the same pseudo-ID as the root agent of the subtree.*

Proof by induction on the depth of the agent in subtree S: The lemma holds for the root agent of the subtree since it sets its pseudo-ID to its agent-ID at the start of the HARP

algorithm [Line 2] and does not set it again later. It does not set it on Line 29 because its parent agent is an affected agent according to Lemma 21 (induction basis). Now assume that the lemma holds for all agents at depth d in the subtree (induction assumption).

We show that it then also holds for all agents at depth $d+1$ in the subtree (induction step). When agent a receives a PSEUDOID message from its parent agent, it sets its pseudo-ID to the pseudo-ID of its parent agent if its parent agent and itself are unaffected agents [Lines 27-29]. Therefore, agent a sets its pseudo-ID to the pseudo-ID of the root agent of subtree S since agent a eventually receives a PSEUDOID message according to Lemma 19, the pseudo-ID of its parent agent is the pseudo-ID of the root agent according to our induction assumption and both its parent agent and itself are unaffected agents according to Definition 5. Furthermore, agent a cannot set its pseudo-ID later since it sets its pseudo-ID at most twice according to Lemma 27.

Lemma 29 *HARP eventually restarts the distributed DFS algorithm starts to reconstruct the pseudo-tree for the current static DCOP problem.*

Proof: The root agent eventually receives a PSEUDOID-ACK message from each one of its child agents according to Lemma 20. After it has received all those PSEUDOID-ACK messages, it sends a STOP message to each one of its child agents and starts the distributed DFS algorithm to reconstruct the pseudo-tree for the current static DCOP problem [Lines 34-36]. When an agent receives a STOP message from its parent agent, it does the same [Lines 42-43]. Therefore, each agent eventually receives a STOP message. ■

Theorem 7 *The HARP algorithm is correct and complete.*

Proof: In the HARP algorithm, if an unaffected subtree S of a pseudo-tree is not part of a larger unaffected subtree, then all agents in subtree S eventually have the same pseudo-ID and all their parent and pseudo-parent agents that are not in subtree S are eventually artificially constrained with each other according to Lemmas 28 and 22. These conditions are sufficient for the distributed DFS pseudo-tree reconstruction algorithm to construct a valid pseudo-tree. HARP eventually starts the DFS algorithm to reconstruct the pseudo-tree according to Lemma 29. Therefore, the HARP algorithm is correct and complete. ■

6.3.2 Complexity

We measure the space complexity of the ReuseBounds procedure and the HARP algorithm in the number of floating point numbers. In addition to the cached information units, every agent needs to store all its constraints and their costs to check if it has Property 1, it needs to store the identity of all its child agents to check if it has Property 2, and it needs to store one variable to check if it has Property 3. (The three properties are described in Section 6.2.2.) There are $O(|A|)$ constraints for each agent, and each (binary) constraint have $O(maxDom^2)$ number of costs, where $maxDom := \max_{a' \in A} |Dom(a')|$ is the maximum domain cardinality over all agents $a' \in A$. Thus, the space complexity of every agent a in the ReuseBounds procedure is $O(|A| \cdot maxDom^{|A|+1} + |A| \cdot maxDom^2 + |C(a)| + 1) = O(|A| \cdot maxDom^{|A|+1} + |A| \cdot maxDom^2 + |A| + 1) = O(|A| \cdot maxDom^{|A|+1})$. The first term is the space complexity to cache all information units, and the second,

third and fourth terms are the space complexities needed to check for Properties 1, 2 and 3, respectively. The space complexity of every agent in the HARP algorithm is $O(|A| \cdot maxDom^{|A|+1} + |A| \cdot maxDom^2 + 1) = O(|A| \cdot maxDom^{|A|+1})$ since it does not need to check for Property 2.

We measure the message complexity in the number of floating point numbers as well. The complexity of START, QUERY, RESPONSE, PSEUDOID, PSEUDOID-ACK, CONSTRAINT-ACK and STOP messages is $O(1)$ since START, QUERY, PSEUDOID-ACK, CONSTRAINT-ACK and STOP messages contain one floating point number and RESPONSE and PSEUDOID messages contain three floating point numbers. The complexity of CONSTRAINT messages is $O(|A|)$ since it contains two floating point numbers and the set of ancestor agents $SCP(a)$ of the sending agent a. Therefore, the message complexity of the ReuseBounds procedure and the HARP algorithm is $O(1)$ and $O(|A|)$, respectively.

6.4 Experimental Evaluation

We now compare ADOPT and BnB-ADOPT with and without the ReuseBounds procedure and the DP2 pre-processing framework described in Section 3.4 with the distributed DFS, Mobed and HARP pseudo-tree reconstruction algorithms described in Section 6.2.3.

6.4.1 Metrics

We measure the runtimes in cycles. We do not measure the runtimes in NCCCs because the number of cycles reflects the number of NCCCs; the number of constraint checks and the number of messages sent in each cycle are very similar for both ADOPT and

BnB-ADOPT. We do not measure the solution costs found by the algorithms since they all find cost-minimal solutions. We vary the number of information units that the agents can cache using the cache factor metric from 0.0 to 1.0.

6.4.2 Problem Types

As described in Section 2.1.5, we run our experiments on four problem types, namely graph coloring, sensor network, meeting scheduling and combinatorial auction problems.

- **Graph coloring problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= vertices to color) to 10 and the density to 2. Each agent always has five possible values (= colors). All costs are randomly generated from 0 to 10,000.

- **Sensor network problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= targets to track) to 12. Each agent always has five values (= time slots).

- **Meeting scheduling problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= meetings to schedule) to 10. Each agent always has five values (= time slots).

- **Combinatorial auction problems:** We use the same experimental setup as Section 3.4.2 except that we set the number of agents (= bids to consider) to 25. Each agent always has two values (= bid results).

We consider the following five types of changes.

- **Type 1: Changes in the Costs of a Constraint:** We randomly choose a constraint and randomly change its costs.
- **Type 2: Removal of a Constraint:** We randomly choose a constraint and remove it.
- **Type 3: Addition of a Constraint:** We randomly choose two agents that are not sharing a constraint and add a constraint with randomly generated costs between them.
- **Type 4: Removal of an Agent and its Constraints:** We randomly choose an agent and remove it and its constraints.
- **Type 5: Addition of an Agent and its Constraints:** We add a new agent and two new constraints with randomly generated costs. Each new constraint is between the new agent and a randomly chosen agent.

We average the experimental results over 50 dynamic DCOP problem instances for each change. We change each dynamic DCOP problem in the following three ways.

- **Change 1:** We change the dynamic DCOP problem with one change of each type in the order described above. Therefore, each dynamic DCOP problem consists of six static DCOP problems – the first is the initial randomly generated problem, the second problem is the first problem with one Type 1 change, the third problem is the second problem with one Type 2 change and so on.

- **Change 2:** We change the dynamic DCOP problem with one change of each type in a random order. Therefore, each dynamic DCOP problem consists of six static DCOP problems – the first is the initial randomly generated problem, the second problem is the first problem with one randomly chosen type change, the third problem is the second problem with one different randomly chosen type change and so on.

- **Change 3:** We change the dynamic DCOP problem with one Type 1 change five times. Therefore, each dynamic DCOP problem consists of six static DCOP problems – the first is the initial randomly generated problem, the second problem is the first problem with one Type 1 change, the third problem is the second problem with another Type 1 change and so on.

We average the experimental results over 50 dynamic DCOP problem instances for each change.

6.4.3 Experimental Results

Figure 6.7 shows our experimental results for ADOPT and BnB-ADOPT with the Reuse-Bounds procedure and the pseudo-tree reconstruction algorithms on graph coloring problems with Change 1. We make the following observations:

- Figures 6.7(a,b) show that the runtimes of ADOPT and BnB-ADOPT decrease as the cache factor increases. As described in Section 5.4.3, the reason for this behavior is that they need to re-expand fewer nodes when they cache more information.

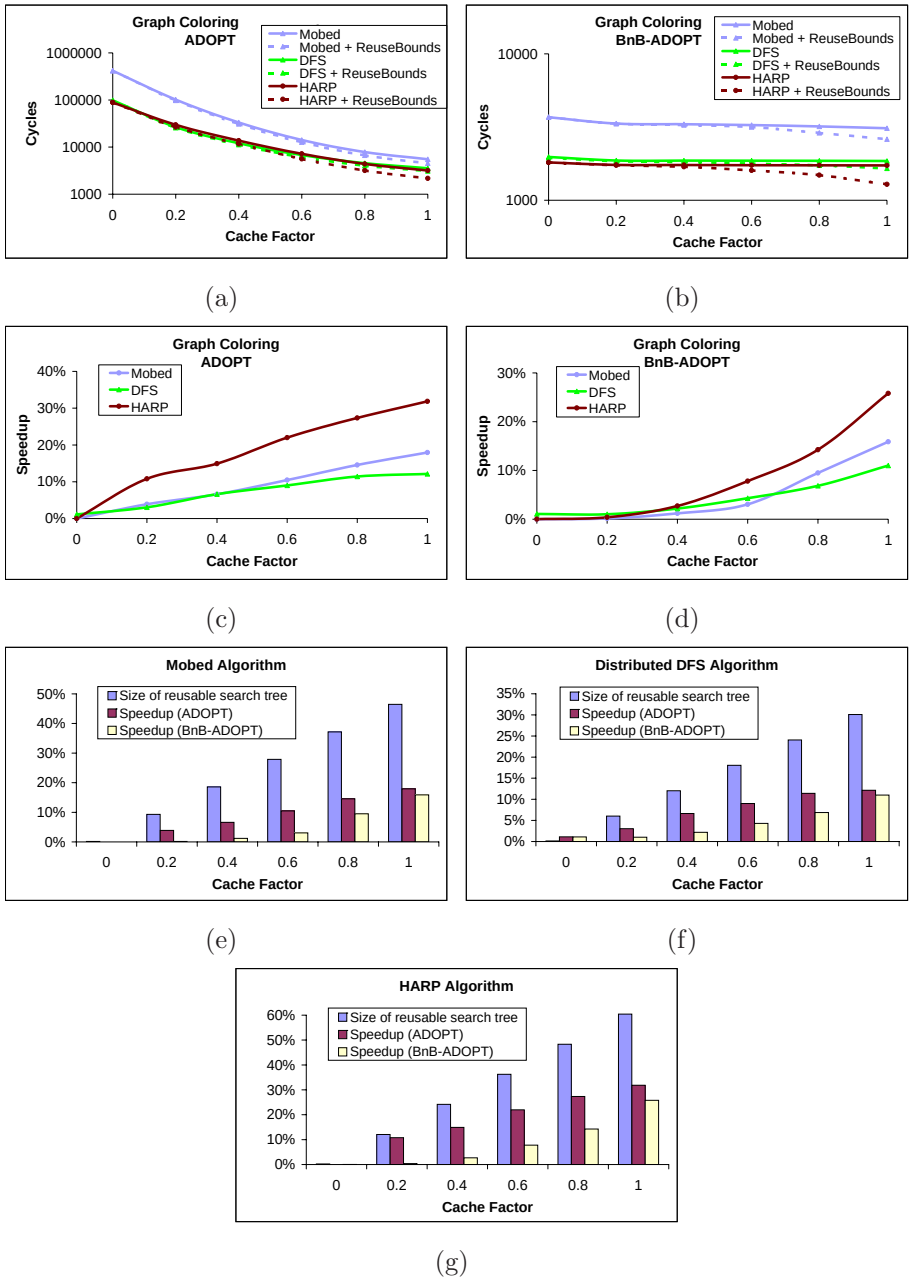


Figure 6.7: Experimental Results Comparing ADOPT and BnB-ADOPT with the Reuse-Bounds Procedure and the Pseudo-tree Reconstruction Algorithms on Graph Coloring Problems with Change 1

- The runtimes of ADOPT and BnB-ADOPT are smaller with the ReuseBounds procedure than without the procedure. Figures 6.7(c,d) show the speedup gained

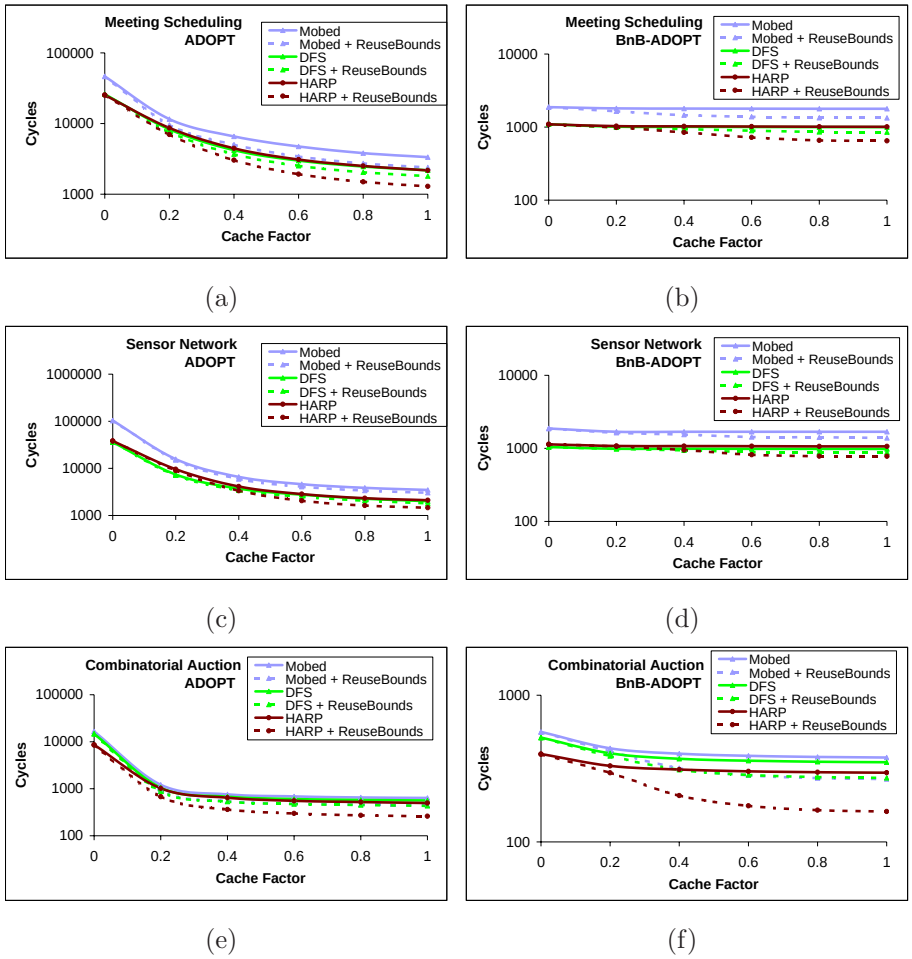


Figure 6.8: Experimental Results Comparing ADOPT and BnB-ADOPT with the Reuse-Bounds Procedure and the Pseudo-tree Reconstruction Algorithms on Sensor Network, Meeting Scheduling and Combinatorial Auction Problems with Change 1 (1)

with the ReuseBounds procedure. We calculate the speedup by taking the difference in the runtimes with and without the ReuseBounds procedure and normalizing it by the runtime without the ReuseBounds procedure. The figures show that the speedup of ADOPT and BnB-ADOPT increases as the cache factor increases for all three pseudo-tree reconstruction algorithms. The reason for this behavior is that the unaffected agents can cache and reuse more lower and upper bounds from the previous static DCOP problems as the cache factor increases.

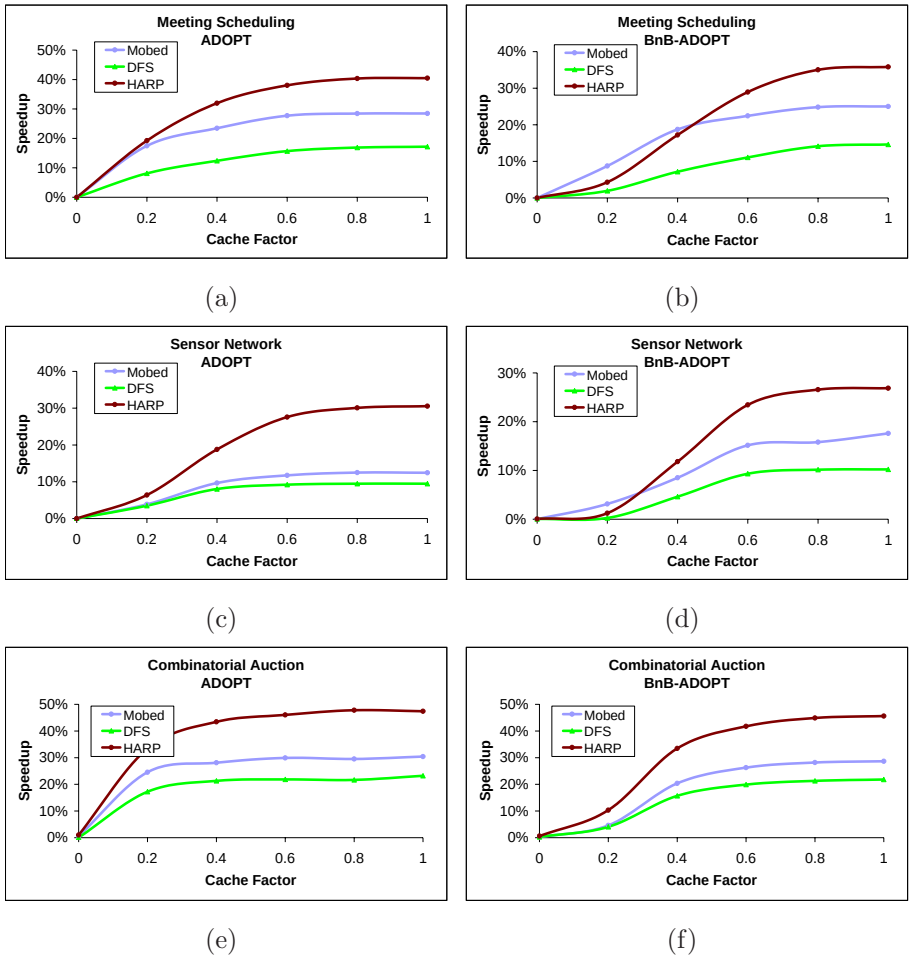


Figure 6.9: Experimental Results Comparing ADOPT and BnB-ADOPT with the Reuse-Bounds Procedure and the Pseudo-tree Reconstruction Algorithms on Sensor Network, Meeting Scheduling and Combinatorial Auction Problems with Change 1 (2)

- Figures 6.7(e-f) give insight into the reasons for the increase in speedup. They compare the size of the reusable search tree (= sum of the cache size of all unaffected agents normalized by the sum of the cache size of all agents) to the speedups of ADOPT and BnB-ADOPT with the ReuseBounds procedure. The Pearson's coefficient shows that there is a correlation of speedup and the size of the reusable search tree with $\rho > 0.65$.

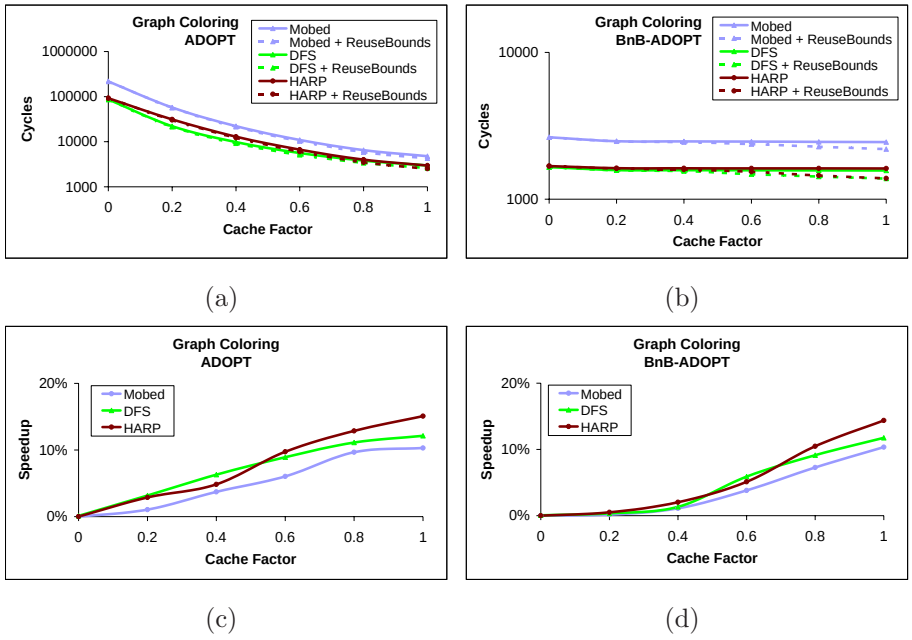


Figure 6.10: Experimental Results Comparing ADOPT and BnB-ADOPT with the ReuseBounds Procedure and the Pseudo-tree Reconstruction Algorithms on Graph Coloring Problems with Change 2

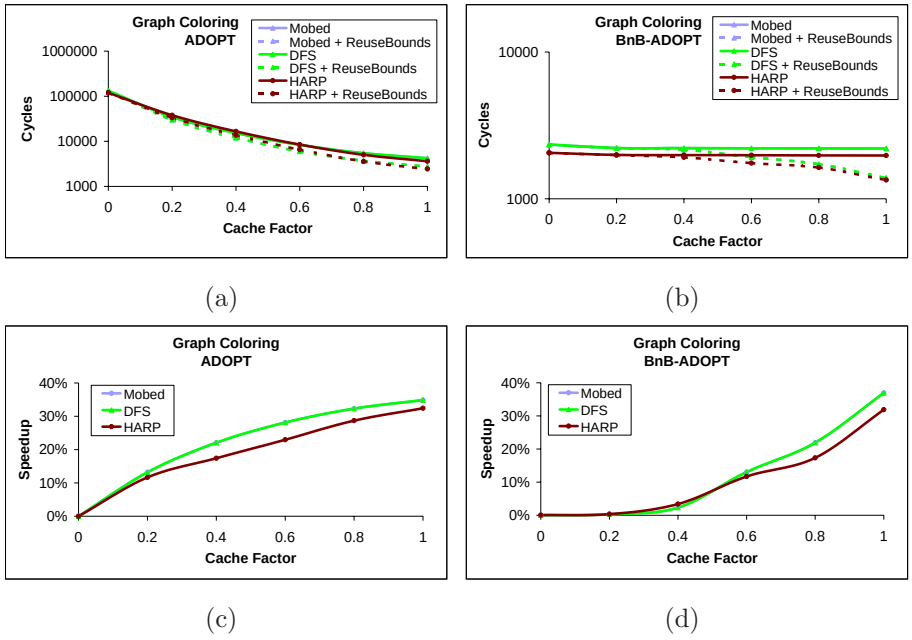


Figure 6.11: Experimental Results Comparing ADOPT and BnB-ADOPT with the ReuseBounds Procedure and the Pseudo-tree Reconstruction Algorithms on Graph Coloring Problems with Change 3

- Figures 6.8 and 6.9 show that the trend for graph coloring problems carries over to sensor network, meeting scheduling and combinatorial auction problems as well.

- Figure 6.10 shows our experimental results for ADOPT and BnB-ADOPT with the ReuseBounds procedure and the pseudo-tree reconstruction algorithms on graph coloring problems with Change 2. The figure shows that the trend for graph coloring problems with Change 1 carries over to graph coloring problems with Change 2.

- Figure 6.11 shows our experimental results for ADOPT and BnB-ADOPT with the ReuseBounds procedure and the pseudo-tree reconstruction algorithms on graph coloring problems with Change 3. The runtimes of ADOPT and BnB-ADOPT with the DFS and Mobed algorithms are identical for each cache factor. Mobed does not reconstruct the pseudo-tree for the current static DCOP problem when the problem is changed with a Type 1 change. DFS reconstructs the exact same pseudo-tree for the current static DCOP problem as the pseudo-tree for the previous static DCOP problem since all the agents are constrained in the exact same way for both static DCOP problems. Unlike the graph coloring problems with Changes 1 and 2, the speedups of ADOPT and BnB-ADOPT with the DFS and Mobed algorithms are larger than the speedups with the HARP algorithm. However, the other trends, namely ADOPT and BnB-ADOPT is faster with the ReuseBounds procedure than without it and the speedups increases with the cache factor, still carry over.

- ADOPT with the HARP algorithm and the ReuseBounds procedure is up to 42% faster than ADOPT with the DFS algorithm. BnB-ADOPT with the HARP algorithm and the ReuseBounds procedure is up to 38% faster than BnB-ADOPT with the DFS algorithm.

Overall, the ReuseBounds procedure experimentally speeds up ADOPT and BnB-ADOPT for all three pseudo-tree reconstruction algorithms and the speedup increases as the cache factor increases. In general, we expect the ReuseBounds procedure to apply to other DCOP search algorithms with other pseudo-tree reconstruction algorithms as well since all DCOP search algorithms perform search and maintain lower and upper bounds on the solution costs. We also expect the HARP algorithm to apply to other DCOP algorithms that operate on pseudo-trees.

6.5 Summary

This chapter modeled a dynamic DCOP problem as a sequence of static DCOP problems and introduced an incremental procedure and an incremental pseudo-tree reconstruction algorithm that allow ADOPT and BnB-ADOPT to reuse information from searches of similar static DCOP problems to guide their search to potentially solve the current static DCOP problem faster. The ReuseBounds procedure is an incremental procedure that allows some agents in ADOPT and BnB-ADOPT to reuse the lower and upper bounds from the previous static DCOP problem to solve the current static DCOP problem. The HARP algorithm is an incremental pseudo-tree reconstruction algorithm that reuses

parts of the pseudo-tree of the previous static DCOP problem to construct the pseudo-tree of the current static DCOP problem. Our experimental results show that ADOPT and BnB-ADOPT with the ReuseBounds procedure and the HARP algorithm terminate faster when they reuse more information. Therefore, these results validate the hypothesis that DCOP search algorithms that reuse information from searches of similar static DCOP problems to guide their search can have runtimes that decrease as they reuse more information.

Chapter 7

Conclusions

The distributed constraint optimization (DCOP) model is becoming popular for formulating and solving agent-coordination problems. As a result, researchers have developed several DCOP algorithms that use search techniques. For example, Asynchronous Distributed Constraint Optimization (ADOPT) is one of the pioneering DCOP search algorithms that has been widely extended. However, solving large problems efficiently become an issue because solving DCOP problems optimally is NP-hard. This dissertation makes the following four contributions:

- In Chapter 3, we investigated how DCOP search algorithms can be sped up by using an appropriate search strategy. We introduced Branch-and-Bound ADOPT (BnB-ADOPT), an extension of ADOPT that changes the search strategy of ADOPT from memory-bounded best-first search to depth-first branch-and-bound search. Experimental results show that BnB-ADOPT tends to be faster than ADOPT when the heuristic values are poorly informed since ADOPT needs to repeatedly reconstruct partial solutions that it purged from memory. In large DCOP problems, where the heuristic values are often poorly informed, BnB-ADOPT is up to one order of

magnitude faster than ADOPT, validating the hypothesis that DCOP search algorithms that employ depth-first branch-and-bound search can be faster than DCOP search algorithms that employ memory-bounded best-first search. Additionally, BnB-ADOPT has great potential as a DCOP search algorithm since heuristic values are often poorly informed for complex DCOP problems, such as DCOP problems with large numbers of agents, large domains, large numbers of constraints or large ranges of constraint costs.

- In Chapter 4, we investigated how DCOP search algorithms can be sped up by sacrificing solution optimality. We introduced an approximation mechanism, called the Weighted Heuristics mechanism, that uses weighted heuristic values to trade off solution costs for smaller runtimes. The new approximation mechanism provides relative error bounds and thus complements existing approximation mechanisms that only provide absolute error bounds. Our experimental results show that, when ADOPT and BnB-ADOPT use the Weighted Heuristics mechanism, they terminate faster with larger weights, validating the hypothesis that DCOP search algorithms that use weighted heuristic values can have runtimes that decrease as larger weights are used. Additionally, the Weighted Heuristics mechanism performs better than the existing approximation mechanisms when used by BnB-ADOPT.

- In Chapter 5, we investigated how DCOP search algorithms can be sped up by using more memory to cache information. We formalized the caching problem for DCOP search algorithms and introduced the MaxPriority, MaxEffort and MaxUtility DCOP-specific caching schemes, which allow ADOPT and BnB-ADOPT to

cache DCOP-specific information when they have more memory available and terminate faster with larger amounts of memory. These three caching schemes uses additional knowledge of the operation of ADOPT and BnB-ADOPT in the form of the lower and upper bounds that every agent maintains. Experimental results show that the MaxEffort and MaxUtility schemes speed up ADOPT more than the currently used generic caching schemes, and the MaxPriority scheme speeds up BnB-ADOPT at least as much as the currently used generic caching schemes. Therefore, these results validate the hypothesis that DCOP-specific caching schemes can reduce the runtime of DCOP search algorithms at least as much as the currently used generic caching schemes.

- In Chapter 6, we investigated how DCOP search algorithms can be sped up by reusing information gained from solving similar DCOP problems. We modeled dynamic DCOP problems as sequences of static DCOP problems and introduced the ReuseBounds procedure and the Hybrid Algorithm for Reconstructing Pseudo-trees (HARP). The ReuseBounds procedure is an incremental procedure that allows ADOPT and BnB-ADOPT to reuse bounds from the previous static DCOP problem to solve the current static DCOP problem. The HARP algorithm is an incremental pseudo-tree reconstruction algorithm that reuses parts of the pseudo-tree of the previous static DCOP problem to construct the pseudo-tree of the current static DCOP problem. Experimental results show that ADOPT and BnB-ADOPT with the ReuseBounds procedure and the HARP algorithm terminate faster when they reuse more information. Therefore, these results validate the hypothesis that DCOP

search algorithms that reuse information from searches of similar DCOP problems to guide their search can have runtimes that decrease as they reuse more information.

Therefore, this dissertation demonstrates that DCOP search algorithms can be sped up by applying the insights gained from centralized search algorithms. For future work, we list other possible approaches used by centralized search algorithms to speed up their search that might be applicable to DCOP search algorithms:

- Centralized approximation algorithms are desirable for solving large problems where finding cost-minimal solutions might be slow. However, it can be the case where there is still time available after the first solution is found. Therefore, researchers have developed centralized anytime search algorithms that find sequences of improving solutions as long as there is time available. ARA* (Likhachev, Gordon, & Thrun, 2003) and AWA* (Hansen & Zhou, 2007) are examples of anytime search algorithms that run repeated Weighted A* searches with decreasing weights.

- Researchers have developed other centralized approximation algorithms aside from Weighted A* that trade off solution costs for smaller runtimes. For example, Optimistic Search (Thayer & Ruml, 2008) is an algorithm that runs a Weighted A* search to find a suboptimal solution and tightens the error bound after finding the solution. Another example is EES (Thayer & Ruml, 2010), which is an algorithm that uses different heuristic values to guide its search and quality guarantee. Researchers have shown that both algorithms can be faster than Weighted A* in sliding tile puzzles and traveling salesman problems (Thayer & Ruml, 2008, 2010).

- Researchers have developed pre-processing techniques to provide well-informed heuristic values to speed up centralized search algorithms. For example, pattern databases store heuristic values for common states in a problem as lookup tables in memory (Culberson & Schaeffer, 1998; Felner, Korf, & Hanan, 2004; Felner, Korf, Meshulam, & Holte, 2007). They have been used to speed up the search for cost-minimal solutions in Rubik's Cube (Korf, 1997), sliding tile puzzles (Korf & Felner, 2002) and multiple sequence alignment problems (Zhou & Hansen, 2004).

These approaches might be applicable to DCOP search algorithms since they use heuristic values to guide their search. Therefore, these approaches can be interesting avenues to investigate to speed up DCOP search algorithms further.

Bibliography

Ali, S., Koenig, S., & Tambe, M. (2005). Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1041–1048.

Arnborg, S., Corneil, D., & Proskurowski, A. (1987). Complexity of finding embeddings in a k-tree. *SIAM Journal of Discrete Mathematics*, *8*(2), 277–284.

Atlas, J., & Decker, K. (2007). A complete distributed constraint optimization method for non-traditional pseudotree arrangements. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 736–743.

Bacchus, F., Chen, X., van Beek, P., & Walsh, T. (2002). Binary vs. non-binary constraints. *Artificial Intelligence*, *140*(1-2), 1–37.

Bayardo, R., & Miranker, D. (1995). On the space-time trade-off in solving constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 558–562.

Bowring, E., Pearce, J., Portway, C., Jain, M., & Tambe, M. (2008). On k-optimal distributed constraint optimization algorithms: New bounds and algorithms. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 607–614.

Bowring, E., Tambe, M., & Yokoo, M. (2006). Multiply-constrained distributed constraint optimization. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1413–1420.

Bowring, E., Yin, Z., Zinkov, R., & Tambe, M. (2009). Sensitivity analysis for distributed optimization with resource constraints. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 633–640.

Brito, I., & Meseguer, P. (2010). Improving DPOP with function filtering. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 141–158.

Burke, D. (2008). *Exploiting Problem Structure in Distributed Constraint Optimisation with Complex Local Problems*. Ph.D. thesis, National University of Ireland, Cork (Ireland).

Burke, D., & Brown, K. (2006). Efficiently handling complex local problems in distributed constraint optimisation. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pp. 701–702.

Carpenter, C., Dugan, C., Kopena, J., Lass, R., Naik, G., Nguyen, D., Sultanik, E., Modi, P., & Regli, W. (2007). Intelligent systems demonstration: Disaster evacuation support. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1964–1965.

Chakrabarti, P., Ghosh, S., Acharya, A., & DeSarkar, S. (1989). Heuristic search in restricted memory. *Artificial Intelligence*, *47*, 197–221.

Chechetka, A., & Sycara, K. (2006a). An any-space algorithm for distributed constraint optimization. In *Proceedings of the AAAI Spring Symposium on Distributed Plan and Schedule Management*, pp. 33–40.

Chechetka, A., & Sycara, K. (2006b). No-commitment branch and bound search for distributed constraint optimization. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1427–1429.

Choxi, H., & Modi, P. (2007). A distributed constraint optimization approach to wireless network optimization. In *Proceedings of the AAAI-07 Workshop on Configuration*, pp. 1–8.

Collin, Z., & Dolev, S. (1994). Self-stabilizing depth first search. *Information Processing Letters*, *49*, 297–301.

Culberson, J., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Davin, J., & Modi, P. (2006). Hierarchical variable ordering for multiagent agreement problems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1433–1435.

Dechter, R. (Ed.). (2003). *Constraint Processing*. Morgan Kaufmann.

Dechter, R., & Dechter, A. (1988). Belief maintenance in dynamic constraint networks. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 37–42.

Dechter, R., & Pearl, J. (1985). Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery*, *32*(3), 505–536.

Farinelli, A., Rogers, A., Petcu, A., & Jennings, N. (2008). Decentralised coordination of low-power embedded devices using the Max-Sum algorithm. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 639–646.

Felner, A., Korf, R., & Hanan, S. (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, *22*, 279–318.

Felner, A., Korf, R., Meshulam, R., & Holte, R. (2007). Compressed pattern databases. *Journal of Artificial Intelligence Research*, *30*, 213–247.

Fitzpatrick, S., & Meertens, L. (2003). Distributed coordination through anarchic optimization. In Lesser, V., Ortiz, C., & Tambe, M. (Eds.), *Distributed Sensor Networks: A Multiagent Perspective*, pp. 257–295. Kluwer.

Freuder, E., & Quinn, M. (1985). Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1076–1078.

Gershman, A., Meisels, A., & Zivan, R. (2009). Asynchronous Forward-Bounding for distributed COPs. *Journal of Artificial Intelligence Research*, *34*, 61–88.

Goldman, C., & Zilberstein, S. (2004). Decentralized control of cooperative systems: Categorization and complexity analysis. *Journal of Artificial Intelligence Research*, *22*, 143–174.

Greenstadt, R. (2009). An overview of privacy improvements to k-optimal DCOP algorithms (extended abstract). In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1279–1280.

Greenstadt, R., Grosz, B., & Smith, M. (2007). SSDPOP: Improving the privacy of DCOP with secret sharing. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1098–1100.

Grinshpoun, T., & Meisels, A. (2008). Completeness and performance of the APO algorithm. *Journal of Artificial Intelligence Research*, *33*, 223–258.

Gutierrez, P., & Meseguer, P. (2010). Saving redundant messages in BnB-ADOPT. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1259–1260.

Hamadi, Y., Bessière, C., & Quinqueton, J. (1998). Distributed intelligent backtracking. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pp. 219–223.

Hansen, E., & Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research*, *28*, 267–297.

Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *SSC4*(2), 100–107.

Hirayama, K., & Yokoo, M. (1997). Distributed partial constraint satisfaction problem. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pp. 222–236.

Hirayama, K., & Yokoo, M. (2000). An approach to over-constrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction. In *Proceedings of the International Conference on Multi-Agent Systems (ICMAS)*, pp. 135–142.

Hirayama, K., & Yokoo, M. (2005). The distributed breakout algorithms. *Artificial Intelligence*, *161*(1-2), 89–115.

Junges, R., & Bazzan, A. (2008). Evaluating the performance of DCOP algorithms in a real world, dynamic problem. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 599–606.

Kiekintveld, C., Yin, Z., Kumar, A., & Tambe, M. (2010). Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 133–140.

Koenig, S., & Likhachev, M. (2002). D* Lite. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 476–483.

Koenig, S., & Likhachev, M. (2005). Adaptive A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1311–1312.

Koenig, S., Likhachev, M., & Furcy, D. (2004a). Lifelong Planning A*. *Artificial Intelligence*, *155*(1–2), 93–146.

Koenig, S., Likhachev, M., Liu, Y., & Furcy, D. (2004b). Incremental heuristic search in artificial intelligence. *AI Magazine*, *25*(2), 99–112.

Koenig, S., Likhachev, M., & Sun, X. (2007). Speeding up moving-target search. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1136–1143.

Korf, R. (1993). Linear-space best-first search. *Artificial Intelligence*, *62*(1), 41–78.

Korf, R. (1997). Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 700–705.

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

Kumar, A., Faltings, B., & Petcu, A. (2009). Distributed constraint optimization with structured resource constraints. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 923–930.

Kumar, A., Petcu, A., & Faltings, B. (2008). H-DPOP: Using hard constraints for search space pruning in DCOP. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 325–330.

Lass, R., Kopena, J., Sultanik, E., Nguyen, D., Dugan, C., Modi, P., & Regli, W. (2008). Coordination of first responders under communication and resource constraints (short paper). In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1409–1413.

Lesser, V., Ortiz, C., & Tambe, M. (Eds.). (2003). *Distributed Sensor Networks: A Multiagent Perspective*. Kluwer.

Leyton-Brown, K., Pearson, M., & Shoham, Y. (2000). Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, pp. 66–76.

Likhachev, M., Gordon, G., & Thrun, S. (2003). ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems 16 (NIPS)*.

Lisỳ, V., Zivan, R., Sycara, K., & Péchoucek, M. (2010). Deception in networks of mobile sensing agents. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1031–1038.

Maheswaran, R., Pearce, J., & Tambe, M. (2004). Distributed algorithms for DCOP: A graphical game-based approach. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, pp. 432–439.

Mailler, R. (2004). *A Mediation-based Approach to Cooperative, Distributed Problem Solving*. Ph.D. thesis, University of Massachusetts at Amherst, Amherst (United States).

Mailler, R., & Lesser, V. (2004). Solving distributed constraint optimization problems using cooperative mediation. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 438–445.

Marinescu, R., & Dechter, R. (2007). Best-first AND/OR search for graphical models. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1171–1176.

Marinescu, R., & Dechter, R. (2009). AND/OR branch-and-bound search for combinatorial optimization in graphical models. *Artificial Intelligence*, *173*(16-17), 1457–1491.

Matsui, T., Matsuo, H., & Iwata, A. (2005). Efficient methods for asynchronous distributed constraint optimization algorithm. In *Proceedings of the International Conference on Artificial Intelligence and Applications (AIA)*, pp. 727–732.

Matsui, T., Silaghi, M., Hirayama, K., Yokoo, M., & Matsuo, H. (2009). Directed soft arc consistency in pseudo trees. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1065–1072.

Matsui, T., Silaghi, M., Hirayama, K., Yokoo, M., & Matsuo, H. (2008). Resource constrained distributed constraint optimization with virtual variables. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 120–125.

Meisels, A., Kaplansky, E., Razgon, I., & Zivan, R. (2002). Comparing performance of distributed constraints processing algorithms. In *Proceedings of the Distributed Constraint Reasoning Workshop*, pp. 86–93.

Modi, P. (2003). *Distributed Constraint Optimization for Multiagent Systems*. Ph.D. thesis, University of Southern California, Los Angeles (United States).

Modi, P., & Ali, S. (2004). Distributed constraint reasoning under unreliable communication. In Zhang, W., & Sorge, V. (Eds.), *Frontiers in Artificial Intelligence and Applications*, Vol. 112, pp. 141–150. IOS Press.

Modi, P., Shen, W.-M., Tambe, M., & Yokoo, M. (2005). ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, *161*(1-2), 149–180.

Neller, T. (2002). Iterative-refinement for action timing discretization. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 492–497.

Ottens, B., & Faltings, B. (2008). Coordinating agent plans through distributed constraint optimization. In *Proceedings of the ICAPS-08 Workshop on Multiagent Planning*.

Pearce, J. (2007). *Local Optimization in Cooperative Agent Networks*. Ph.D. thesis, University of Southern California, Los Angeles (United States).

Pearce, J., Maheswaran, R., & Tambe, M. (2006). Solution sets for DCOPs and graphical games. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 577–584.

Pearce, J., & Tambe, M. (2007). Quality guarantees on k-optimal solutions for distributed constraint optimization problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1446–1451.

Pearl, J. (1985). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pearl, J., & Kim, J. (1982). Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *4*(4), 391–399.

Pecora, F., Modi, P., & Scerri, P. (2006). Reasoning about and dynamically posting n-ary constraints in ADOPT. In *Proceedings of the Distributed Constraint Reasoning Workshop*, pp. 57–71.

Petcu, A. (2007). *A Class of Algorithms for Distributed Constraint Optimization*. Ph.D. thesis, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland).

Petcu, A., & Faltings, B. (2005a). Approximations in distributed optimization. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pp. 802–806.

Petcu, A., & Faltings, B. (2005b). A scalable method for multiagent constraint optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1413–1420.

Petcu, A., & Faltings, B. (2005c). Superstabilizing, fault-containing multiagent combinatorial optimization. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 449–454.

Petcu, A., & Faltings, B. (2007). MB-DPOP: A new memory-bounded algorithm for distributed optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1452–1457.

Petcu, A., Faltings, B., & Mailler, R. (2007). PC-DPOP: A new partial centralization algorithm for distributed optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 167–172.

Petcu, A., Faltings, B., & Parkes, D. (2008). M-DPOP: Faithful distributed implementation of efficient social choice problems. *Journal of Artificial Intelligence Research*, *32*, 705–755.

Pohl, I. (1970). First results on the effect of error in heuristic search. *Machine Intelligence*, *5*, 219–236.

Pohl, I. (1973). The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 12–17.

Pynadath, D., & Tambe, M. (2002). The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, *16*, 389–423.

Russell, S. (1992). Efficient memory-bounded search methods. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pp. 1–5.

Sandholm, T. (2002). Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, *135*(1-2), 1–54.

Schiex, T., & Verfaillie, G. (1994). Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, *3*(2), 187–207.

Schurr, N., Okamoto, S., Maheswaran, R., Scerri, P., & Tambe, M. (2005). Evolution of a teamwork model. In Sun, R. (Ed.), *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, pp. 307–327. Cambridge University Press.

Silaghi, M., Landwehr, J., & Larrosa, J. (2004). Asynchronous branch & bound and A* for disWCSPs with heuristic function based on consistency-maintenance. In Zhang, W., & Sorge, V. (Eds.), *Frontiers in Artificial Intelligence and Applications*, Vol. 112, pp. 49–62. IOS Press.

Silaghi, M., Lass, R., Sultanik, E., Regli, W., Matsui, T., & Yokoo, M. (2008). The operation point units of distributed constraint solvers. In *Proceedings of the Distributed Constraint Reasoning Workshop*, pp. 1–16.

Silaghi, M., & Yokoo, M. (2009). ADOPT-ing: Unifying asynchronous distributed optimization with asynchronous backtracking. *Autonomous Agents and Multi-Agent Systems*, *19*(2), 89–123.

Stentz, A. (1995). The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1652–1659.

Stranders, R., Delle Fave, F., Rogers, A., & Jennings, N. (2010). A decentralised coordination algorithm for mobile sensors. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 874–880.

Stranders, R., Farinelli, A., Rogers, A., & Jennings, N. (2009a). Decentralised coordination of continuously valued control parameters using the Max-Sum algorithm. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 601–608.

Stranders, R., Farinelli, A., Rogers, A., & Jennings, N. (2009b). Decentralised coordination of mobile sensors using the Max-Sum algorithm. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 299–304.

Sultanik, E., Lass, R., & Regli, W. (2009). Dynamic configuration of agent organizations. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 305–311.

Sun, X., & Koenig, S. (2007). The Fringe-Saving A* search algorithm – a feasibility study. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 2391–2397.

Sun, X., Koenig, S., & Yeoh, W. (2008). Generalized Adaptive A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 469–476.

Sun, X., Yeoh, W., & Koenig., S. (2009a). Dynamic Fringe-Saving A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 891–898.

Sun, X., Yeoh, W., & Koenig, S. (2009b). Efficient incremental search for moving target search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 615–620.

Sun, X., Yeoh, W., & Koenig, S. (2010a). Generalized Fringe-Retriving A*: Faster moving target search on state lattices. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1081–1087.

Sun, X., Yeoh, W., & Koenig, S. (2010b). Moving Target D* Lite. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 67–74.

Thayer, J., & Ruml, W. (2008). Faster than Weighted A*: An optimistic approach to bounded suboptimal search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 355–362.

Thayer, J., & Ruml, W. (2010). Finding acceptable solutions faster using inadmissible information. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*.

Ueda, S., Iwasaki, A., & Yokoo, M. (2010). Coalition structure generation based on distributed constraint optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 197–203.

Vinyals, M., Pujol, M., Rodriguez-Aguilarhas, J., & Cerquides, J. (2010). Divide-and-coordinate: DCOPs by agreement. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 149–156.

Vinyals, M., Rodríguez-Aguilar, J., & Cerquides, J. (2009). Generalizing DPOP: Action-GDL, a new complete algorithm for DCOPs (extended abstract). In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1239–1240.

Yeoh, W., Felner, A., & Koenig, S. (2009). IDB-ADOPT: A depth-first search DCOP algorithm. In Oddi, A., Fages, F., & Rossi, F. (Eds.), *Recent Advances in Constraints*, Vol. 5655 of *Lecture Notes in Artificial Intelligence*, pp. 132–146. Springer.

Yeoh, W., Felner, A., & Koenig, S. (2010). BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, *38*, 85–133.

Yeoh, W., Sun, X., & Koenig, S. (2009a). Trading off solution quality for faster computation in DCOP search algorithms. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 354–360.

Yeoh, W., Varakantham, P., & Koenig, S. (2009b). Caching schemes for DCOP search algorithms. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 609–616.

Yokoo, M. (Ed.). (2001). *Distributed Constraint Satisfaction: Foundation of Cooperation in Multi-agent Systems*. Springer.

Yokoo, M., Durfee, E., Ishida, T., & Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, *10*(5), 673–685.

Yokoo, M., Durfee, E., Ishida, T., & Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pp. 614–621.

Zhang, W. (2000). Depth-first branch-and-bound versus local search: A case study. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 930–935.

Zhang, W., & Korf, R. (1995). Performance of linear-space search algorithms. *Artificial Intelligence*, *79*(2), 241–292.

Zhang, W., Xing, Z., Wang, G., & Wittenburg, L. (2003). An analysis and application of distributed constraint satisfaction and optimization algorithms in sensor networks. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 185–192.

Zhou, R., & Hansen, E. (2004). Space-efficient memory-based heuristics. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 677–682.

Zivan, R. (2008). Anytime local search for distributed constraint optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 393–398.

Zivan, R., Glinton, R., & Sycara, K. (2009). Distributed constraint optimization for large teams of mobile sensing agents. In *Proceedings of the International Conference on Intelligent Agent Technology (IAT)*, pp. 347–354.