UNIVERSITY OF CALIFORNIA,
IRVINE

AND/OR Search Strategies for Combinatorial Optimization in Graphical Models

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Radu Marinescu

Dissertation Committee:
Professor Rina Dechter, Chair
Professor Padhraic Smyth
Professor Alex Ihler

2008

The dissertation of Radu Marinescu
is approved and is acceptable in quality and form for
publication on microfilm and in digital formats:

_____

_____

_____

Committee Chair

University of California, Irvine
2008

DEDICATION

To my wife, Beti.

TABLE OF CONTENTS

LIST OF FIGURES viii

LIST OF TABLES xi

ACKNOWLEDGMENTS xiii

CURRICULUM VITAE xiv

ABSTRACT OF THE DISSERTATION xvi

1 Introduction 1
- 1.1 Dissertation Outline and Contributions . . . 2
  - 1.1.1 Systematic versus Non-systematic Search for Bayesian MPE . . 4
  - 1.1.2 AND/OR Branch-and-Bound Search for Graphical Models . . . 6
  - 1.1.3 Memory Intensive AND/OR Search for Graphical Models . . . 7
  - 1.1.4 AND/OR Search for 0-1 Integer Linear Programming 9
  - 1.1.5 AND/OR Multi-Valued Decision Diagrams for Optimization . . 10
- 1.2 Preliminaries . . . 11
  - 1.2.1 Notations . . . 11
  - 1.2.2 Graph Concepts . . . 12
  - 1.2.3 Propositional Theories . . . 13
  - 1.2.4 AND/OR Search Spaces . . . 13
  - 1.2.5 Graphical Models . . . 14
  - 1.2.6 Constraint Networks . . . 16
  - 1.2.7 Cost Networks . . . 17
  - 1.2.8 Belief Networks . . . 20
- 1.3 Search and Inference for Optimization in Graphical Models . . . 23
  - 1.3.1 Bucket and Mini-Bucket Elimination . . . 23
  - 1.3.2 Systematic Search with Mini-Bucket Heuristics . . . 27
  - 1.3.3 Branch-and-Bound Search for Weighted CSP . . . 34

2 Systematic versus Non-systematic Search for Most Probable Explanations 39
- 2.1 Introduction . . . 39
- 2.2 Background . . . 41
  - 2.2.1 Cluster-Tree Elimination . . . 42
  - 2.2.2 Mini-Cluster-Tree Elimination . . . 44
- 2.3 Partition-based Branch-and-Bound Search . . . 45
  - 2.3.1 BBBT: Branch-and-Bound with Dynamic Heuristics . . . 45
  - 2.3.2 BBMB: Branch-and-Bound with Static Heuristics . . . 46

2.4 Non-Systematic Algorithms . . . 47
2.4.1 Local Search . . . 47
2.4.2 Iterative Join-Graph Propagation . . . 48
2.5 Experiments . . . 49
2.5.1 Random Bayesian Networks and Noisy-OR Networks . . . 50
2.5.2 Random Grid Networks . . . 57
2.5.3 Random Coding Networks . . . 59
2.5.4 Real-World Networks . . . 60
2.6 Conclusion to Chapter 2 . . . 61

3 AND/OR Branch-and-Bound Search for Graphical Models 62
3.1 Introduction . . . 62
3.2 AND/OR Search Trees For Graphical Models . . . 65
3.3 AND/OR Branch-and-Bound Search . . . 71
3.4 Lower Bound Heuristics for AND/OR Search . . . 82
3.4.1 Static Mini-Bucket Heuristics . . . 82
3.4.2 Dynamic Mini-Bucket Heuristics . . . 87
3.4.3 Local Consistency Based Heuristics for AND/OR Search . . . 89
3.5 Dynamic Variable Orderings . . . 90
3.5.1 Partial Variable Ordering (PVO) . . . 91
3.5.2 Full Dynamic Variable Ordering (DVO) . . . 92
3.5.3 Dynamic Separator Ordering (DSO) . . . 93
3.6 Experimental Results . . . 94
3.6.1 Overview and Methodology . . . 94
3.6.2 Finding Good Pseudo Trees . . . 97
3.6.3 Results for Empirical Evaluation on Bayesian Networks . . . 100
3.6.4 The Impact of Determinism in Bayesian Networks . . . 119
3.6.5 Results for Empirical Evaluation on Weighted CSPs . . . 126
3.6.6 The Impact of Dynamic Variable Orderings . . . 133
3.7 Related Work . . . 142
3.8 Conclusion to Chapter 3 . . . 143

4 Memory Intensive AND/OR Search for Graphical Models 147
4.1 Introduction . . . 147
4.2 AND/OR Search Graphs for Graphical Models . . . 149
4.3 AND/OR Branch-and-Bound with Caching . . . 153
4.3.1 Naive Caching . . . 156
4.3.2 Adaptive Caching . . . 158
4.4 Best-First AND/OR Search . . . 160
4.5 Experimental Results . . . 163
4.5.1 Overview and Methodology . . . 163
4.5.2 Results for Empirical Evaluation of Bayesian Networks . . . 167
4.5.3 The Anytime Behavior of AND/OR Branch-and-Bound Search 188
4.5.4 The Impact of Determinism in Bayesian Networks . . . 197
4.5.5 Results for Empirical Evaluation of Weighted CSPs . . . 201

4.6 Conclusion to Chapter 4 . . . 215

5 AND/OR Search for 0-1 Integer Programming 221
5.1 Introduction . . . 221
5.1.1 Contribution . . . 222
5.1.2 Chapter Outline . . . 222
5.2 Background . . . 223
5.2.1 Integer Programming . . . 223
5.2.2 Branch-and-Bound Search for Integer Programming . . . 225
5.2.3 Branch-and-Cut Search for Integer Programming . . . 226
5.2.4 State-of-the-art Software Packages . . . 226
5.3 Extending AND/OR Search Spaces to 0-1 ILPs . . . 227
5.3.1 AND/OR Search Trees for 0-1 ILPs . . . 227
5.3.2 AND/OR Search Graphs for 0-1 ILPs . . . 229
5.4 Depth-First AND/OR Branch-and-Bound Search . . . 230
5.5 Best-First AND/OR Search . . . 234
5.6 Dynamic Variable Orderings . . . 236
5.7 Experimental Results . . . 238
5.7.1 Combinatorial Auctions . . . 241
5.7.2 Uncapacitated Warehouse Location Problems . . . 248
5.7.3 MAX-SAT Instances . . . 257
5.8 Conclusion to Chapter 5 . . . 261

6 AND/OR Multi-Valued Decision Diagrams for Constraint Optimization 264
6.1 Introduction . . . 264
6.2 Review of Binary Decision Diagrams . . . 266
6.3 Weighted AND/OR Multi-Valued Decision Diagrams . . . 267
6.4 Using AND/OR Search to Generate AOMDDs . . . 269
6.4.1 The Search Based Compile Algorithm . . . 272
6.5 Experiments . . . 274
6.5.1 Weighted CSPs . . . 274
6.5.2 0-1 Integer Linear Programs . . . 278
6.5.3 Summary of Empirical Results . . . 282
6.6 Conclusion to Chapter 6 . . . 282

7 Software 283
7.1 REES: Reasoning Engine Evaluation Shell . . . 283
7.1.1 REES Architecture . . . 284
7.1.2 A Closer Look . . . 286
7.2 AND/OR Search for Optimization . . . 290
7.2.1 AOLIB-MPE . . . 291
7.2.2 AOLIB-WCSP . . . 292
7.2.3 AOLIB-ILP . . . 294

8 Conclusion 296

Bibliography **307**

LIST OF FIGURES

1.1 Constraint network. 17
1.2 A cost network. 18
1.3 Belief network. 20
1.4 Execution of BE and MBE(i). 27
1.5 Search space for $f^*(a, e, d)$ 29
1.6 Four equivalent WCSPs (for $\top = 4$) 37

2.1 Cluster-Tree Elimination. 43
2.2 Branch-and-Bound with MBTE(i) - BBBT(i) 45
2.3 Random Bayesian networks. Accuracy vs. time. 52
2.4 Random Noisy-OR networks. Accuracy vs. time. 53
2.5 Random Bayesian networks. Solution quality. 55
2.6 Random Bayesian networks. Solution quality. 56

3.1 AND/OR search spaces for graphical models. 67
3.2 Arc weights for a cost network with 5 variables and 4 cost functions. . . 70
3.3 A partial solution tree and possible extensions to solution trees. 72
3.4 Cost of a partial solution tree. 77
3.5 Illustration of the pruning mechanism. 79
3.6 Static mini-bucket heuristics for $i = 3$. 83
3.7 AND/OR versus OR static mini-bucket heuristics for $i = 3$. 87
3.8 Dynamic mini-bucket heuristics for $i = 3$. 89
3.9 Full dynamic variable ordering for AND/OR Branch-and-Bound search. 92
3.10 Static vs. dynamic mini-buckets. Random belief networks 102
3.11 Coding network. 104
3.12 Static vs. dynamic mini-buckets. Random coding networks 107
3.13 Static vs. dynamic mini-buckets. Grid networks. 109
3.14 Min-fill vs. hypergraph pseudo trees. Grid networks. 109
3.15 A fragment of a belief network used in genetic linkage analysis. 110
3.16 Min-fill vs. hypergraph pseudo trees. Genetic linkage analysis. 113
3.17 Min-fill vs hypergraph pseudo trees. ISCAS'89 networks 115
3.18 Min-fill vs. hypergraph pseudo trees. UAI'06 Dataset. 117
3.19 Static vs. dynamic mini-buckets. Grid networks. 123
3.20 Static vs. dynamic mini-buckets. SPOT5 networks 128
3.21 Min-fill vs. hypergraph pseudo trees. SPOT5 benchmarks. 129
3.22 Static vs. dynamic mini-buckets. ISCAS'89 circuits (WCSP) 131
3.23 Min-fill vs. hypergraph pseudo trees. ISCAS'89 circuits (WCSP). . . . 132

3.24 Min-fill vs. hypergraph pseudo trees. Mastermind game instances. . . . 134
3.25 Min-fill vs. hypergraph pseudo trees. SPOT5 benchmark. 136
3.26 Results for sparse random binary WCSPs. 140
3.27 Results for dense random binary WCSPs. 141

4.1 AND/OR search graph for graphical models. 151
4.2 Naive caching 157
4.3 Adaptive caching 159
4.4 Static vs. dynamic mini-buckets. Coding networks. 171
4.5 Static vs. dynamic mini-buckets. Grid networks. 174
4.6 Naive vs. adaptive caching. Grid networks. 176
4.7 Min-fill vs. hypergraph pseudo trees. Grid networks. 178
4.8 Memory usage. Grid networks. 179
4.9 Impact of i-bound on static mini-buckets. Linkage networks. 181
4.10 Min-fill vs. hypergraph pseudo trees. Linkage networks. 183
4.11 Naive vs. adaptive caching. Genetic linkage analysis 184
4.12 Min-fill vs. hypergraph pseudo trees. UAI'06 networks. 189
4.13 Anytime AOBB search. Linkage networks. 192
4.14 Anytime AOBB search. Grid networks. 192
4.15 Anytime AOBB search. UAI'06 networks. 193
4.16 Static vs. dynamic mini-buckets. SPOT5 networks. 204
4.17 Min-fill vs. hypergraph pseudo trees. SPOT5 networks. 206
4.18 Static vs. dynamic mini-buckets. ISCAS'89 circuits (WCSP) 210
4.19 Naive vs. adaptive caching. ISCAS'89 circuits (WCSP) 211
4.20 Min-fill vs. hypergraph pseudo trees. ISCAS'89 networks (WCSP). . . 212
4.21 Naive vs. adaptive caching. Mastermind networks. 214
4.22 Min-fill vs. hypergraph pseudo trees. Mastermind networks. 216
4.23 Memory usage. Mastermind networks. 217

5.1 Example of a 0-1 Integer Linear Program. 224
5.2 AND/OR search tree for 0-1 ILP. 228
5.3 AND/OR search graph for 0-1 ILP. 229
5.4 Illustration of the pruning mechanism. 233
5.5 Results for *regions-upv* combinatorial auctions. 242
5.6 Results for *arbitrary-upv* combinatorial auctions. 243
5.7 Results for *regions-upv* combinatorial auctions (CPLEX) 244
5.8 Results for *arbitrary-upv* combinatorial auctions (CPLEX) 245
5.9 Results on *regions-npv* combinatorial auctions 249
5.10 Results on *arbitrary-npv* combinatorial auctions 250
5.11 Results on *regions-npv* combinatorial auctions (CPLEX) 251
5.12 Results on *arbitrary-npv* combinatorial auctions (CPLEX) 252
5.13 Results for `dubois` instances 260

6.1 Boolean function representations 266
6.2 Reduction rules 267

6.3 AND/OR search tree for COP 270
6.4 AND/OR graphs for COP 270
6.5 AOMDD for SPOT5 networks. 275
6.6 AOMDD for combinatorial auctions. 280
6.7 AOMDD for `dubois` instances. 281

7.1 REES Plug-In Architecture 285
7.2 REES Graphical Interface. (a) Model. (b) Experiment 287
7.3 REES Results Display Window. 289

LIST OF TABLES

2.1 Random Bayesian networks. Average accuracy and time. 50
2.2 Random Noisy-OR networks. Average accuracy and time. 50
2.3 Random Bayesian networks. Accuracy. 54
2.4 Random Bayesian networks. BBBT vs BBMB. 57
2.5 Random grid networks. Average accuracy and time. 58
2.6 Random coding networks. Average BER and time. 58
2.7 Real-world Bayesian networks. Average accuracy and time. 59

3.1 Experiments for Bayesian networks. 95
3.2 Experiments for Weighted CSPs. 96
3.3 Min-fill versus hypergraph pseudo trees. 98
3.4 Results for random belief networks. 101
3.5 Results for random coding networks with 64 bits. 105
3.6 Results for random coding networks with 128 bits. 106
3.7 Results for grid networks. 108
3.8 Results for genetic linkage analysis networks. 111
3.9 Results for ISCAS'89 networks (BN). 114
3.10 Results for UAI'06 Evaluation Dataset. 116
3.11 Results for Bayesian Network Repository. 118
3.12 Deterministic CPT $P(C|A,B)$ 121
3.13 Results for deterministic grid networks. 122
3.14 Results for deterministic ISCAS'89 networks (BN). 124
3.15 Results for deterministic linkage networks. 125
3.16 Results for SPOT5 benchmarks. 126
3.17 Results for ISCAS'89 circuits (WCSP). 130
3.18 Results for Mastermind game instances. 133
3.19 Results for SPOT5 benchmark. Dynamic variable ordering. 135
3.20 Results for CELAR benchmark. Dynamic variable ordering. 137
3.21 Results for random binary WCSPs. 137

4.1 Experiments for Bayesian networks. 165
4.2 Experiments for WCSP. 166
4.3 Results for coding networks. 169
4.4 Results for grid networks. Static mini-buckets. 172
4.5 Results for grid networks. Dynamic mini-buckets. 173
4.6 Results for genetic linkage analysis (1) 179
4.7 Results for genetic linkage analysis (2) 180

4.8 Min-fill vs. hypergraph pseudo trees. Genetic linkage networks. 182
4.9 Results for UAI'06 networks (1). 186
4.10 Results for UAI'06 networks (2). 187
4.11 Results for ISCAS'89 circuits (BN). Static mini-buckets. 190
4.12 Results for ISCAS'89 circuits (BN). Dynamic mini-buckets. 191
4.13 Genetic linkage networks with GLS (1) 194
4.14 Genetic linkage networks with GLS (2) 195
4.15 Results for grid networks with GLS. 196
4.16 Results for UAI'06 networks with GLS (1) 197
4.17 Results for UAI'06 networks with GLS (2). 198
4.18 Results for grid networks with SAT. 199
4.19 Results for ISCAS'89 ciruits (BN) with SAT. 200
4.20 Results for SPOT5 networks. Static mini-buckets. 202
4.21 Results for SPOT5 networks. Dynamic mini-buckets. 203
4.22 Results for ISCAS'89 circuits (WCSP). Static mini-buckets. 207
4.23 Results for ISCAS'89 circuits (WCSP). Dynamic mini-buckets. 208
4.24 Results on Mastermind networks. 213

5.1 Detailed outline of the experimental evaluation for 0-1 ILP. 238
5.2 Results for UWLP instances with 50 warehouses and 200 stores 254
5.3 Results for UWLP instances with 50 warehouses and 400 stores 255
5.4 Results for `pret` instances. 258

6.1 AOMDD for ISCAS'89 circuits. 276
6.2 AOMDD for planning instances. 277
6.3 AOMDD for MIPLIB instances. 279
6.4 AOMDD for `pret` instances. 280

ACKNOWLEDGMENTS

I am indebted to all those who have helped me finish this dissertation. Many thanks to my committee members, Professor Padhraic Smyth and Professor Alex Ihler, for their comments on earlier drafts of the dissertation.

Thanks also to my research group members, Kalev Kask, Bozhena Bidyuk, Robert Mateescu, Vibhav Gogate and Lars Otten for many insightful discussions.

Last but not least, I am much obliged to my advisor, Rina Dechter. I thank her for her wise advice, for her constructive criticism, and for cheering me up every time I felt despair. It has been a treat working with her.

During my graduate school years I was supported by the NSF grants IIS-0086529 and IIS-0412854, by the MURI ONR award N00014-00-1-0617, by the NIH grant R01-HG004175-02, and by the Donald Bren School of Information and Computer Science at University of California, Irvine.

CURRICULUM VITAE

Radu Marinescu

EDUCATION

Ph.D. Information and Computer Science, 2008
Donald Bren School of Information and Computer Science
University of California, Irvine
Dissertation: AND/OR Search Strategies for Combinatorial Optimization in Graphical Models
Advisor: Rina Dechter

M.S. Information and Computer Science, 2004
Donald Bren School of Information and Computer Science
University of California, Irvine

M.S. Computer Science, 2000
University "Politehnica" of Bucharest, Romania

B.S. Computer Science, 1999
University "Politehnica" of Bucharest, Romania

PUBLICATIONS

[1] Robert Mateescu, Radu Marinescu and Rina Dechter. AND/OR Multi-Valued Decision Diagrams for Constraint Optimization. In *Proceedings of the Fourteenth International Conference on Principles and Practice of Constraint Programming (CP)*, 2007.

[2] Radu Marinescu and Rina Dechter. Best-First AND/OR Search for Graphical Models. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI)*, 2007.

[3] Radu Marinescu and Rina Dechter. Best-First AND/OR Search for Most Probable Explanations. In *Proceedings of the Twenty-Third International Conference on Uncertainty in Artificial Intelligence (UAI)*, 2007.

[4] Radu Marinescu and Rina Dechter. Best-First AND/OR Search for 0/1 Integer Programming. In *Proceedings of the Fourth International Conference on the Integration of AI and OR Techniques for Combinatorial Optimization (CPAIOR)*, 2007.

[5] Radu Marinescu and Rina Dechter. AND/OR Branch-and-Bound Search for 0/1 Integer Programming. In *Proceedings of the Third International Conference on the Integration of AI and OR Techniques for Combinatorial Optimization (CPAIOR)*, 2006.

[6] Radu Marinescu and Rina Dechter. Dynamic Orderings for AND/OR Branch-and-Bound Search in Graphical Models. In *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI)*, 2006.

[7] Radu Marinescu and Rina Dechter. Memory Intensive Branch-and-Bound Search in Graphical Models. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI)*, 2006.

[8] Radu Marinescu and Rina Dechter. AND/OR Search for Genetic Linkage Analysis. In *Workshop on Heuristic Search, Memory Based Heuristics and their Applications of the 21st National Conference on Artificial Intelligence (AAAI)*, 2006.

[9] Radu Marinescu and Rina Dechter. AND/OR Branch-and-Bound Search in Graphical Models. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

[10] Radu Marinescu, Kalev Kask and Rina Dechter. Systematic versus Non-systematic Search Algorithms for Most Probable Explanations. In *Proceedings of the Nineteenth International Conference on Uncertainty in Artificial Intelligence (UAI)*, 2003.

Abstract of the Dissertation

AND/OR Search Strategies for Combinatorial Optimization in Graphical Models

By

Radu Marinescu

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2008

Professor Rina Dechter, Chair

This thesis presents a new generation of search algorithms for solving combinatorial optimization problems over graphical models. The new algorithms exploit the principles of problem decomposition using the AND/OR search spaces, avoid redundant solution of subproblems using memory, focus on relevant promising portions of the solution space using the power of the mini-bucket heuristics and prune irrelevant spaces using constraint propagation. As we show throughout the chapters of this thesis, putting all these principles together yields powerful algorithms whose performance improves upon earlier schemes significantly, sometimes by several orders of magnitude. We demonstrate the applicability and the generality of our algorithms on optimization tasks over both probabilistic and deterministic graphical models, often showing superior performance on real application such as linkage analysis and circuit design and diagnosis. The following paragraphs elaborate.

Our algorithms explore the AND/OR search spaces of the underlying graphical model. The AND/OR search space is a unifying paradigm for advanced search schemes for graphical models exploiting problem decomposability, which can translate into exponential time

savings for search algorithms. In conjunction with the AND/OR search space we also investigate a class of partition-based heuristic functions, based on the Mini-Bucket approximation.

We start by introducing depth-first Branch-and-Bound search algorithms that explore the AND/OR tree, use a variety of sources for heuristic guidance and incorporate some dynamic variable ordering heuristics. We then extend the depth-first AND/OR Branch-and-Bound and best-first search algorithms with the ability to recognize identical subproblems and avoid redundant solutions by caching (similar to good and no-good recording), thus traversing the AND/OR search graph. We also extend all the principles acquired within the general framework of depth-first and best-first schemes to the well known 0-1 Integer Linear Programs.

Our empirical evaluation shows conclusively that the new AND/OR search algorithms improve dramatically over current state-of-the-art approaches exploring the traditional OR search space, in many cases by several orders of magnitude. We illustrate one by one the gain obtained by exploiting problem's decomposition (using AND modes), equivalence (by caching), branching strategy (via dynamic variable ordering heuristics), control strategy (depth-first or best-first) as well as the impact of the lower bound heuristic strength. As well, we investigate the impact of exploiting hard constraint ($i.e.$, determinism) in the problem, the initial upper bound provided to the algorithm, and the quality of the guiding variable orderings.

In the last part of the thesis we also show how our AND/OR search algorithms can be used as compilation algorithms for AND/OR decision diagrams. We present a new algorithm for compiling AND/OR Multi-Valued Decision Diagrams (AOMDDs) that represent the set of optimal solutions. We extend earlier work on AND/OR decision diagrams by considering general weighted graphical models based on cost functions rather than constraints. On various domains we show that we sometimes get a substantial reduction beyond the initial trace of state-of-the-art search algorithms.

Finally, the starting chapter of this thesis (Chapter 2) sets the stage for this whole work by comparing the power of static and dynamic mini-bucket heuristics over regular search spaces and compares against a number of popular stochastic local search algorithms, as well as against the class of iterative belief propagation algorithms.

Chapter 1

Introduction

Graphical models such as belief networks, constraint networks, Markov networks or influence diagrams are a widely used knowledge representation framework for reasoning with probabilistic and deterministic information. These models use *graphs* (directed or undirected) that provide an intuitively appealing mechanism by which one can model highly interacting sets of variables. This allows for a concise representation of the knowledge that lends itself naturally to the design of efficient graph-based query processing algorithms.

Optimization queries such as finding the most likely state of a belief network, finding a solution that violates the least number of constraints in a constraint network or finding a set of actions that maximizes the expected utility in an influence diagram can be defined within the framework of graphical models. These tasks are NP-hard and they are typically solved by either *inference* or *search* algorithms.

Inference-based algorithms were always known to be good at exploiting the independencies captured by the underlying graphical model yielding worst case time and space guarantees exponential in the treewidth of the graph. Traditional search-based algorithms traverse the model's search space where each path represents a partial or full solution. The linear structure of the search space does not retain the conditional independencies represented in the model and, therefore, search algorithms may not be nearly as effective as inference algorithms in using this information. They are time exponential in the number of variables. However, since they may operate in linear space, search algorithms are often the preferred choice for problems with large treewidth.

The focus of this dissertation is on search algorithms for optimization that do exploit the structure of the problem. We present depth-first and best-first search regimes that are sensitive to the conditional independencies encoded in the model's graph and decompose the problem into independent components using the AND/OR data structure, often resulting in exponential time savings.

1.1 Dissertation Outline and Contributions

We next provide a brief description of the structure of the dissertation, while the subsequent subsections will provide more details.

- Chapter 2 explores the power of two systematic Branch-and-Bound search algorithms that traverse the traditional OR search space and exploit the mini-bucket heuristics, BBBT(i) - Branch-and-Bound with Mini-Bucket-Tree heuristics (for which the heuristic information is constructed during search and allows dynamic variable and value ordering) and its predecessor BBMB(i) - Branch-and-Bound with Mini-Bucket heuristics (for which the heuristic information is pre-compiled). We compare them against a number of popular stochastic local search (SLS) algorithms, as well as against the recently popular iterative belief propagation algorithms. We show empirically that the new Branch-and-Bound algorithm, BBBT(i), demonstrates tremendous pruning of the search space far beyond its predecessor, BBMB(i), which translates into impressive time saving for some classes of problems. Second, when viewed as approximation anytime schemes, BBBT(i) and BBMB(i) together are highly competitive with the best known SLS algorithms and are superior, especially when the domain sizes increase beyond 2. The results also show that the class of belief propagation algorithms can in general outperform SLS, but they are quite inferior to BBMB(i) and BBBT(i).

- Chapter 3 is the first (in 3 chapters) to present and evaluate the power of a new

framework for optimization in graphical models, based on AND/OR search spaces. It focuses on linear space search which explores the AND/OR search *tree* rather than the search *graph* and makes no attempt to cache information. Specifically, we introduce a depth-first Branch-and-Bound algorithms that explore the AND/OR search tree using static and dynamic variable orderings. We also investigate the power of the mini-bucket heuristics in both static and dynamic setups within this AND/OR search framework. We focus on two popular optimization problems in graphical models: finding the Most Probable Explanation in belief networks and solving Weighted CSPs. In extensive empirical evaluations using a variety of benchmarks we demonstrate conclusively that this new depth-first AND/OR Branch-and-Bound approach improves dramatically over the traditional OR search.

- Chapter 4 extends the depth-first AND/OR Branch-and-Bound algorithm to explore an AND/OR search *graph*, rather than a tree, by equipping it with a context-based adaptive caching scheme similar to good and no-good recording. Since *best-first* strategies are known to be superior to depth-first when memory is utilized, exploring the best-first control strategy is called for. Therefore, we also introduce a new class of best-first AND/OR search algorithms that explore the context minimal AND/OR search graph. Our empirical results demonstrate conclusively the superiority of the new memory intensive AND/OR search approach over traditional OR search with caching as well as over AND/OR Branch-and-Bound without caching discussed in Chapter 3.

- Chapter 5 extends both depth-first and best-first AND/OR search algorithms to solving 0-1 Integer Linear Programs (0-1 ILPs). We also extend dynamic variable ordering heuristics while exploring an AND/OR search tree for 0-1 ILPs. We demonstrate the effectiveness of the new search algorithms on a variety of benchmarks, including real-world combinatorial auctions, random uncapacitated warehouse location prob-

lems and MAX-SAT instances.

- Chapter 6 presents a new top down search-based algorithm for compiling AND/OR Multi-Valued Decision Diagrams (AOMDDs), as representations of the optimal set of solutions for constraint optimization problems. The approach is based on AND/OR search spaces for graphical models, on AND/OR Branch-and-Bound with caching, and on decision diagram reduction techniques. We extend earlier work on AOMDDs by considering general weighted graphs based on cost functions rather than constraints. An extensive experimental evaluation on a variety of benchmarks proves the efficiency of the weighted AOMDD data structure.

- Chapter 7 presents the software implementation of the algorithms described in the dissertation. Chapter 8 concludes the thesis.

The following subsections provide more details of the content in each chapter.

1.1.1 Systematic versus Non-systematic Search for Bayesian MPE

The chapter explores the power of two systematic Branch-and-Bound search algorithms that exploit partition-based heuristics for solving the Most Probable Explanation (MPE) task in Bayesian networks. While it is known that the MPE task is NP-hard [22], it is nonetheless a common task in applications such as diagnosis, abduction, and explanation. For example, given data on clinical findings, MPE can postulate a patient's probable affliction. In decoding, the task is to identify the most likely input message transmitted over a noisy channel given the observed output. Researchers in natural language consider the understanding of text to consist of finding the most likely facts (in an internal representation) that explain the existence of the given text. In computer vision and image understanding, researchers formulate the problem in terms of finding the most likely set of objects that explain the image.

Contribution

We introduce a new algorithm, called BBBT(i), for solving the MPE task in Bayesian networks. It takes the idea of mini-bucket partition-based heuristics one step further and explores the feasibility of generating such heuristics *during search*, rather than in a *pre-processing manner*. This, in particular, allows dynamic variable orderings – a feature that can have a tremendous effect on search. The dynamic generation of these heuristics is facilitated by a recent extension of Mini-Bucket Elimination (MBE) [30] to *Mini-Bucket Tree Elimination (MBTE)*, a partition-based approximation defined over cluster trees and described in [36]. This yields algorithm BBBT(i) that computes the MBTE(i) heuristic at each node of the search tree. We compare BBBT(i) against BBMB(i) [65], a Branch-and-Bound algorithm for which the heuristic information is pre-compiled. We also compare the two Branch-and-Bound algorithms against several best-known SLS algorithms as well as a class of generalized belief propagation algorithms adapted for the MPE task.

We provide an extensive empirical evaluation on various random and real-world benchmarks showing that BBMB(i) and BBBT(i) do not dominate one another. While BBBT(i) can sometimes significantly improve over BBMB(i), in many other instances its (quite significant) pruning power does not outweigh its time overhead. Both algorithms are powerful in different cases. In general, for large i-bounds, which are more effective, BBMB(i) is more powerful, however when space is at issue BBBT(i) with small i-bounds is often more powerful. More significantly, we show that the SLS algorithms we used are overall inferior to BBBT(i) and BBMB(i), except when the domain size is small. The superiority of BBBT(i) and BBMB(i) is especially significant because unlike local search they can prove optimality if given enough time. We also demonstrate that generalized belief propagation algorithms are often superior to the SLS class we used as well.

1.1.2 AND/OR Branch-and-Bound Search for Graphical Models

Search-based algorithms ($e.g.$, depth-first Branch-and-Bound, best-first search) traverse the search space of the problem, where each path represents a partial or full solution. The linear structure of search spaces does not retain the independencies represented in the underlying graphical models and, therefore, search-based algorithms may not be as effective as inference-based algorithms in using this information. On the other hand, the space requirements of search algorithms may be much less severe than those of inference algorithms as they can operate in linear space. In addition, search methods can accommodate an *implicit* specification of the functional relationships ($i.e.$, procedural or functional form) while inference schemes often rely on an explicit tabular representation over the (discrete) variables. For these reasons, search algorithms are the only choice available for models with large treewidth and with implicit representation. In earlier work, AND/OR search spaces were introduced as data structures that can be used to exploit problem decomposition during search.

The AND/OR search space for graphical models [38] is a relatively new framework for search that is sensitive to the conditional independencies in the model, often resulting in significantly reduced complexities. It is guided by a *pseudo tree* [48, 106] that captures independencies in the graphical model, resulting in a search space exponential in the depth of the pseudo tree, rather than in the number of variables.

Contribution

In this chapter we develop a new generation of AND/OR Branch-and-Bound algorithms (AOBB) that explore the AND/OR search tree in a depth-first manner for solving optimization problems in graphical models. As in traditional Branch-and-Bound search, the efficiency of these algorithms depends heavily also on their guiding heuristic function. We extend the *mini-bucket heuristics*, which were shown to be powerful for optimization problems in the context of OR search spaces [65], to the AND/OR search framework. The

Mini-Bucket algorithm [42] provides a general scheme for extracting the heuristic information automatically, from the functional specification of the graphical model. Since the accuracy of this algorithm is controlled by a bounding parameter, called i-bound, it allows heuristics having varying degrees of accuracy and results in a spectrum of search algorithms that can trade off heuristic computation and search [65]. In this chapter we show how the pre-computed mini-bucket heuristic as well as any other heuristic information can be incorporated into AND/OR search and we subsequently introduce *dynamic mini-bucket heuristics*, which are computed dynamically at each node of the search tree.

Since variable selection can influence dramatically the search performance, we also introduce a collection of *dynamic* variable ordering heuristics that can be accommodated by the AND/OR decomposition principle.

We apply our depth-first AND/OR Branch-and-Bound approach to both the MPE task in belief networks [104] and to Weighted CSPs [9]. Our empirical results show conclusively that the new depth-first AND/OR Branch-and-Bound algorithms improve dramatically over traditional OR search space, especially when the heuristic estimates are inaccurate.

1.1.3 Memory Intensive AND/OR Search for Graphical Models

It is often the case that a search space that is a tree can become a graph if we merge nodes that root identical subproblems. Some of these nodes can be identified based on *contexts* [38]. The context of a node is a subset of the currently assigned variables that completely determines the remaining subproblem using graph information only. The AND/OR search tree can be transformed into a graph by merging identical subtrees. Consequently, algorithms that explore the search graph involve controlled memory management that allows improving their time performance by increasing their use of memory.

Contribution

In this chapter we extend the AND/OR Branch-and-Bound algorithm to explore the context minimal AND/OR search *graph*, rather than the AND/OR search tree, using a flexible caching mechanism that can adapt to memory limitations. The caching scheme is based on contexts and is similar to good and no-good recording and recent schemes appearing in Recursive Conditioning [24], Valued Backtracking [4] as well as Backtracking with Tree Decomposition [59].

Since best-first search is known to be superior among memory intensive search algorithms [40], we present a new best-first AND/OR search algorithm that explores the context minimal AND/OR search graph. Under conditions of admissibility and monotonicity of the guiding heuristic function, best-first search is known to expand the minimal number of nodes, at the expense of using additional memory [40].

The efficiency of the proposed memory intensive depth-first and best-first AND/OR search methods also depends on the accuracy of the guiding heuristic function, which is based on the Mini-Bucket approximation. Here, we explore empirically the efficiency of the mini-bucket heuristics in both static and dynamic settings, as well as the interaction between the heuristic strength within the cache-based search spaces.

Our empirical results (on both MPE and Weighted CSP) demonstrate conclusively that the new memory intensive AND/OR search algorithms improve dramatically (up to several orders of magnitude) over competitive approaches, especially when the heuristic estimates are less accurate. We illustrate the impressive gains in performance caused by exploiting equivalence (caching), control strategy (depth-first or best-first) as well as strength of the guiding lower bound function. We also investigate key factors that impact the performance of any search algorithm such as: the availability of hard constraints ($i.e.$, determinism) in the problem, the availability of good initial upper bounds provided to the algorithm, and the availability of good quality guiding pseudo trees.

1.1.4 AND/OR Search for 0-1 Integer Linear Programming

One of the most important optimization problems in operations research and computer science is *integer programming* [95]. Applications of integer programming include scheduling, routing, VLSI circuit design, combinatorial auctions and facility locations [95]. A 0-1 Integer Linear Program (0-1 ILP) is to optimize ($i.e.$, minimize or maximize) a linear objective function of binary integer decision variables, subject to a set of linear equality or inequality constraints defined on subsets of variables. The classical approach to solving 0-1 ILPs is the *Branch-and-Bound* method [74]. The algorithm keeps in memory the best solution found so far (the *incumbent*). Once a node in the search tree is generated, a lower bound ($i.e.$, heuristic evaluation function) on the solution value is computed by solving the linear relaxation ($i.e.$, relaxing the integrality constraints for all undecided variables) of the current subproblem ($e.g.$, using the *simplex* method [23]), while honoring the commitments made on the search path so far. A path terminates when the lower bound is at least the value of the incumbent, or the subproblem is infeasible or yields an integer solution. Once all paths have terminated, the incumbent is a provably optimal solution.

Contribution

In this chapter we extend the general principles of solving constraint optimization problems using AND/OR search with context-based caching to the class of 0-1 ILPs. We explore both depth-first and best-first control strategies. We also incorporate our dynamic variable ordering heuristics for AND/OR search and explore their impact on 0-1 ILPs. We demonstrate empirically the benefit of the AND/OR algorithms on benchmarks including combinatorial auctions, random uncapacitated warehouse location problems and MAX-SAT problem instances. Our results show conclusively that the new search algorithms improve dramatically over the traditional OR search on this domain, in some cases with several orders of magnitude of improved performance. We illustrate the tremendous gain obtained by exploiting problem's decomposition (using AND nodes), equivalence (by caching), branching

strategy (via dynamic variable ordering heuristics) and control strategy. We also show that the AND/OR algorithms are sometimes able (though not frequently) to outperform significantly commercial solvers such as CPLEX.

1.1.5 AND/OR Multi-Valued Decision Diagrams for Optimization

The compilation of graphical models, including constraint and probabilistic networks, has recently been under intense investigation. Compilation techniques are useful when an extended off-line computation can be traded for fast real-time answers. Typically, a tractable compiled representation of the problem is desired. Since the tasks of interest are in general NP-hard, this is not always possible in the worst case. In practice, however, it is often the case that the compiled representation is much smaller than the worst case bound, as was observed for Ordered Binary Decision Diagrams (OBDDs) [13] which are extensively used in hardware and software verification.

In the context of constraint networks, compilation schemes are very useful for interactive solving or product configuration type problems [45, 52]. These are combinatorial problems where a compact representation of the feasible set of solutions is necessary. The system has to be *complete* (to represent all sets of solutions), *backtrack-free* (to never encounter dead-ends) and *real-time* (to provide fast answers).

Contribution

In this chapter we present a compilation scheme for constraint optimization. Our goal is to obtain a compact representation of the set of optimal solutions, by employing techniques from search, optimization and decision diagrams. Our approach is based on three main ideas: (1) AND/OR search spaces for graphical models [38], (2) Branch-and-Bound search for optimization, applied to AND/OR search spaces [79, 82] and (3) reduction rules similar to OBDDs, that lead to the compilation of the search algorithm trace into an AND/OR Multi-Valued Decision Diagram (AOMDD) [89]. The novelty over previous results con-

sists in: (1) the treatment of general weighted graphs based on cost functions, rather than constraints; (2) a top down search based approach for generating the AOMDD, rather than Variable Elimination based as in [89]; (3) extensive experimental evaluation that proves the compilation potential of the weighted AOMDD. We show that the compilation scheme can often be accomplished relatively efficiently and that we sometimes get a substantial reduction beyond the initial trace of state-of-the-art search algorithms.

1.2 Preliminaries

The remainder of this chapter contains preliminary notation and definitions, gives examples of graphical models and reviews previous work on inference and search based algorithms for optimization tasks over graphical models.

1.2.1 Notations

A reasoning problem is defined in terms of a set of variables taking values on finite domains and a set of functions defined over these variables. We denote variables by uppercase letters ($e.g.$, $X, Y, Z, ...$) and values of variables by lower case letters ($e.g.$, $x, y, z, ...$). Sets are usually denoted by bold letters, for example $\mathbf{X} = \{X_1, ..., X_n\}$ is a set of variables. An assignment $(X_1 = x_1, ..., X_n = x_n)$ can be abbreviated as $x = (\langle X_1, x_1 \rangle, ..., \langle X_n, x_n \rangle)$ or $x = (x_1, ..., x_n)$. For a subset of variables $\mathbf{Y}$, $D_{\mathbf{Y}}$ denotes the Cartesian product of the domains of variables in $\mathbf{Y}$. The projection of an assignment $x = (x_1, ..., x_n)$ over a subset $\mathbf{Y}$ is denoted by $x_{\mathbf{Y}}$ or $x[\mathbf{Y}]$. We will denote by $\mathbf{Y} = y$ (or y for short) the assignment of values to variables in $\mathbf{Y}$ from their respective domains. We denote functions by letters f, h, g etc., and the scope (set of arguments) of a function f by $scope(f)$.

1.2.2 Graph Concepts

A *directed graph* is a pair $G = \{\mathbf{V}, \mathbf{E}\}$, where $\mathbf{V} = \{X_1, ..., X_n\}$ is a set of vertices (nodes), and $\mathbf{E} = \{(X_i, X_j) | X_i, X_j \in \mathbf{V}\}$ is a set of edges (arcs). If $(X_i, X_j) \in \mathbf{E}$, we say that X_i points to X_j. The degree of a vertex is the number of incident arcs to it. For each vertex X_i, $pa(X_i)$ or pa_i, is the set of vertices pointing to X_i in G, while the set of child vertices of X_i, denoted $ch(X_i)$, comprises the variables that X_i points to. The family of X_i, denoted F_i, includes X_i and its parent vertices. A directed graph is acyclic if it has no directed cycles. An *undirected* graph is defined similarly to a directed graph, but there is no directionality associated with the edges.

DEFINITION **1 (induced width)** *An* ordered graph *is a pair* (G, d) *where* G *is an undirected graph, and* $d = (X_1, ..., X_n)$ *is an ordering of the nodes. The* width of a node *is the number of the node's neighbors that precede it in the ordering. The* width of an ordering d *is the maximum width over all nodes. The* induced width of an ordered graph, *denoted by* $w^*(d)$, *is the width of the induced ordered graph obtained as follows: nodes are processed from last to first; when node* X_i *is processed, all its preceding neighbors are connected. The* induced width *of a graph, denoted by* w^*, *is the minimal induced width over all its orderings.*

DEFINITION **2 (hypergraph)** *A* hypergraph *is a pair* $H = (\mathbf{X}, \mathbf{S})$, *where* $\mathbf{S} = \{S_1, ..., S_t\}$ *is a set of subsets of* $\mathbf{X}$ *called hyperedges.*

DEFINITION **3 (tree decomposition)** *A* tree decomposition *of a hypergraph* $H = (\mathbf{X}, \mathbf{S})$, *is a tree* $T = (\mathbf{V}, \mathbf{E})$, *where* $\mathbf{V}$ *is a set of nodes, also called "clusters", and* $\mathbf{E}$ *is a set of edges, together with a labeling function* χ *that associates with each vertex* $v \in \mathbf{V}$ *a set* $\chi(v) \subseteq \mathbf{X}$ *satisfying:*

(1) For each $S_i \in \mathbf{S}$ *there exists a vertex* $v \in \mathbf{V}$ *such that* $S_i \subseteq \chi(v)$*;*

(2) (running intersection property) For each $X_i \in \mathbf{X}$, the set $\{v \in \mathbf{V} | X_i \in \chi(v)\}$ induces a connected subtree of T.

DEFINITION **4 (treewidth, pathwidth)** *The* width *of a tree decomposition of a hypergraph is the size of the largest cluster minus 1 (i.e., $max_v|\chi(v) - 1|$). The* treewidth *of a hypergraph is the minimum width along all possible tree decompositions. The* pathwidth *is the treewidth over the restricted class of chain decompositions.*

It is easy to see that given an induced graph, the set of maximal cliques (also called clusters) provide a tree decomposition of the graph, namely the clusters can be connected in a tree structure that satisfies the running intersection property. It is well known that the induced width of a graph is identical to its treewidth [41]. For various relationships between these and other graph parameters see [3, 11, 10].

1.2.3 Propositional Theories

Propositional variables which can take only two values $\{true, false\}$ or $\{1, 0\}$ are denoted by uppercase letters $P, Q, R, ...$ Propositional literals ($i.e.$, P, $\neg P$) stand for $P = true$ or $P = false$, and disjunctions of literals, or *clauses*, are denoted by $\alpha, \beta, ...$ For instance, $\alpha = P \vee \neg Q \vee R$ is a clause. A *unit clause* is a clause of size 1. The *resolution* operation over two clauses $(\alpha \vee Q)$ and $(\beta \vee \neg Q)$ results in a clause $(\alpha \vee \beta)$, thus eliminating Q. A formula φ in *conjunctive normal form* (CNF) is a set of clauses $\varphi = \{\alpha_1, ..., \alpha_t\}$ that denotes their conjunction. The set of *models* or *solutions* of a formula φ, denoted $m(\varphi)$, is the set of all truth assignments to all its symbols (variables) that do not violate any clause.

1.2.4 AND/OR Search Spaces

An AND/OR state space representation of a problem is defined by a 4-tuple $\langle S, O, S_g, s_0 \rangle$ [96]. S is a set of states which can be either OR or AND states (the OR states represent alternative ways for solving the problem while the AND states often represent problem

decomposition into subproblems, all of which need to be solved). O is a set of operators. An OR operator transforms an OR state into another state, and an AND operator transforms an AND state into a set of states. There is also a set of goal states $S_g \subseteq S$ and a start node $s_0 \in S$.

The AND/OR state space model induces an explicit AND/OR search *graph*. Each state is a node and child nodes are obtained by applicable AND or OR operators. The search graph includes a *start* node. The terminal nodes (having no children) are labeled as SOLVED or UNSOLVED.

A *solution tree* of an AND/OR search graph G is a tree S_G which: (1) contains the start node s_0; (2) if $n \in S_G$ is an OR node then it contains one of its child nodes in G and if $n \in S_G$ is an AND node it contains all its children in G; (3) all its terminal nodes are labeled SOLVED. AND/OR graphs can have a cost associated with each arc, and the cost of a solution tree is a function ($e.g.$, sum-cost) of the arcs included in the tree. In this case we may seek a solution tree with optimal (maximum or minimum) cost [96].

1.2.5 Graphical Models

Graphical models include constraint networks [34] defined by relations of allowed tuples, (directed or undirected) probabilistic networks [104], defined by conditional probability tables over subsets of variables, cost networks defined by cost functions, and influence diagrams [55] which include both probabilistic functions and cost functions ($i.e.$, utilities) [33]. Each graphical model comes with its typical queries, such as finding a solution (over constraint networks), finding the most probable assignment or updating the posterior probabilities given evidence (posed over probabilistic networks), or finding optimal solutions for cost networks. The task for influence diagrams is to choose a sequence of actions that maximizes the expected utility. Markov random fields are the undirected counterparts of probabilistic networks. They are defined by a collection of probabilistic functions called potentials, over arbitrary subsets of variables.

In general, a graphical model is defined by a collection of functions $\mathbf{F}$, over a set of variables $\mathbf{X}$, conveying probabilistic, deterministic or preferential information, whose structure is captured by a graph.

DEFINITION **5 (graphical model)** *A* graphical model $\mathcal{R}$ *is a 4-tuple* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$*, where:*

1. $\mathbf{X} = \{X_1, ..., X_n\}$ *is a set of variables;*
2. $\mathbf{D} = \{D_1, ..., D_n\}$ *is the set of their respective finite domains of values;*
3. $\mathbf{F} = \{f_1, ..., f_r\}$ *is a set of real-valued functions, each defined over a subset of variables* $S_i \subseteq \mathbf{X}$ *(i.e., the scope);*
4. $\otimes_i f_i \in \{\prod_i f_i, \sum_i f_i\}$ *is a combination operator.*

The graphical model represents the combination of all its functions: $\otimes_{i=1}^{r} f_i$*.*

DEFINITION **6 (cost of a full and partial assignment)** *Given a graphical model* $\mathcal{R}$*, the cost of a full assignment* $x = (x_1, ..., x_n)$ *is defined by* $c(x) = \otimes_{f \in \mathbf{F}} f(x[scope(f)])$*. Given a subset of variables* $\mathbf{Y} \subseteq \mathbf{X}$*, the cost of a partial assignment* y *is the combination of all the functions whose scopes are included in* $\mathbf{Y}$ *(*$\mathbf{F_Y}$*) evaluated at the assigned values. Namely,* $c(y) = \otimes_{f \in \mathbf{F_Y}} f(y[scope(f)])$*. We will often abuse notation writing* $c(y) = \otimes_{f \in \mathbf{F_Y}} f(y)$ *instead.*

DEFINITION **7 (primal graph)** *The* primal graph *of a graphical model has the variables as its nodes and an edge connects any two variables that appear in the scope of the same function.*

There are various queries (tasks) that can be posed over graphical models. We refer to all as *automated reasoning problems*. In general, an optimization task is a reasoning problem defined as a function from a graphical model to a set of elements, most commonly, the real numbers.

DEFINITION **8 (constraint optimization problem)** *A* constraint optimization problem (or COP for short) *is a pair* $\mathcal{P} = \langle \mathcal{R}, \Downarrow_{\mathbf{X}} \rangle$*, where* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$ *is a graphical model. If* S *is the scope of function* $f \in \mathbf{F}$ *and* $\Downarrow_S f \in \{max_S f, min_S f\}$*, the optimization problem is to compute* $\Downarrow_{\mathbf{X}} \otimes_{i=1}^{r} f_i$*.*

The min/max ($\Downarrow$) operator is sometimes called an *elimination* operator because it removes the arguments from the input functions' scopes.

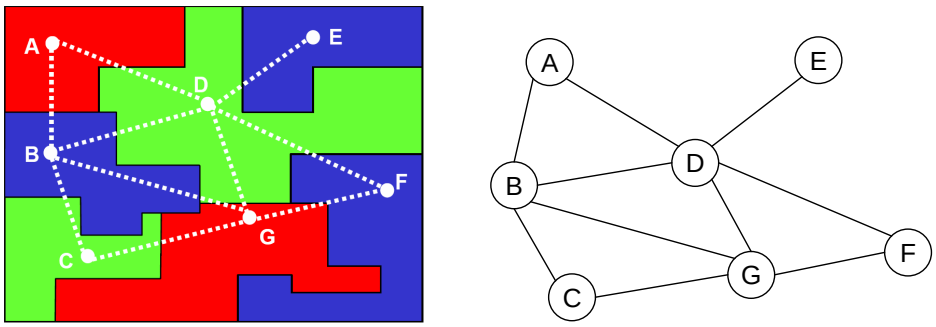
We next elaborate on the three popular graphical models of constraint networks, cost networks and belief networks which will be the primary focus of this dissertation.

1.2.6 Constraint Networks

Constraint networks provide a framework for formulating real world problems, such as scheduling and design, planning and diagnosis, and many more as a set of constraints between variables. The *constraint satisfaction* (CSP) task is to find an assignment of values to all the variables that does not violate any constraints, or else to conclude that the problem is inconsistent. Other tasks are finding all solutions and counting the solutions.

DEFINITION **9 (constraint network)** *A* constraint network (CN) *is a graphical model* $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \bowtie \rangle$*, where* $\mathbf{X} = \{X_1, ..., X_n\}$ *is a set of variables, associated with discrete-valued domains* $\mathbf{D} = \{D_1, ..., D_n\}$*, and a set of constraints* $\mathbf{C} = \{C_1, ..., C_r\}$*. Each constraint* C_i *is a pair* (S_i, R_i)*, where* R_i *is a relation* $R_i \subseteq D_{S_i}$ *defined on a subset of variables* $S_i \subseteq \mathbf{X}$*. The relation denotes all compatible tuples of* D_{S_i} *allowed by the constraint. The combination operator* $\otimes$ *is join,* $\bowtie$*. The primal graph of a constraint network is called a* constraint graph*. A solution is an assignment of values to all variables* $x = (x_1, ..., x_n)$*,* $x_i \in D_i$*, such that* $\forall C_i \in C$*,* $x_{S_i} \in R_i$*. The constraint network represents its set of solutions,* $\bowtie_i C_i$*. The elimination operator in this case is* projection*.*

Example 1 *Figure 1.1(a) shows a graph coloring problem that can be modeled by a constraint network. Given a map of regions, the problem is to color each region by one of the*



(a) Graph coloring problem (b) Constraint graph

Figure 1.1: Constraint network.

given colors {*red, green, blue*}*, such that neighboring regions have different colors. The variables of the problem are the regions, and each one has the domain* {*red,green,blue*}*. The constraints are the relation "different" between neighboring regions. Figure 1.1(b) shows the constraint graph, and a solution* $(A = red, B = blue, C = green, D = green, E = blue, F = blue, G = red)$ *is given in Figure 1.1(a).*

Propositional Satisfiability. A special case of a CSP is *propositional satisfiability* (SAT). A formula ϕ in *conjunctive normal form* (CNF) is a conjunction of clauses $\alpha_1, ..., \alpha_t$, where a clause is a disjunction of *literals* (propositions or their negations). For example, $\alpha = (P \vee \neg Q \vee \neg R)$ is a clause, where P, Q and R are propositions, and P, $\neg Q$ and $\neg R$ are literals. The SAT problem is to decide whether a given CNF theory has a model, $i.e.$, a truth-assignment to its propositions that does not violate any clause. Propositional satisfiability can be defined as a CSP, where propositions correspond to variables, domains are $\{0, 1\}$, and constraints are represented by clauses. For example the clause $(\neg A \vee B)$ is a relation over its propositional variables that allows all tuples over (A, B) except $(A = 1, B = 0)$.

1.2.7 Cost Networks

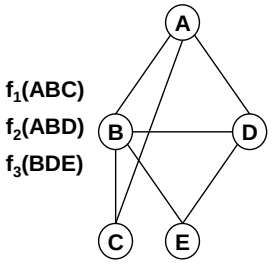
An immediate extension of constraint networks are *cost networks* where the set of functions are real-valued cost functions, the combination and elimination operators are *summation* and *minimization*, respectively, and the primary constraint optimization task is to find a

A	B	C	f_1(ABC)
0	0	0	∞
0	0	1	∞
0	1	0	∞
0	1	1	2
1	0	0	∞
1	0	1	2
1	1	0	∞
1	1	1	2

A	B	D	f_2(ABD)
0	0	0	1
0	0	1	∞
0	1	0	0
0	1	1	2
1	0	0	6
1	0	1	5
1	1	0	6
1	1	1	5

B	D	E	f_3(BDE)
0	0	0	∞
0	0	1	3
0	1	0	∞
0	1	1	4
1	0	0	∞
1	0	1	3
1	1	0	∞
1	1	1	4

(a) Cost functions



(b) Constraint graph

Figure 1.2: A cost network.

solution with minimum cost, namely finding $min_{\mathbf{X}} \sum_{i=1}^{r} f_i$.

A special class of cost networks which has gained a lot of interest in recent years is the Weighted CSP (WCSP) [9]. WCSP extends the classical CSP formalism with *soft constraints* which are represented as positive integer-valued cost functions. Formally,

DEFINITION **10 (WCSP)** *A* Weighted CSP (WCSP) *is a graphical model* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$ *where each of the functions* $f_i \in \mathbf{F}$ *assigns "0" (no penalty) to allowed tuples and a positive integer penalty cost to the forbidden tuples. Namely,* $f_i : D_{S_{i_1}} \times ... \times D_{S_{i_t}} \rightarrow \mathbb{N}$*, where* $S_i = \{S_{i_1}, ..., S_{i_t}\}$ *is the scope of the function. The optimization problem is to find a value assignment to the variables with minimum penalty cost, namely finding* $min_{\mathbf{X}} \sum_i f_i$*.*

DEFINITION **11 (MAX-CSP)** *A* MAX-CSP *is a WCSP with all penalty costs equal to* 1*. Namely,* $\forall f_i \in \mathbf{F}$*,* $f_i : D_{S_{i_1}} \times ... \times D_{S_{i_t}} \rightarrow \{0, 1\}$*, where* $scope(f_i) = S_i = \{S_{i_1}, ..., S_{i_t}\}$*.*

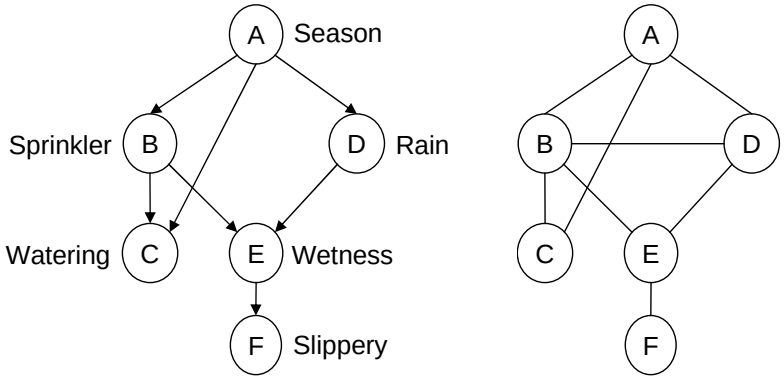
Solving a MAX-CSP task can also be interpreted as finding an assignment that violates the minimum number of constraints (or maximizes the number of satisfied constraints). Many real-world problems can be formulated as MAX-CSP/WCSPs, including resource allocation problems [15], scheduling [7], bioinformatics [27, 120], combinatorial auctions [111, 34] or maximum satisfiability problems [26].

Example 2 *Figure 1.2 shows an example of a WCSP instance with binary variables. The cost functions are given in Figure 1.2(a). The value* ∞ *indicates a hard constraint. Figure*

1.2(b) depicts the constraint graph. The minimal cost solution of the problem is 5 and corresponds to the optimal assignment $(A = 0, B = 1, C = 1, D = 0, E = 1)$.

Maximum Satisfiability. Given a set of Boolean variables and a collection of clauses defined over subsets of variables, the goal of **maximum satisfiability** (MAX-SAT) is to find a truth assignment that violates the least number of clauses. If each clause is associated with a positive weight, the **weighted maximum satisfiability** (Weighted MAX-SAT) is to find a truth assignment such that the combined weight of the violated clauses is minimized.

Related Work on MAX-CSP/WCSP. MAX-CSP and WCSP can also be formulated using the semiring framework introduced by [9]. As an optimization version of constraint satisfaction, MAX-CSP/WCSP is NP-hard. A number of complete and incomplete algorithms have been developed for MAX-CSP/WCSP. Stochastic Local Search (SLS) algorithms, such as GSAT [110, 114], developed for Boolean Satisfiability and Constraint Satisfaction can be directly applied to MAX-CSP [123]. Since they are incomplete, SLS algorithms cannot guarantee an optimal solution, but they have been successful in practice on many classes of SAT and CSP problems. A number of search-based complete algorithms, using partial forward checking [49] for heuristic computation, have been developed [51, 58]. The Branch-and-Bound algorithms proposed by [65, 36] use bounded inference to compute the guiding heuristic function. More recently, [71, 25] introduced a family of depth-first Branch-and-Bound algorithms that maintain various levels of directional soft arc-consistency for solving WCSPs. The optimization method, called *Backtracking with Tree Decomposition* (BTD), developed by [59] uses a tree decomposition of the graphical model to capture the problem structure and guide the search more effectively.



(a) Directed acyclic graph (b) Moral graph

Figure 1.3: Belief network.

1.2.8 Belief Networks

Belief networks [104], also known as Bayesian networks, provide a formalism for reasoning about partial beliefs under conditions of uncertainty. They are defined by a directed acyclic graph over vertices representing random variables of interest ($e.g.$, the temperature of a device, the gender of a patient, a feature of an object, the occurrence of an event). The arcs can signify the existence of direct causal influences between linked variables quantified by conditional probabilities that are attached to each cluster of parents-child vertices in the network.

DEFINITION **12 (belief networks)** *A* belief network *is a graphical model* $\langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$*, where* $\mathbf{X} = \{X_1, ..., X_n\}$ *is a set of variables over multi-valued domains* $\mathbf{D} = \{D_1, ..., D_n\}$*. Given a directed acyclic graph* G *over* $\mathbf{X}$ *as nodes,* $\mathbf{P}_G = \{P_i\}$*, where* $P_i = \{P(X_i|pa(X_i))\}$ *are conditional probability tables (CPTs) associated with each variable* X_i*, and* $pa(X_i)$ *are the parents of* X_i *in the acyclic graph* G*. A belief network represents a joint probability distribution over* $\mathbf{X}$*,* $P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i|x_{pa(X_i)})$*. An evidence set* e *is an instantiated subset of variables. The primal graph of a belief network is called a* moral graph*.*

Example 3 *Figure 1.3(a) gives an example of a belief network over 6 variables and Figure 1.3(b) shows its moral graph. The example expresses the causal relationship between*

variables "Season" (A), "The configuration of an automatic sprinkler system" (B), "The amount of rain expected" (C), "The amount of manual watering necessary" (D), "The wetness of the pavement" (E) and "Whether or not the pavement is slippery" (F). The belief network expresses the probability distribution $P(A, B, C, D, E, F) = P(A) \cdot P(B|A) \cdot P(C|A) \cdot P(E|B, C) \cdot P(F|E)$.

The most popular optimization tasks for belief networks are defined below:

DEFINITION **13 (most probable explanation, maximum a posteriori hypothesis)** *Given a belief network* $\langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$*, the* most probable explanation (MPE) *task is to find a complete assignment which agrees with the evidence, and which has the highest probability among all such assignments. Namely to find an assignment* $x_1^o, ..., x_n^o$ *such that:*

$$P(x_1^o, ..., x_n^o) = max_{x_1,...,x_n} \prod_{k=1}^{n} P(x_k, e|x_{pa_k})$$

The more general query, called maximum a posteriori hypothesis (MAP)*, requires finding a maximum probability assignment to a subset of hypothesis variables, given the evidence.*

Both tasks arise in a wide variety of applications, such as probabilistic error correcting coding, speech recognition, medical diagnosis, airplane maintenance, monitoring and diagnosis in complex distributed computer systems, and so on. MPE queries are often used as ways of *completing* unknown information. For example, in probabilistic decoding, the task is to reconstruct a message ($e.g.$, a vector of bits) sent through a noisy channel, given the channel output. In speech recognition and image understanding, the objective is to find a sequence of objects ($e.g.$, letters, images) that is most likely to produce the observed sequence such as phonemes or pixel intensities. Yet another example is diagnosis, where the task is to reconstruct the hidden state of nature ($e.g.$, a set of possible diseases and unobserved symptoms the patient may have, or a set of failed nodes in a computer network) given observations of the test outcomes ($e.g.$, symptoms, medical tests, or network transactions results).

The general MAP queries are more applicable, used in cases such as medical diagnosis, when we observe part of the symptoms, and can accommodate some of the tests, and still wish to find the most likely assignments to the diseases only, rather than to both diseases and all unobserved variables. Although the MAP query is more general, MPE is an important special case because it is computationally simpler and thus should be applied when appropriate. It often serves as a *surrogate* task for MAP due to computational reasons. Since all the above problems can be posed as MPE or MAP queries, finding efficient algorithms clearly has a great practical value.

Related Work on MPE. It is known that solving the MPE task is NP-hard [22]. Complete algorithms use either the cycle cutset technique (also called conditioning) [104], the join-tree-clustering technique [115, 60], or the bucket-elimination scheme [31]. However, these methods work well only if the network is sparse enough to allow small cutsets or small clusters. The complexity of algorithms based on the cycle cutset idea is time exponential in the cutset size but require only linear space. The complexity of join-tree-clustering and bucket-elimination algorithms are both time and space exponential in the cluster size that equals the induced-width of the network's moral graph. Following Pearl's stochastic simulation algorithms [104], the suitability of Stochastic Local Search (SLS) algorithms for MPE was studied in the context of medical diagnosis applications [105] and more recently in [64, 102, 57]. Best-First search algorithms were proposed [116] as well as algorithms based on linear programming [113]. Some extensions are also available for the task of finding the k most-likely explanations [77, 118]. More recently, [65, 86] introduced a collection of Branch-and-Bound algorithms that use bounded inference, in particular the Mini-Bucket approximation [42], for computing a heuristic function that guides the search.

1.3 Search and Inference for Optimization in Graphical Models

It is convenient to classify algorithms that solve optimization problems in graphical models as either *search* ($e.g.$, depth-first Branch-and-Bound, best-first search) or *inference* ($e.g.$, variable elimination, join-tree clustering). Search is time-exponential in the number of variables, yet it can be accomplished in linear memory. Inference exploits the graph structure of the model and can be accomplished in time and space exponential in the treewidth of the problem. When the treewidth is big, inference must be augmented with search to reduce the memory requirements. We next overview these two classes of algorithms.

1.3.1 Bucket and Mini-Bucket Elimination

Bucket Elimination (BE) is a unifying algorithmic framework for dynamic programming algorithms applicable to probabilistic and deterministic reasoning [31]. Many algorithms for probabilistic inference, such as belief updating, finding the most probable explanation, finding the maximum a posteriori hypothesis, as well as algorithms for constraint optimization, such as MAX-CSP or WCSP, can be expressed as bucket elimination algorithms.

The input to the Bucket Elimination algorithm, described here by Algorithm 1, is an optimization problem, namely a collection of functions or relations ($e.g.$, clauses in propositional satisfiability, constraints or cost functions, or conditional probability tables for belief networks). For simplicity and without loss of generality we consider an optimization problem for which the elimination and combination operators are minimization and summation, respectively. Given a variable ordering, the algorithm partitions the functions into buckets, each associated with a single variable. A function is placed in the bucket of its argument that appears latest in the ordering. The algorithm has two phases. During the first, top-down phase, it processes each bucket, from last to first by a variable elimination procedure that computes a new function which is placed in a lower bucket. The variable elimination

Algorithm 1: BE: Bucket Elimination

Input: An optimization problem $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum, min \rangle$, ordering of the variables d.
Output: Minimal cost solution to $\mathcal{P}$ and the optimal assignment.
`// Initialize`
1 Partition the functions in $\mathbf{F}$ into $bucket_1$, ..., $bucket_n$, where $bucket_i$ contains all functions whose highest variable is X_i. Let $S_1, ..., S_j$ be the scopes of the functions (original and intermediate) in the processed bucket.
`// Backward`
2 **for** $p \leftarrow n$ *down-to* 1 **do**
3 Let $h_1, h_2, ..., h_j$ be the functions in $bucket_p$
4 **if** X_p *is instantiated* $(X_p = x_p)$ **then**
5 Assign $X_p = x_p$ to each h_i and put each resulting each into its appropriate bucket.
6 **else**
7 Generate the function $h^p : \ h^p = min_{X_p} \sum_{i=1}^{j} h_i$.
8 Add h^p to the bucket of the largest-index variable in $scope(h^p)$, where $scope(h^p) = \bigcup_{i=1}^{p} S_i - \{X_p\}$.
`// Forward`
9 Assign a value to each variable in the ordering d s.t. the combination of functions in each bucket is minimized.
10 **return** *the function computed in the bucket of the first variable and the optimizing assignment.*

procedure computes the sum of all cost functions and minimizes over the bucket's variable. During the second, bottom-up phase, the algorithm constructs a solution by assigning a value to each variable along the ordering, consulting the functions created during the top-down phase.

Bucket Elimination can be viewed as message passing from leaves to root along a bucket tree [66]. Let $\{B(X_1), ..., B(X_n)\}$ denote a set of buckets, one for each variable, along an ordering $d = (X_1, ..., X_n)$. A *bucket tree* of a graphical model $\mathcal{R}$ has buckets as its nodes. Bucket $B(X)$ is connected to bucket $B(Y)$ if the function generated in bucket $B(X)$ by BE is placed in $B(Y)$. The variables of $B(X)$, are those appearing in the scopes of any of its new and old functions. Therefore, in a bucket tree, every vertex $B(X)$ other than the root, has one parent vertex $B(Y)$ and possibly several child vertices $B(Z_1), ..., B(Z_t)$.

The structure of the bucket tree can also be extracted from the induced-ordered graph of $\mathcal{R}$ along d using the following definition.

DEFINITION **14 (bucket tree [31])** *Let G_d^* be the induced graph along d of a graphical model $\mathcal{R}$ whose primal graph is G. The vertices of the bucket tree are the n buckets each associated with a variable. Each vertex $B(X)$ points to $B(Y)$ (or, $B(Y)$ is the parent of $B(X)$) if Y is the latest neighbor of X that appear before X in G_d^*. Each variable X and its earlier neighbors in the induced graph are the variables of bucket $B(X)$. If $B(Y)$ is the*

Algorithm 2: `MBE(`i`)`: Mini-Bucket Elimination

Input: An optimization problem $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum, min \rangle$, ordering of the variables d, parameter i.
Output: Lower-bound on the minimal cost solution to $\mathcal{P}$, an assignment to all the variables, and the ordered augmented buckets.

`// Initialize`

Partition the functions in $\mathbf{F}$ into $bucket_1$, ..., $bucket_n$, where $bucket_i$ contains all functions whose highest variable is X_i. Let $S_1, ..., S_j$ be the scopes of the functions (original and intermediate) in the processed bucket.

`// Backward`

for $p \leftarrow n$ *down-to* 1 **do**
    Let $h_1, h_2, ..., h_j$ be the functions in $bucket_p$
    if X_p *is instantiated* $(X_p = x_p)$ **then**
        Assign $X_p = x_p$ to each h_i and put each resulting each into its appropriate bucket.
    else
        Generate an i-partitioning $Q' = \{Q_1, ..., Q_t\}$.
        foreach $Q_l \in Q'$ **do**
            Let $h_{l_1}, ..., h_{l_t}$ be the functions in Q_l.
            Generate the function $h^l : \ h^l = min_{X_p} \sum_{i=1}^{t} h_{l_i}$.
            Add h^l to the bucket of the largest-index variable in $scope(h^l)$, where $scope(h^l) = \bigcup_{i=1}^{t} scope(h_{l_i}) - \{X_p\}$.

`// Forward`

Assign a value to each variable in the ordering d s.t. the combination of functions in each bucket is minimized.
return *the function computed in the bucket of the first variable and the optimizing assignment.*

parent of $B(X)$ in the bucket tree, then the separator of X and Y is the set of variables appearing in $B(X)$ $B(Y)$, denoted $sep(X, Y)$.

THEOREM **1 (complexity [31])** *The time and space complexity of bucket elimination applied along order d is $O(r \cdot k^{(w^*+1)})$ and $O(n \cdot k^{w^*})$ respectively, where w^* is the induced width of the primal graph along the ordering d, r is the number of functions, n is the number of variables and k bounds the domain size.*

The main drawback of bucket elimination algorithms is that they require too much space for storing intermediate functions. *Mini-Bucket Elimination* (MBE) is an approximation designed to avoid the space and time problem of full bucket elimination [42] by partitioning large buckets into smaller subsets, called *mini-buckets* which are processed independently. Here is the rationale. Let $h_1, ..., h_j$ be the functions in $bucket_p$. When *Bucket Elimination* processes $bucket_p$, it computes the function h^p: $h^p = min_{X_p} \sum_{i=1}^{j} h_i$, where $scope(h^p) = \cup_{i=1}^{j} S_i - \{X_p\}$. The *Mini-Bucket* algorithm, on the other hand, creates a partition $Q' = \{Q_1, ..., Q_t\}$ where the mini-bucket Q_l contains the functions $h_{l_1}, ..., h_{l_t}$. The approximation processes each mini-bucket (by using the combination and elimination

operators) separately, therefore computing $g^p = \sum_{l=1}^{t} min_{X_p} \sum_{i=1}^{t} h_{l_i}$. Clearly, g^p is a lower bound on h^p, namely $g^p \leq h^p$ (for maximization, g^p is an upper bound). Therefore, the bound computed in each bucket yields an overall bound on the cost of the solution.

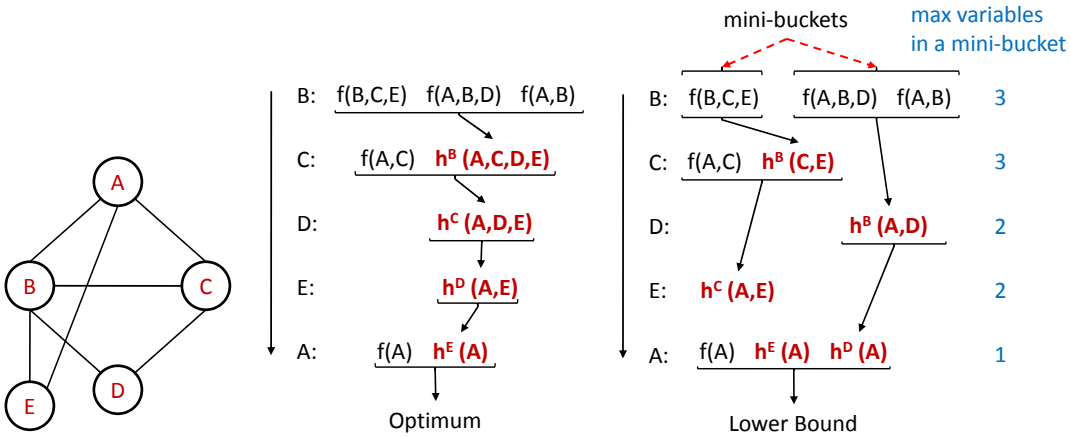
The quality of the bound depends on the degree of partitioning into mini-buckets. Given a bounding parameter i (called here i-bound), the algorithm creates an i-partitioning, where each mini-bucket includes no more than i variables. Algorithm MBE(i), described by Algorithm 2, is parameterized by this i-bound. It outputs not only a lower bound on the cost of the optimal solution and an assignment, but also the collection of the augmented buckets. By comparing the bound computed by MBE(i) to the cost of the assignment output by MBE(i), we can always have an interval bound on the error for that given instance. For example, if MBE(i) provides a lower bound on the optimal assignment in its first bucket, while the cost of the assignment generated yields an upper bound.

The complexity of the algorithm is time and space $O(exp(i))$ where $i < n$. When the i-bound is large enough ($i.e.$, $i \geq w^*$), the Mini-Bucket algorithm coincides with full bucket elimination. In summary,

THEOREM **2 (complexity [42])** *Algorithm MBE(i) generates an interval bound of the optimal solution, and its time and space complexity are $O(r \cdot k^i)$ and $O(r \cdot k^{i-1})$ respectively, where r is the number of functions and k bounds the domain size.*

Example 4 *Figures 1.4(b) and 1.4(c) illustrate how algorithms BE and MBE(i) for $i = 3$ process the cost network in Figure 1.4(a) along the ordering (A, E, D, C, B). We assume a minimization task.*

Algorithm BE records the new functions $h^B(A, C, D, E)$, $h^C(A, D, E)$, $h^D(A, E)$, and $h^E(A)$. Then, in the bucket of A, it computes the cost of the optimal solution, $opt = min_A(f(A) + h^E(A))$. Subsequently, an optimal assignment $(A = a_0; B = b_0; C = c_0; D = d_0; E = e_0)$ is computed for each variable from A to B by selecting a value that minimizes the sum of functions in the corresponding bucket, conditioned on the previously



(a) Cost network (b) Bucket elimination (BE) (c) Mini-Bucket Elimination (MBE(3))

Figure 1.4: Execution of BE and MBE(i).

assigned values. Namely, $a_0 = argmin_A(f(A) + h^E(A))$, $e_0 = argmin_E h^D(a_0, E)$, $d_0 = argmin_D f(a_0, D, e_0)$, *and so on.*

*The approximation MBE(*3*) splits bucket* B *into two mini-buckets, each containing no more than 3 variables, and generates* $h^B(C, E)$ *and* $h^B(A, D)$. *A lower bound on the optimal value is computed by* $L = min_A(f(A) + h^E(A) + h^D(A))$. *A suboptimal tuple is computed by MBE(*3*) similarly to the optimal tuple computed by BE, by assigning a value to each variable that minimizes the sum of cost functions in the corresponding bucket, given the assignments to the previous variables. The value of this assignment is an upper bound on the optimal value.*

1.3.2 Systematic Search with Mini-Bucket Heuristics

Most exact search algorithms for solving optimization problems in graphical models follow a *Branch-and-Bound* schema. These algorithms perform a depth-first traversal on the search tree defined by the problem, where internal nodes represent partial assignments and leaf nodes stand for complete ones. Throughout the search, Branch-and-Bound maintains a global bound on the cost of the optimal solution, which corresponds to the cost of the best full variable instantiation found thus far. At each node, the algorithm computes a heuristic

estimate of the best solution extending the current partial assignment and prunes the respective subtree if the heuristic estimate is not better than the current global bound (that is - not greater for maximization problems, not smaller for minimization problems).

The algorithm requires only a limited amount of memory and can be used as an anytime scheme, namely whenever interrupted, it outputs the best solution found so far.

The effectiveness of the Branch-and-Bound search method depends on the quality of the heuristic function. Therefore, one of the most important issues in heuristic search is obtaining a good heuristic function. Often there is a trade-off between the quality of the heuristic and the complexity of its computation. In the following section we will provide an overview of a general scheme for generating heuristic estimates automatically from the functional specification of the problem, based on the Mini-Bucket approximation.

Mini-Bucket Heuristics

The idea was first introduced in [65] and showed that the intermediate functions recorded by the Mini-Bucket algorithm can be used to assemble a heuristic function that estimates the cost of the completion of any partial assignment to a full solution, and therefore can serve as an evaluation function that can guide search. The following definition summarizes an automatic procedure that can generate heuristic functions for any partial assignment.

DEFINITION **15 (mini-bucket heuristic [65])** *Given an ordered set of augmented buckets generated by the Mini-Bucket algorithm MBE(*i*) along the ordering* $d = (X_1, ..., X_p, ..., X_n)$ *and given a partial assignment* $\bar{x}^p = (x_1, ..., x_p)$*, the heuristic function* $h(\bar{x}^p)$ *is defined as the combination of all the intermediate* h_j^k *function that satisfy the following two properties:*

1. *They are generated in buckets* $p + 1$ *through* n*,*
2. *They reside in buckets* 1 *through* p*.*

Following [65], consider for illustration the cost network shown in Figure 1.4(a), and consider a given variable ordering $d = (A, E, D, C, B)$ and the bucket and mini-buckets

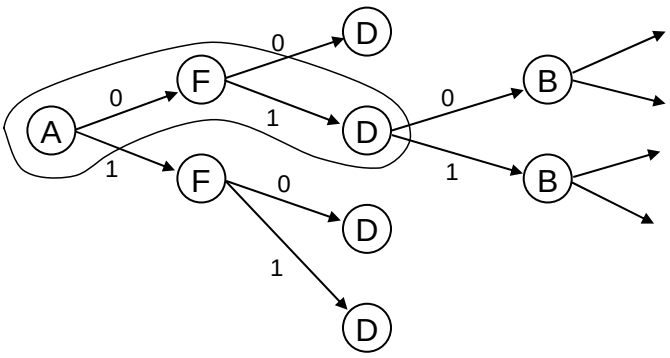


Figure 1.5: Search space for $f^*(a, e, d)$ [65].

configuration in the output, as displayed in Figures 1.4(b) and 1.4(b), respectively. Let us assume, without loss of generality, that variables A, E and D have been instantiated during search (see Figure 1.5, $a = 0$, $e = 1$, $d = 1$). Let $f^*(a, e, d)$ be the cost of the best completion of the partial assignment $(A = a, E = e, D = d)$. By definition,

$$\begin{aligned} f^*(a,e,d) &= min_{b,c}(f(a) + f(a,c) + f(a,b) + f(b,c,e) + f(a,b,d)) \\ &= f(a) + min_{b,c}(f(a,c) + f(a,b) + f(b,c,e) + f(a,b,d)) \\ &= g(a,e,d) + h^*(a,e,d) \end{aligned}$$

where

$$\begin{aligned} g(a,e,d) &= f(a) \\ h^*(a,e,d) &= min_{b,c}(f(a,c) + f(a,b) + f(b,c,e) + f(a,b,d)) \end{aligned}$$

We can derive:

$$\begin{aligned} h^*(a,e,d) &= min_{b,c}(f(a,c)+f(a,b)+f(b,c,e)+f(a,b,d)) \\ &= min_c(f(a,c)+min_b(f(a,b)+f(b,c,e)+f(a,b,d)) \\ &= min_c(f(c,e)+h^B(a,d,c,e)) \\ &= h^C(a,d,e) \end{aligned}$$

where

$$\begin{aligned} h^B(a,c,d,e) &= min_b(f(b,c,e)+f(a,b,d)+f(a,b)) \\ h^C(a,d,e) &= min_c(f(a,c)+h^B(a,c,d,e)) \end{aligned}$$

Interestingly, the functions $h^B(a,c,d,e)$ and $h^C(a,d,e)$ are already produced by the bucket elimination algorithm BE (see Figure 1.4(b)). Specifically, the function $h^B(a,c,d,e)$, generated in $bucket_B$, is the result of a minimization operation over variable B. In practice, however, this function may be too hard to compute as it requires processing a function on four variables and recording a function on three variables. So, it can be replaced by an approximation, where the minimization is split into two parts. This yields a function, which we denote $h(a,e,d)$, that is a *lower bound* on $h^*(a,e,d)$, namely,

$$\begin{aligned} h^*(a,e,d) &= min_c(f(a,c)+min_b(f(b,c,e)+f(a,b)+f(a,b,d))) \\ &\geq min_c(f(a,c)+min_b f(b,c,e)+min_b(f(a,b)+f(a,b,d))) \\ &= min_c(f(a,c)+h^B(c,e)+h^B(a,d)) \\ &= h^B(a,d)+min_c(f(a,c)+h^B(c,e)) \\ &= h^B(a,d)+h^C(a,e) \\ &\triangleq h(a,e,d) \end{aligned}$$

where

$$\begin{aligned} h^B(c,e) &= min_b f(b,c,e) \\ h^B(a,d) &= min_b (f(a,b) + f(a,b,d)) \\ h^C(a,e) &= min_c (f(a,c) + h^B(c,e)) \end{aligned}$$

Notice now that the functions $h^B(c,e)$, $h^B(a,d)$ and $h^C(a,e)$ were already computed by the mini-bucket algorithm MBE(i) (see Figure 1.4(c)). Using the lower bound function $h(a,e,d)$, we can now define the function $f(a,e,d)$ that provides a lower bound on the exact value $f^*(a,e,d)$. Namely, replacing $h^*(a,e,d)$ by $h(a,e,d)$ in $f^*(a,e,d)$, we get:

$$f(a,e,d) = g(a,e,d) + h(a,e,d) \leq f^*(a,e,d)$$

It was shown that:

THEOREM **3 (monotonicity and admissibility [65])** *For every partial assignment $\bar{x}_p = (x_1, ..., x_p)$ of the first p variables, the heuristic function $h(\bar{x}_p)$ is admissible and monotonic.*

We next elaborate on how to use the mini-bucket heuristic to guide depth-first Branch-and-Bound and best-first search algorithms. These algorithms were first presented in [65].

The tightness of the bound generated by the Mini-Bucket approximation depends on its i-bound. Larger values of i generally yield better bounds, but require more computation. Since the Mini-Bucket algorithm is parameterized by i, when using the heuristics in each of the search methods, we get an entire class of Branch-and-Bound search and Best-First search algorithms that are parameterized by i and which allow a controllable trade-off between preprocessing and search, or between heuristic strength and its overhead.

Algorithms 3 and 4 show Branch-and-Bound with Mini-Bucket heuristics (BBMB(i)) and Best-First search with Mini-Bucket heuristics (BFMB(i)), developed in [65]. Both algorithms have a preprocessing step of running the Mini-Bucket algorithm that produces

Algorithm 3: BBMB(i): Branch-and-Bound search with Mini-Bucket heuristics

Input: An optimization problem $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum, min \rangle$, variable ordering $d = (X_1, ..., X_p, ..., X_n)$, time limit.

Output: Minimal cost solution to $\mathcal{P}$.

1 INITIALIZE: Run MBE(i) algorithm generating a set of ordered augmented buckets and a lower bound on the minimal cost. Initialize global upper bound UB to ∞. Set p to 0.

2 SEARCH: Execute the following procedure until variable X_1 has no legal values left or until out of time, in which case output the current best solution

3 EXPAND: Given a partial instantiation $\bar{x}^p$, compute all partial assignments $\bar{x}^{p+1} = (\bar{x}^p, v)$ for each value v of X_{p+1}. For each node $\bar{x}^{p+1}$ compute its heuristic value $f(\bar{x}^{p+1}) = g(\bar{x}^{p+1}) + h(\bar{x}^{p+1})$. Discard those assignments for which $f(\bar{x}^{p+1}) \geq UB$. Add the remaining assignments to the search tree as children of $\bar{x}^p$.

4 FORWARD: If X_{p+1} has no legal values left, goto BACKTRACK. Otherwise, let $\bar{x}^{p+1} = (\bar{x}^p, v)$ be the best extension to $\bar{x}^p$ according to f. If $p + 1 = n$, then set $UB = f(\bar{x}^n)$ and goto BACKTRACK. Otherwise remove v from the list of legal values. Set $p = p + 1$ and goto EXPAND.

5 BACKTRACK: If $p = 1$, Exit. Otherwise set $p = p - 1$ and repeat from the FORWARD step.

a set of ordered augmented buckets.

Branch-and-Bound Search with Mini-Bucket Heuristics

BBMB(i) traverses the search space in a depth-first manner, instantiating variables from first to last. Throughout the search, the algorithm maintains a global bound on the cost of the optimal solution, which corresponds to the cost of the best full variable instantiation found thus far. When the algorithm processes variable X_p, all the variables preceding X_p in the ordering are already instantiated, so it can compute $f(\bar{x}^{p-1}, X_p = v) = g(\bar{x}^{p-1}, v) + h(\bar{x}^{p-1}, v)$ for each extension $X_p = v$. The algorithm prunes all values v whose heuristic evaluation function $f(\bar{x}^{p-1}, X_p = v)$ is greater or equal than the current upper bound, because such a partial assignment $(x_1, ..., x_{p-1}, v)$ cannot be extended to an improved full assignment. The algorithm assigns the best value v to variable X_p and proceeds to variable X_{p+1}, and when variable X_p has no values left, it backtracks to variable X_{p-1}. Search terminates when it reaches a time bound or when the first variable has no values left. In the

Algorithm 4: BFMB(i): Best-First search with Mini-Bucket heuristics

Input: An optimization problem $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum, min \rangle$, variable ordering $d = (X_1, ..., X_p, ..., X_n)$, time limit.

Output: Minimal cost solution to $\mathcal{P}$.

1 INITIALIZE: Run MBE(i) algorithm generating a set of ordered augmented buckets and a lower bound on the minimal cost. Insert a dummy node $\bar{x}_0$ in the set L of open nodes.

2 SEARCH:

3 If out of time, output Mini-Bucket assignment

4 Select and remove a node $\bar{x}^p$ with the best heuristic value $f(\bar{x}^p)$ from the set of open nodes L.

5 If $p = n$ then $\bar{x}^p$ is an optimal solution. Exit.

6 Expand $\bar{x}^p$ by computing all child nodes $(\bar{x}^p, v)$ for each value v in the domain of X_{p+1}. For each node $\bar{x}^{p+1}$ compute its heuristic value $f(\bar{x}^{p+1}) = g(\bar{x}^{p+1}) + h(\bar{x}^{p+1})$.

7 Add all nodes $(\bar{x}^p, v)$ to L and goto SEARCH.

latter case, the algorithm has found an optimal solution.

Best-First Search with Mini-Bucket Heuristics

Algorithm BFMB(i) maintains a list of open nodes. Each node corresponds to a partial assignment $\bar{x}^p$ and has an associated heuristic value $f(\bar{x}^p)$. The basic step of the algorithm consists of selecting an assignment $\bar{x}^p$ from the list of open nodes having the best heuristic value (that is - the highest value for maximization problems; the smallest value for minimization problems) $f(\bar{x}^p)$, expanding it by computing all partial assignments $(\bar{x}^p, v)$ for all values v of X_{p+1}, and adding them to the list of open nodes.

Since the generated mini-bucket heuristics are admissible and monotonic, their use within Best-First search yields A^* type algorithms whose properties are well understood. The algorithm is guaranteed to terminate with an optimal solution. When provided with more powerful heuristics, it explores a smaller search space, but otherwise it requires substantial space. It is known that Best-First search algorithms are optimal. Namely, when given the same heuristic information, Best-First search is the most efficient algorithm in terms of the size of the search space it explores [40]. In particular, Branch-and-Bound will

expand any node that is expanded by Best-First search (up to some tie breaking conditions), and in many cases it explores a larger space. Still, Best-First search may occasionally fail because of its memory requirements, or because it has to maintain a large subset of open nodes during search, or because of tie breaking rules at the last frontier of nodes having evaluation function value that equals the optimal solution.

Impact of the Mini-Bucket i-bound

For any accuracy parameter i, we can determine the space complexity of Mini-Bucket preprocessing in advance. This can be done by computing signatures ($i.e.$, arguments) of all intermediate functions, without computing the actual functions. Based on the signatures of original and intermediate functions, we can compute the total space needed. Knowing the space complexity, we can estimate the time complexity. Thus given the time and space at our disposal, we can select the parameter i that would fit. However, the cost-effectiveness of the heuristic produced by Mini-Bucket preprocessing may not be predicted a priori. It was observed [65] that in general, as the problem graph is more dense, higher levels of Mini-Bucket heuristic become more cost-effective.

1.3.3 Branch-and-Bound Search for Weighted CSP

The area of Weighted CSP has seen substantial advances in the last years by exploiting and extending local consistency to cost functions, called *soft local consistency*. Several increasingly stronger local consistency algorithms were introduced (soft node and arc consistency [72], full directional arc consistency (DAC/FDAC) [71], existential directional arc consistency (EDAC) [25]), leading to increasingly efficient Branch-and-Bound algorithms.

As in the classical case of constraint propagation, enforcing soft local consistency on the initial problem provides, in polynomial time, an *equivalent* problem defining the same cost distribution on complete assignments, with possible smaller domains. It also produces a lower bounding heuristic evaluation function that can be exploited during search. Con-

sequently, the OR Branch-and-Bound algorithm maintaining FDAC/EDAC during search introduced recently in [71, 72, 25] was demonstrated to be one of the best performing algorithms for solving binary WCSPs.

Review of Local Consistency for Weighted CSPs

As in the classical CSP, enforcing soft local consistency on the initial problem provides in polynomial time an *equivalent* problem defining the same cost distribution on complete assignments, with possible smaller domains [71, 72, 25].

Assume a binary Weighted CSP $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$, where $\mathbf{X} = \{X_1, ..., X_n\}$ and $\mathbf{D} = \{D_1, ..., D_n\}$ are the variables and their corresponding domains. $\mathbf{C}$ is the set of binary and unary cost functions (or soft constraints). A binary soft constraint $C_{ij}(X_i, X_j) \in \mathbf{C}$ (or C_{ij} in short) is $C_{ij}(X_i, X_j) \ : \ D_i \times D_j \rightarrow \mathbb{N}$. A unary soft constraint $C_i(X_i) \in \mathbf{C}$ (or C_i in short) is $C_i(X_i) \ : \ D_i \rightarrow \mathbb{N}$. We assume the existence of a unary constraint $C_i(X_i)$ for every variable, and a zero-arity constraint, denoted by $C_\emptyset$. If no such constraints are defined, we can always define dummy ones, as $C_i(x_i) = 0, \forall x_i \in D_i$ or $C_\emptyset = 0$. We denote by $\top$, the maximum allowed cost ($e.g.$, $\top = \infty$). The cost of a tuple $\bar{x} = (x_1, ..., x_n)$, denoted by $cost(\bar{x})$, is defined by:

$$cost(\bar{x}) = \sum_{C_{ij} \in \mathbf{C}} C_{ij}(\bar{x}[i, j]) + \sum_{C_i \in \mathbf{C}} C_{X_i}(\bar{x}[i]) + C_\emptyset$$

For completeness, we define next some local consistencies in WCSP, in particular *node*, *arc* and *directional arc consistency*, as in [71, 72]. We assume that the set of variables $\mathbf{X}$ is totally ordered. We note that there are several stronger local consistencies which were defined in recent years, such as *full directional arc consistency* (FDAC) [71, 72] or *existential directional arc consistency* (EDAC) [25].

DEFINITION **16 (soft node consistency [71, 72])** *Let $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ be a binary WCSP. (X_i, x_i) is star node consistent (NC^*) if $C_\emptyset + C_i(x_i) < \top$. Variable X_i is NC^* if: i) all*

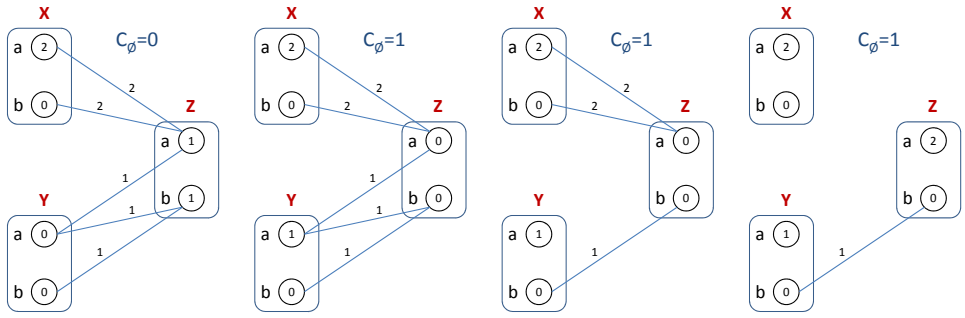
its values are NC^* and ii) there exists a value $x_i \in D_i$ such that $C_i(x_i) = 0$. Value x_i is a support *for variable X_i. $\mathcal{R}$ is NC^* if every variable is NC^*.*

DEFINITION **17 (soft arc consistency [71, 72])** *Let $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ be a binary WCSP. (X_i, x_i) is arc consistent (AC) with respect to constraint C_{ij} if there exists a value $x_j \in D_j$ such that $C_{ij}(x_i, x_j) = 0$. Value x_j is called a* support *for the value x_i. Variable X_i is AC if all its values are AC wrt. every binary constraint affecting X_i. $\mathcal{R}$ is AC^* if every variable is AC and NC^*.*

DEFINITION **18 (soft directional arc consistency [71, 72])** *Let $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ be a binary WCSP. (X_i, x_i) is directional arc consistent (DAC) with respect to constraint C_{ij}, $i < j$, if there exists a value $x_j \in D_j$ such that $C_{ij}(x_i, x_j) + C_j(x_j) = 0$. Value x_j is called a* full support *of x_j. Variable X_i is DAC is all its values are DAC wrt. every C_{ij}, $i < j$. $\mathcal{R}$ is DAC^* if every variable is DAC and NC^*.*

For our purpose, we point out that enforcing such local consistencies is done by the repeated application of atomic operations called *arc equivalence preserving transformations* [20]. This process may increase the value of $C_\emptyset$ and the unary costs $C_i(x_i)$ associated with domain values. The zero-arity cost function $C_\emptyset$ defines a *strong lower bound* which can be exploited by Branch-and-Bound algorithms while the updated $C_i(x_i)$ can inform variable and value orderings [71, 72, 25].

If we consider two cost functions $C_{ij}(X_i, X_j)$, defined over variables X_i and X_j, and $C_i(X_i)$, defined over variable X_i, a value $x_i \in D_i$ and a cost α, we can add α to $C_i(x_i)$ and subtract α from every $C_{ij}(x_i, x_j)$ for all $x_j \in D_j$. Simple arithmetic shows that the global cost distribution is unchanged while costs may have moved from the binary to the unary level (if $\alpha > 0$, this is called a *projection*) or from the unary to the binary level (if $\alpha < 0$, this is called an *extension*). In these operations, any cost above $\top$, the maximum allowed cost, can be considered as infinite and is thus unaffected by subtraction. If no negative cost appears and if all costs above $\top$ are set to $\top$, the remaining problem is always a valid and



(a) WCSP (b) NC^* (c) DAC^* (d) AC^*

Figure 1.6: Four equivalent WCSPs (for $\top = 4$) [71].

equivalent WCSP. The same mechanism, at the unary level, can be used to move costs from the $C_i(X_i)$ to $C_\emptyset$. Finally, any value such that $C_i(x_i) + C_\emptyset$ is equal to $\top$ can be deleted. For a detailed description of these operations, we refer the reader to [71, 72, 25].

Example 5 *Figure 1.6(a) shows a WCSP with a sets of costs* $[0, ..., 4]$ *and with* $\top = 4$. *The network has three variables* $\mathbf{X} = \{X, Y, Z\}$, *each with values* $\{a, b\}$. *There are 2 binary constraints* $C(X, Z)$, $C(Y, Z)$ *and two non-trivial unary constraints* $C(X)$ *and* $C(Z)$. *Unary costs are depicted inside their domain value. Binary costs are depicted as labeled edges connecting the corresponding pair of values. Zero costs are not shown. Initially,* $C_\emptyset$ *is set to 0. One optimal solution is* $(X = b, Y = b, Z = b)$, *with cost 2.*

The problem in Figure 1.6(a) is not NC^* *since* Z *has no support. To enforce* NC^* *we must force a support for* Z *by projecting* $C_Z(Z)$ *onto* $C_\emptyset$. *The resulting problem in Figure 1.6(b) is* NC^* *but not* AC^*. *To enforce* AC^*, *it suffices to enforce a support for* (Y, a) *and* (Z, a), *as follows: we project* $C_{YZ}(Y, Z)$ *over* (Y, a) *by adding 1 to* $C_Y(a)$ *and subtracting 1 from* $C_{YZ}(a, a)$ *and* $C_{YZ}(a, b)$, *and similarly project* $C_{XZ}(X, Z)$ *over* (Z, a). *Consequently, we get problem 1.6(d) which is* AC^*. *Observe also that problem 1.6(b) is not* DAC^* *for order* (X, Y, Z) *since* (Y, a) *has no full support on* Z. *Problem 1.6(c) is an equivalent* DAC^* *problem.*

There is a strong relation between directional arc consistency and mini-buckets. It was shown in [71] that given a WCSP with $\top = \infty$, and a variable ordering, the lower bound in-

duced by mini-buckets involving at most 2 variables is the same as the lower bound induced by $C_\emptyset$ after the problem is made directional arc consistent. However, the mini-bucket computation provides only a lower bound while DAC enforcing provides both a lower bound and a directional arc consistent equivalent problem. All the work done to compute the lower bound is captured in this problem which offers the opportunity to perform incremental updates of the lower bound.

Chapter 2

Systematic versus Non-systematic Search for Most Probable Explanations

2.1 Introduction

As noted in Chapter 1, the Most Probable Explanation (MPE) task in Bayesian networks arises in a wide variety of applications, such as probabilistic error correcting coding, speech recognition, genetic linkage analysis, medical diagnosis, airplane maintenance, monitoring and diagnosis in complex distributed computer systems. For example, in probabilistic decoding, the task is to reconstruct a message ($e.g.$, a vector of bits) sent through a noisy channel, given the channel output. In speech recognition and image understanding, the objective is to find a sequence of objects ($e.g.$, letters, images) that is most likely to produce the observed sequence such as phonemes or pixel intensities.

A Branch-and-Bound with mini-bucket heuristics (BBMB(i)) was introduced in [65] (see also Chapter 1). The algorithm uses the functions generated by Mini-Bucket Elimination MBE(i) in a pre-processing step to create a heuristic function that guides the search.

In this chapter we introduce algorithm BBBT(i) [36] which explores the feasibility of generating the mini-bucket heuristics *during search*, rather than in a *pre-processing manner*. This can yield more accurate heuristics and allow dynamic variable ordering - a feature that can have tremendous effect on search. The dynamic generation of these heuristics is facilitated by an extension of Mini-Bucket Elimination to Mini-Bucket Tree Elimination

(MBTE(i)), a partition-based approximation defined over cluster-trees described in [36]. MBTE(i) outputs multiple (lower or upper) bounds for each possible variable and value extension at once, which is much faster than running MBE(i) n times, one for each variable, to generate the same result. BBBT(i) applies the MBTE(i) heuristic computation at each node of the search tree. Clearly, the algorithm has a much higher time overhead compared with BBMB(i) for the same i-bound, but it can prune the search space much more effectively, hopefully yielding overall superior performance for some classes of hard problems. Preliminary tests of the algorithms for the MAX-CSP task in constraint satisfaction showed that, on a class of hard enough problems, BBBT(i) with the smallest i-bound (i=2) is cost-effective [36]. In this chapter we extend the algorithm to solving the MPE task over Bayesian networks. We will compare the performance of BBBT(i) and BBMB(i) against two incomplete schemes of stochastic local search and belief propagation adapted to optimization tasks.

Stochastic Local Search (SLS) is a class of incomplete approximation algorithms which, unlike complete algorithms, are not guaranteed to find an optimal solution, but as shown during the last decade, are often far superior to complete systematic algorithms on CSP and SAT problems. Some of these SLS algorithms are applied directly on the Bayesian network and some translate the problem into a weighted SAT problem first and then apply a weighted MAX-SAT algorithm.

The Iterative Join-Graph Propagation (IJGP(i)) methods apply Pearl's belief propagation algorithm to loopy join-graphs of the belief network [39].

Contribution

The contribution of this chapter is an extensive empirical study of highly competitive approaches for solving the MPE task in Bayesian networks introduced in recent years. We compare two depth-first Branch-and-Bound algorithms, BBBT(i) and BBMB(i), that exploit bounded inference for heuristic guidance on the one hand, against some of the best-

known incomplete approximation algorithms, such as SLS and generalized iterative belief propagation adapted for the MPE task, on the other.

Our empirical results on various random and real-world benchmarks show that BBMB(i) and BBBT(i) do not dominate one another. While BBBT(i) can sometimes significantly improve over BBMB(i), in many other instances its (quite significant) pruning power does not outweigh its time overhead. Both algorithms are powerful in different cases. In general when large i-bounds are effective BBMB(i) is more powerful, however when space is at issue BBBT(i) with small i-bound is often more powerful. We also show that SLS algorithms are overall inferior to BBBT(i) and BBMB(i), except when the domain size is small. The superiority of BBBT(i) and BBMB(i) is especially significant because unlike local search they can prove optimality if given enough time. Finally, we demonstrate that generalized belief propagation algorithms are often superior to the SLS class as well.

The research presented in this chapter is based in part on [86].

Chapter Outline

The chapter is organized as follows. Section 2.2 presents relevant recent work on cluster-tree and mini-cluster elimination underlying the Branch-and-Bound algorithms. Section 2.3 illustrates the process of using bounded inference to guide Branch-and-Bound search. Section 2.4 overviews current state-of-the-art non-systematic search approaches for solving the MPE task. In Section 2.5 we provide our experimental results, while Section 2.6 provides concluding remarks.

2.2 Background

DEFINITION **19 (cluster-tree decomposition [50])** *Let $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, G, \mathbf{P} \rangle$ be a belief network. A* cluster-tree decomposition *for $\mathcal{B}$ is a triple $\langle T, \chi, \psi \rangle$, where $T = (\mathbf{V}, \mathbf{E})$ is a tree, and χ and ψ are labeling functions which associate with each vertex $v \in \mathbf{V}$ two sets,*

$\chi(v) \subseteq \mathbf{X}$ *and* $\psi(v) \subseteq \mathbf{P}$.

1. *For each function* $P_i \in \mathbf{P}$*, there is* exactly *one vertex* $v \in \mathbf{V}$ *such that* $p_i \in \psi(v)$*, and* $scope(p_i) \subseteq \chi(v)$.

2. *For each variable* $X_i \in \mathbf{X}$*, the set* $\{v \in \mathbf{V} | X_i \in \chi(v)\}$ *induces a connected subtree of* T *(the running intersection property).*

Let (u, v) *be an edge of a cluster-tree decomposition, the* separator *of* u *and* v *is defined as* $sep(u, v) = \chi(u) \cap \chi(v)$*; the* eliminator *of* u *and* v *is defined as* $elim(u, v) = \chi(u) - sep(u, v)$. *The* hyperwidth *is* $hw = \max_{v \in \mathbf{V}} |\psi(v)|$.

In addition to finding the global optimum ($i.e.$, MPE), of particular interest to us is the special case of finding, for each assignment $X_i = x_i$, the highest probability of the complete assignment that agrees with $X_i = x_i$. Formally,

DEFINITION **20 (singleton optimization)** *Given a belief network* $\langle \mathbf{X}, \mathbf{D}, \mathbf{P}, \Pi \rangle$, singleton optimization *is the task of finding the best cost extension to every singleton tuple* (X_i, x_i)*, namely finding* $z(X_i) = \max_{\mathbf{X} - \{X_i\}} (\prod_{k=1}^{n} P_k)$ *for each variable* $X_i \in \mathbf{X}$.

The common exact algorithms for Bayesian inference are join-tree clustering defined over tree decompositions [73] and variable elimination algorithms [32]. The variant we use was presented for constraint networks [36] and we overview it next.

2.2.1 Cluster-Tree Elimination

Algorithm Cluster-Tree Elimination (CTE) [36] provides a unifying space conscious description of join-tree clustering algorithms. It is a message-passing scheme that runs on the cluster-tree decomposition, well-known for solving a wide range of automated reasoning

Procedure CTE
Input: A Bayesian network BN, a cluster-tree decomposition $\langle T, \chi, \psi \rangle$.
Output: A set of functions z_i as a solution to the singleton-optimality task.
Repeat
1. Select an edge (u, v) such that $m_{(u,v)}$ has not been computed and u has received messages
from all adjacent vertices other than v.
2. $m_{(u,v)} \leftarrow \max_{elim(u,v)} \prod_{g \in cluster(u), g \neq m_{(v,u)}} g$ (where $cluster(u) = \psi(u) \cup \{m_{(w,u)} | (w, u) \in T\}$).
Until all messages have been computed.
Return for each i, $z(X_i) = \max_{\chi(u)-X_i} \prod_{g \in cluster(u)} g$, such that $X_i \in cluster(u)$.

Figure 2.1: Algorithm cluster-tree elimination (CTE) for singleton-optimality task.

problems. We will briefly describe its partition-based mini-clustering approximation that forms the basis for our heuristic generation scheme.

CTE provided in Figure 2.1 computes a solution to the singleton functions $z(X_i)$ in a Bayesian network. It works by computing *messages* that are sent along edges in the tree. Message $m_{(u,v)}$ sent from vertex u to vertex v, can be computed as soon as all incoming messages to u other than $m_{(v,u)}$ have been received. As leaves compute their messages, their adjacent vertices also qualify and computation goes on until all messages have been computed. The set of functions associated with a vertex u augmented with the set of incoming messages is called a *cluster*, $cluster(u) = \psi(u) \cup_{(w,u) \in T} m_{(w,u)}$. A message $m_{(u,v)}$ is computed as the product of all functions in $cluster(u)$ excluding $m_{(v,u)}$ and the subsequent elimination of variables in the eliminator of u and v. Formally, $m_{(u,v)} = \max_{elim(u,v)} (\prod_{g \in cluster(u), g \neq m_{(v,u)}} g)$. The computation is done by enumeration, recording only the output message. The algorithm terminates when all messages are computed. The functions $z(X_i)$ can be computed in any cluster that contains X_i by eliminating all variables other than X_i.

It was shown [36] that the complexity of CTE is time $O(r \cdot (hw + dg) \cdot d^{tw+1})$ and space $O(r \cdot d^s)$, where r is the number of vertices in the cluster-tree decomposition, hw is the hyperwidth, dg is the maximum degree ($i.e.$, number of adjacent vertices) in the tree, tw

is the treewidth, d is the largest domain size and s is the maximum separator size. This assumes that step 2 is computed by enumeration.

There is a variety of ways in which a cluster-tree decomposition can be obtained. We will choose a particular one called *bucket-tree decomposition*, inspired by viewing the Bucket Elimination algorithm as message passing along a tree [36]. Since a bucket-tree is a special case of a cluster-tree, we define the CTE algorithm applied to a bucket-tree to be called Bucket-Tree Elimination (BTE). BTE has time and space complexity $O(r \cdot d^{tw+1})$.

2.2.2 Mini-Cluster-Tree Elimination

The main drawback of CTE and any variant of join-tree algorithms is that they are time and space exponential in the treewidth (tw) and separator (s) size, respectively [36, 90], which are often very large. In order to overcome this problem, partition-based algorithms were introduced. Instead of combining all the functions in a cluster, when computing a message, we first partition the functions in the cluster into a set of mini-clusters such that each mini-cluster is bounded by a fixed number of variables (i-bound), and then process them separately. The algorithm, called Mini-Cluster-Tree Elimination (MCTE(i)) approximates CTE and it computes upper bounds on values computed by CTE.

In the Mini-Cluster-Tree Elimination the message $M_{(u,v)}$ that node u sends to node v is a set of functions computed as follows. The functions in $cluster(u) - M_{(v,u)}$ are partitioned into $\mathbf{P} = \mathbf{P_1}, \cdots, \mathbf{P_k}$, where $|\mathbf{scope}(\mathbf{P_j})| \leq i$, for a given i. The message $M_{(u,v)}$ is defined as $M_{(u,v)} = \{\max_{elim(u,v)} \prod_{g \in \mathbf{P_j}} g | \mathbf{P_j} \in \mathbf{P}\}$. Algorithm MCTE($i$) applied to the bucket-tree is called Mini-Bucket-Tree Elimination (MBTE(i)) [36].

Since the scope size of each mini-cluster is bounded by i, the time and space complexity of MCTE(i)/MBTE(i) is exponential in i. However, because of the partitioning, the functions $z(X_j)$ cannot be computed exactly any more. Instead, the output functions of MCTE(i)/MBTE(i), called $mz(X_j)$, are upper bounds on the exact functions $z(X_j)$ [36].

Clearly, increasing i is likely to provide better upper bounds at a higher cost. Therefore,

Procedure BBBT($\mathcal{T}$,i,s,L)
Input: Bucket-tree $\mathcal{T}$, parameter i,set of instantiated variables $S = s$, lower bound L.
Output: MPE probability conditioned on s.
1. If $S = \mathbf{X}$, return the probability of the current complete assignment.
2. Run MBTE(i); Let $\{mz_j\}$ be the set of heuristic values computed by MBTE(i) for each $X_j \in \mathbf{X} - S$.
3. Prune domains of uninstantiated variables, by removing values $x \in D_{X_l}$ for which $mz_l(x) \leq L$.
4. Backtrack: If $D_{X_l} = \emptyset$ for some variable X_l, return 0.
5. Otherwise let X_j be the uninstantiated variable with the smallest domain: $X_j = argmin_{X_k \in \mathbf{X}-S}|D_{X_k}|$.
6. Repeat while $D_{X_j} \neq \emptyset$
i. Let x_k be the value of X_j with the largest heuristic estimate: $x_k = argmax_{x_j \in D_{X_j}} mz_j(x_j)$.
ii. Set $D_{X_j} = D_{X_j} - x_k$.
iii. Compute $mpe = BBBT(\mathcal{T}, i, s \cup \{X_j = x_k\}, L)$.
iv. Set $L = max(L, mpe)$.
v. Prune D_{X_j} by L.
7. Return L.

Figure 2.2: BBBT(i): Branch-and-Bound with MBTE(i) based heuristics.

MCTE(i)/MBTE(i) allows trading upper bound accuracy for time and space complexity.

2.3 Partition-based Branch-and-Bound Search

This section focuses on the two systematic algorithms we used. Both use partition-based mini-bucket heuristics.

2.3.1 BBBT: Branch-and-Bound with Dynamic Heuristics

Since MBTE(i) computes upper bounds for each singleton-variable assignment simultaneously, when incorporated within a depth-first Branch-and-Bound algorithm, MBTE(i) can facilitate domain pruning and dynamic variable ordering.

Such a Branch-and-Bound algorithm, called BBBT(i), for solving the MPE problem is given in Figure 2.2. Initially it is called with BBBT($\langle T, \chi, \psi \rangle, i, \emptyset, 0$). At all times it

maintains a lower bound L which corresponds to the probability of the best assignment found so far. At each step, it executes MBTE(i) which computes the singleton assignment costs mz_i for each uninstantiated variable X_i (step 2), and then uses these costs to prune the domains of uninstantiated variables by comparing L against the heuristic estimate of each value (step 3). If the cost of the value is not more than L, it can be pruned because it is an upper bound. If as a result a domain of a variable becomes empty, then the current partial assignment is guaranteed not to lead to a better assignment and the algorithm can backtrack (step 4). Otherwise, BBBT(i) expands the current assignment picking a variable X_j with the smallest domain (variable ordering in step 5) and recursively solves a set of subproblems, one for each value of X_j, in decreasing order of heuristic estimates of its values (value ordering in step 6). If during the solution of the subproblem a better new assignment is found, the lower bound L can be updated (step 6iv).

Thus, at each node in the search space, BBBT(i) first executes MBTE(i), then prunes domains of all un-instantiated variables, and then recursively solves a set of subproblems. BBBT(i) performs a look-ahead computation that is similar (but not identical) to i-consistency at each search node.

2.3.2 BBMB: Branch-and-Bound with Static Heuristics

The strength of the BBMB(i) algorithm described in Chapter 1 was established in several empirical studies [65]. We highlight next the main differences between BBBT(i) and BBMB(i):

- BBMB(i) uses as a pre-processing step the Mini-Bucket-Elimination, which compiles a set of functions that can be used to assemble efficiently heuristic estimates during search. The main overhead is therefore the pre-processing step which is exponential in the i-bound but does not depend on the number of search nodes. BBBT(i) on the other hand computes the heuristic estimates solely during search using MBTE(i).

Consequently its overhead is exponential in the i-bound multiplied by the number of nodes visited.

- Because of the pre-computation of heuristics, BBMB(i) is limited to static variable ordering, while BBBT(i) uses a dynamic variable ordering.

- Finally, since at each step, BBBT(i) computes heuristic estimates for all un-instantiated variables, it can prune their domains, which provides a form of look-ahead. BBMB(i) on the other hand generates a heuristic estimate only for the next variable in the static ordering and prunes only its domain.

2.4 Non-Systematic Algorithms

This section focuses on two different types of incomplete algorithms: stochastic local search and iterative belief propagation for solving the MPE task in belief networks.

2.4.1 Local Search

Local search is a general optimization technique which can be used alone or as method for improving solutions found by other approximation schemes. Unlike the Branch-and-Bound algorithms, these methods do not guarantee an optimal solution.

Discrete Lagrangian Multipliers

The method of **Discrete Lagrangian Multipliers** (DLM) [122] is based on an extension of constraint optimization using Lagrange multipliers for continuous variables. In the weighted MAX-SAT domain, the clauses are the constraints, and the sum of the unsatisfied clauses is the cost function. In addition to the weight w_C, a Lagrangian multiplier λ_C is associated with each clause. The cost function for DLM is of the form: $\sum_C w_C + \sum_C \lambda_C$, where C ranges over the unsatisfied clauses. Every time a local maxima is encountered, the

λs corresponding to the unsatisfied clauses are incremented by adding a constant. DLM can be applied over the equivalent weighted MAX-SAT encoding of the given belief network, as shown in [102].

Guided Local Search

Guided Local Search (GLS) [93, 102] is a heuristically developed method for solving combinatorial optimization problems. It has been shown to be extremely efficient at solving general weighted MAX-SAT problems. Like DLM, GLS associates an additional weight with each clause C (λ_C). The cost function in this case is essentially $\sum_C \lambda_C$, where C ranges over the unsatisfied clauses. Every time a local maxima is reached, the λs of the unsatisfied clauses with maximum utility are increased by adding constant, where the utility of a clause C is given by $w_C/(1 + \lambda_C)$. Unlike DLM, which increments all the weights of the unsatisfied clauses, GLS modifies only a few of them.

Stochastic Local Search

Stochastic Local Search (SLS) [64] is a local search algorithm that at each step performs either a hill climbing or a stochastic variable change. Periodically, the search is restarted in order to escape local maxima. It was shown to be superior to simulated annealing and some pure greedy search algorithms. SLS can be applied directly on the given belief network.

2.4.2 Iterative Join-Graph Propagation

The **Iterative Join Graph Propagation** (IJGP) [39] algorithm belongs to the class of generalized belief propagation methods, recently proposed to generalize Pearl's belief propagation algorithm [104] using analogy with algorithms in statistical physics. This class of algorithms, developed initially for belief updating, is an iterative approximation method that applies the message passing algorithm of join-tree clustering to join-graphs, iteratively. It uses a parameter i that bounds the complexity and makes the algorithm anytime.

2.5 Experiments

We tested the performance of our scheme for solving the MPE task on several types of belief networks - random uniform Bayesian networks, NxN grids, coding networks, CPCS networks and 9 real world networks obtained from the Bayesian Network Repository[1]. On each problem instance we ran BBBT(i) and BBMB(i) with various i-bounds, as heuristics generators, as well as the local search algorithms discussed earlier. We also ran the Iterative Join Graph Propagation algorithm (IJGP(i)) on some of these problems.

We used the *min-degree* heuristic for computing the ordering of variables. It places a variable with the smallest degree at the end of the ordering, connects all of its neighbors, removes the variable from the graph and repeats the whole procedure.

We treat all algorithms as approximation anytime schemes. Algorithms BBBT(i) and BBMB(i) have any-time behavior and, if allowed to run until completion, will solve the problem exactly. However, in practice, both algorithms may be terminated at a time bound and may return sub-optimal solutions. On the other hand, neither the local search techniques, nor the belief propagation algorithms guarantee an optimal solution, even if given enough time.

To measure performance we used the accuracy ratio *opt* = P_{alg} / P_{MPE} between the value of the solution found by the test algorithm (P_{alg}) and the value of the optimal solution (P_{MPE}), whenever P_{MPE} was available. We only report results for the range *opt* $\geq$ 0.95. We also recorded the average running time for all algorithms, as well as the average number of search tree nodes visited by the Branch-and-Bound algorithms. When the size and difficulty of the problem did not allow an exact computation, we compared the quality of the solutions produced by the respective algorithms in the given time bound. For each problem class we chose a number of evidence variables randomly and fixed their values.

[1] www.cs.huji.ac.il/labs/compbio/Repository

k	BBBT BBMB IJGP i=2 %[time]{nodes}	BBBT BBMB IJGP i=4 %[time]{nodes}	BBBT BBMB IJGP i=6 %[time]{nodes}	BBBT BBMB IJGP i=8 %[time]{nodes}	BBBT BBMB IJGP i=10 %[time]{nodes}	GLS %[time]	DLM %[time]	SLS %[time]
2	90[6.30]{3.9K} 71[2.19]{1.6M} 62[0.04]	100[1.19]{781} 92[0.17]{0.1M} 66[0.06]	100[0.65]{366} 92[0.02]{10K} 66[0.13]	**100[0.44]{212}** 86[0.01]{3K} 71[0.32]	**100[0.43]{161}** 91[0.01]{1.2K} 67[0.87]	100[1.05]	0[30.01]	0[30.01]
3	28[46.6]{19K} 5[43.1]{16M} 34[0.07]	65[27.5]{5.5K} 78[24.4]{8.2M} 37[0.18]	86[15.4]{1.1K} **90[3.20]{0.8M}** 36[0.94]	86[19.3]{453} 89[1.23]{0.3M} 43[5.38]	80[27.5]{213} 83[0.58]{52.5K} 44[32.5]	39[44.02]	0[60.01]	0[60.01]
4	24[95.5]{63K} 3[89.4]{47M} 17[0.14]	46[74.7]{13.4K} 42[85.5]{37M} 14[0.47]	65[54.1]{1.6K} 89[25.4]{8M} 14[4.33]	67[65.7]{443} 90[5.44]{1.5M} 17[43.3]	37[151.2]{74} **99[4.82]{0.3M}** 20[468.5]	5[114.9]	0[120.01]	0[120.01]

Table 2.1: Average accuracy and time. **Random Bayesian networks** ($n = 100$, $c = 90$, $p = 2$). $w^* = 17$, 10 evidence, 100 samples.

k	BBBT BBMB IJGP i=2 %[time]	BBBT BBMB IJGP i=4 %[time]	BBBT BBMB IJGP i=6 %[time]	BBBT BBMB IJGP i=8 %[time]	BBBT BBMB IJGP i=10 %[time]	GLS %[time]	DLM %[time]	SLS %[time]
2	84[7.34] 61[3.49] 62[0.04]	98[2.48] 91[0.30] 66[0.06]	100[0.88] 89[0.05] 66[0.13]	100[0.66] 88[0.02] 71[0.31]	**100[0.59]** 88[0.02] 67[0.86]	100[1.25]	0[30.02]	0[30.02]
3	36[42.2] 8[47.5] 34[0.04]	78[19.1] **95[1.81]** 37[0.10]	**95[9.64]** **95[1.81]** 36[0.49]	94[10.7] 86[0.71] 43[2.86]	93[16.8] 84[0.33] 44[17.0]	49[38.7]	0[60.02]	0[60.01]
4	24[97.7] 2[114.4] 17[0.06]	40[80.3] 39[92.3] 14[0.23]	61[62.4] 84[33.2] 14[2.12]	58[82.0] 90[7.39] 17[21.9]	30[269] **99[7.95]** 20[226.8]	5[115.03]	0[120.01]	0[120.01]

Table 2.2: Average accuracy and time. **Random Noisy-OR networks** ($n = 100$, $c = 90$, $p = 2$). $P_{noise} = 0.2, P_{leak} = 0.01$. w^*=17, 10 evidence, 100 samples.

2.5.1 Random Bayesian Networks and Noisy-OR Networks

The random Bayesian networks were generated using parameters (n, k, c, p), where n is the number of variables, k is their domain size, c is the number of conditional probability tables (CPTs) and p is the number of parents in each CPT. The structure of the network is created by randomly picking c variables out of n and, for each, randomly selecting p parents from their preceding variables, relative to some ordering. For random uniform Bayesian networks, each probability table is generated uniformly randomly. For Noisy-OR networks, each probability table represents an OR-function with a given noise and leak probabilities: $P(X = 0|Y_1, \ldots, Y_p) = P_{leak} \times \prod_{Y_i=1} P_{noise}$

Table 2.1 presents experiments with random uniform Bayesian networks. In each table, parameters n, c and p are fixed, while k, controlling the domain size of the network's

variables, is changing. For each value of k, we generate 100 instances. We gave each algorithm a time limit of 30, 60 and 120 seconds, depending on the value of the domain size. Each test case had 10 randomly selected evidence variables. We have highlighted the best performance point in each row.

For example, Table 2.1 reports the results with random problems having $n = 100$, $c = 90$, $p = 2$. Each horizontal block corresponds to a different value of k. The columns show results for BBBT(i), BBMB(i) and IJGP(i) at various levels of i, as well as for GLS, DLM and SLS. Looking at the first line in Table 2.1 we see that in the accuracy range *opt* $\geq$ 0.95 and for the smallest domain size ($k = 2$) BBBT(i) with i=2 solved 90% of the instances using 6.30 seconds on average and exploring 3.9K nodes, while BBMB(i) with i=2 only solved 71% of the instances using 2.19 seconds on average and exploring a much larger search space (1.6M nodes). GLS significantly outperformed the other local search methods, as also observed in [102], and solved all instances using 1.05 seconds on average. However, as BBBT(i)'s bound increases, it is better than GLS. As the domain size increases, the problem instances become harder. The overall performance of local search algorithms, especially GLS's performance, deteriorates quite rapidly.

When comparing BBBT(i) to BBMB(i) we notice that at larger domain sizes ($k \in \{3, 4\}$) the superiority of BBBT(i) is more pronounced for small i-bounds ($i \in \{2, 4\}$), both in terms of the quality of the solution and search space explored. This may be significant, because small i-bounds require restricted space.

Figures 2.3 and 2.4 provide an alternative view of the performance of BBBT(i) and BBMB(i) against GLS as anytime algorithms. Let $F_{alg}(t)$ be the fraction of problems solved completely by the algorithm alg by time t. Each graph in Figure 2.3 plots $F_{BBBT(i)}(t)$, $F_{BBMB(i)}(t)$ for some selected values of i, as well as $F_{GLS}(t)$. Different values of the domain size are discussed, namely $k = 2$ and $k = 3$, respectively. Figure 2.3 shows the distributions of $F_{BBBT(i)}(t)$, $F_{BBMB(i)}(t)$ and $F_{GLS}(t)$ for the random Bayesian networks when $N = 100$, $C = 90$, $P = 2$ (corresponding to the first two rows in Table 2.1), while

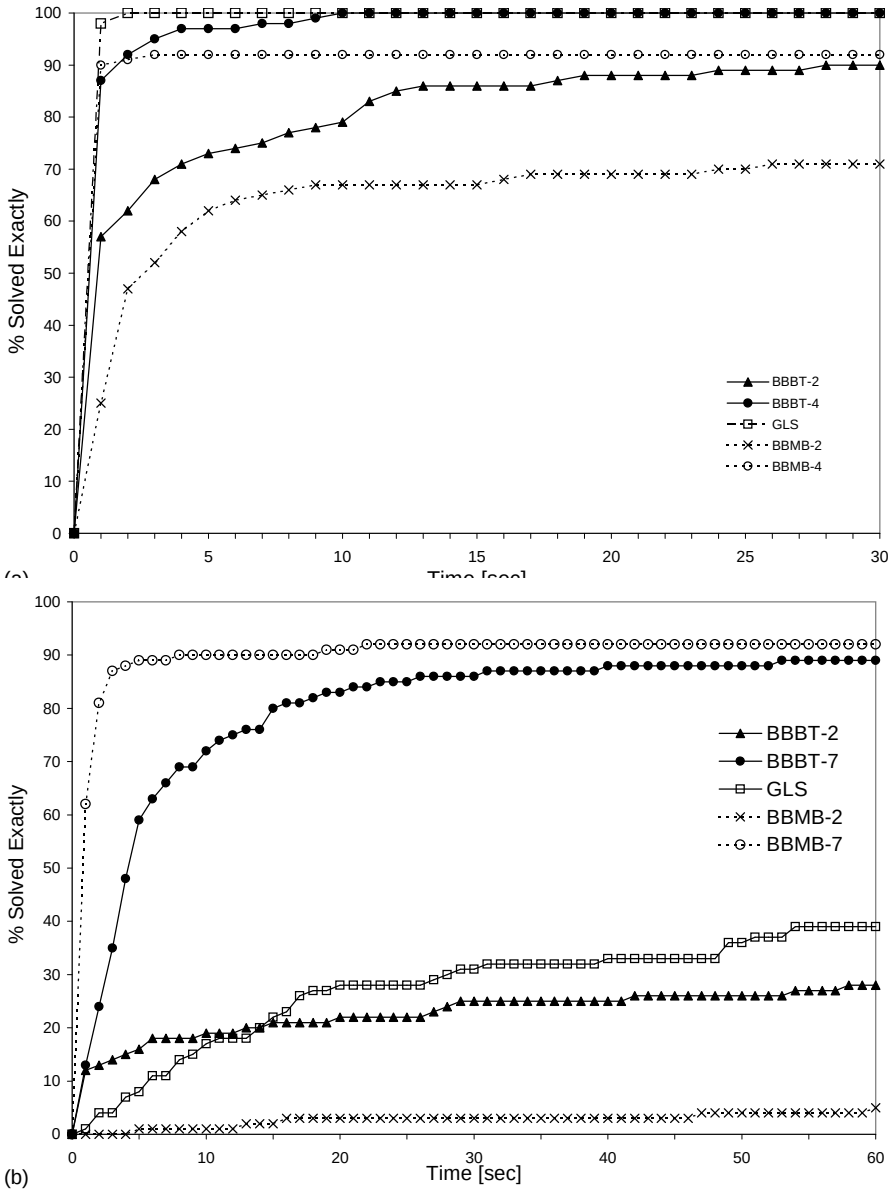


Figure 2.3: Accuracy versus time. **Random Bayesian networks** ($n = 100$, $c = 90$, $p = 2$). (a) $k = 2$, (b) $k = 3$. 10 evidence, 100 samples.

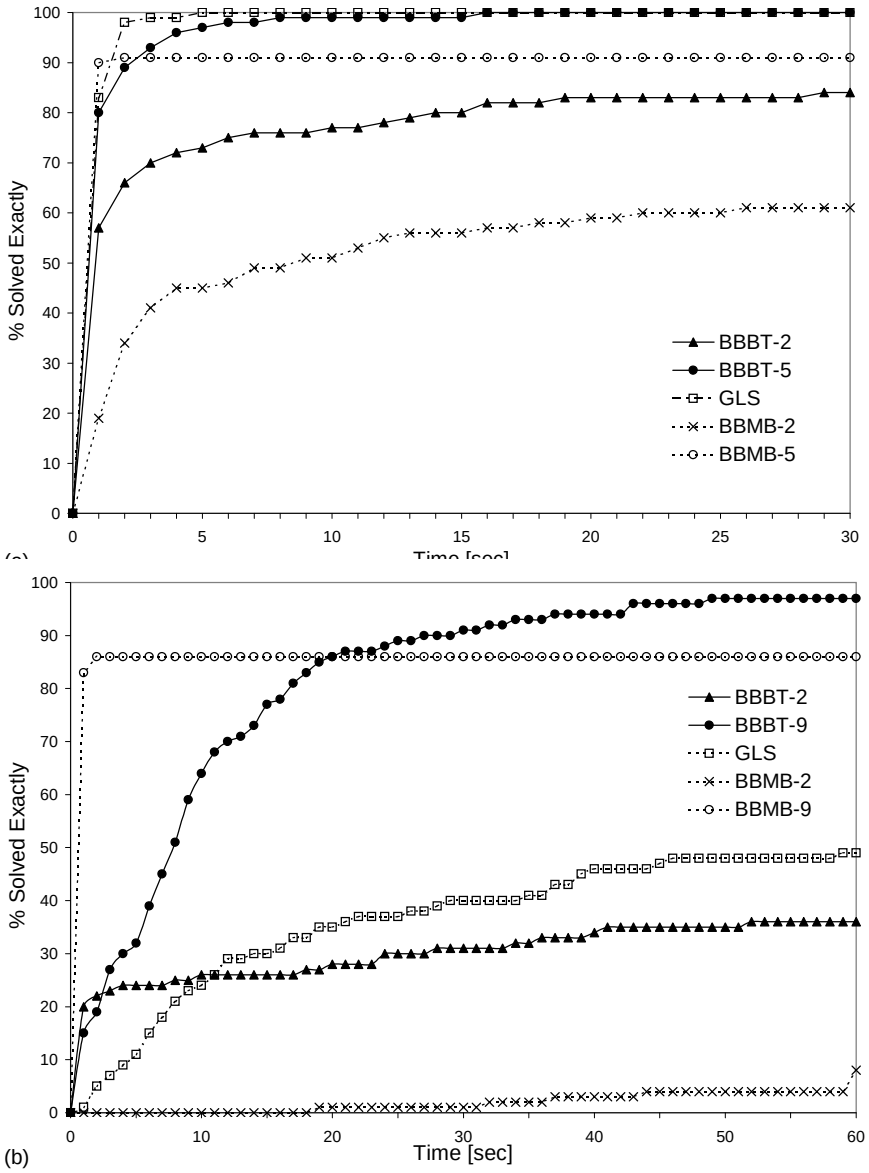


Figure 2.4: Accuracy versus time. **Random Noisy-OR networks** ($n = 100$, $c = 90$, $p = 2$). (a) $k = 2$, (b) $k = 3$. 10 evidence, 100 samples.

k	BBBT / GLS BBMB / GLS i=2 # best	BBBT / GLS BBMB / GLS i=3 # best	BBBT / GLS BBMB / GLS i=4 # best	BBBT / GLS BBMB / GLS i=5 # best	BBBT / GLS BBMB / GLS i=6 # best
2	0/29 0/24	0/25 0/19	0/23 0/19	0/21 0/5	0/20 0/5
3	4/26 1/29	5/25 2/28	5/25 2/28	9/21 2/28	10/20 4/26
5	28/2 5/25	28/2 5/25	30/0 7/23	30/0 12/18	30/0 23/7
7	25/5 18/12	22/8 15/15	24/6 17/13	19/11 20/10	21/9 25/5

Table 2.3: # wins given fixed time bound. **Random Bayesian networks** ($n = 100$, $c = 90$, $p = 3$). $w^* = 30$, 10 evidence, 30 samples.

Figure 2.4 corresponds to the random Noisy-OR networks from Table 2.2.

Clearly, if $F_{alg^i}(t) > F_{alg^j}(t)$, then $F_{alg^i}(t)$ completely dominates $F_{alg^j}(t)$. For example, in Figure 2.3(a), GLS is highly competitive with BBBT(4) and both significantly outperform BBBT(i) and BBMB(i) for smaller i-bounds. In contrast, Figure 2.3(b) shows how the best local search method deteriorates as the domain size increases. The same pattern appears for the case of Noisy-OR networks shown in Figure 2.4. GLS dominates slightly BBBT(5) for $k = 2$ and is clearly outperformed by BBBT(9) for $k = 3$.

We also experimented with a much harder set of random Bayesian networks. The dataset consisted of random networks with parameters $n = 100$, $c = 90$, $p = 3$. In this case, the induced width of the problem instances was around 30, thus it was not possible to compute exact solutions. We studied four domain sizes $k \in \{2, 3, 5, 7\}$. For each value of k, we generated 30 problem instances. Each algorithm was allowed a time limit of 30, 60, 120 and 180 seconds, depending on the domain size. We found that the costs of the solutions generated by DLM and SLS were several orders of magnitude smaller than those found by GLS, BBBT(i) and BBMB(i). Hence, we only report the latter three algorithms.

Table 2.3 compares the frequency that the solution was the best for each of the three algorithms (ties are removed). We notice again that GLS excelled at finding the best solution for smaller domain sizes, in particular for $k = 2$ and $k = 3$. On the other hand, for larger

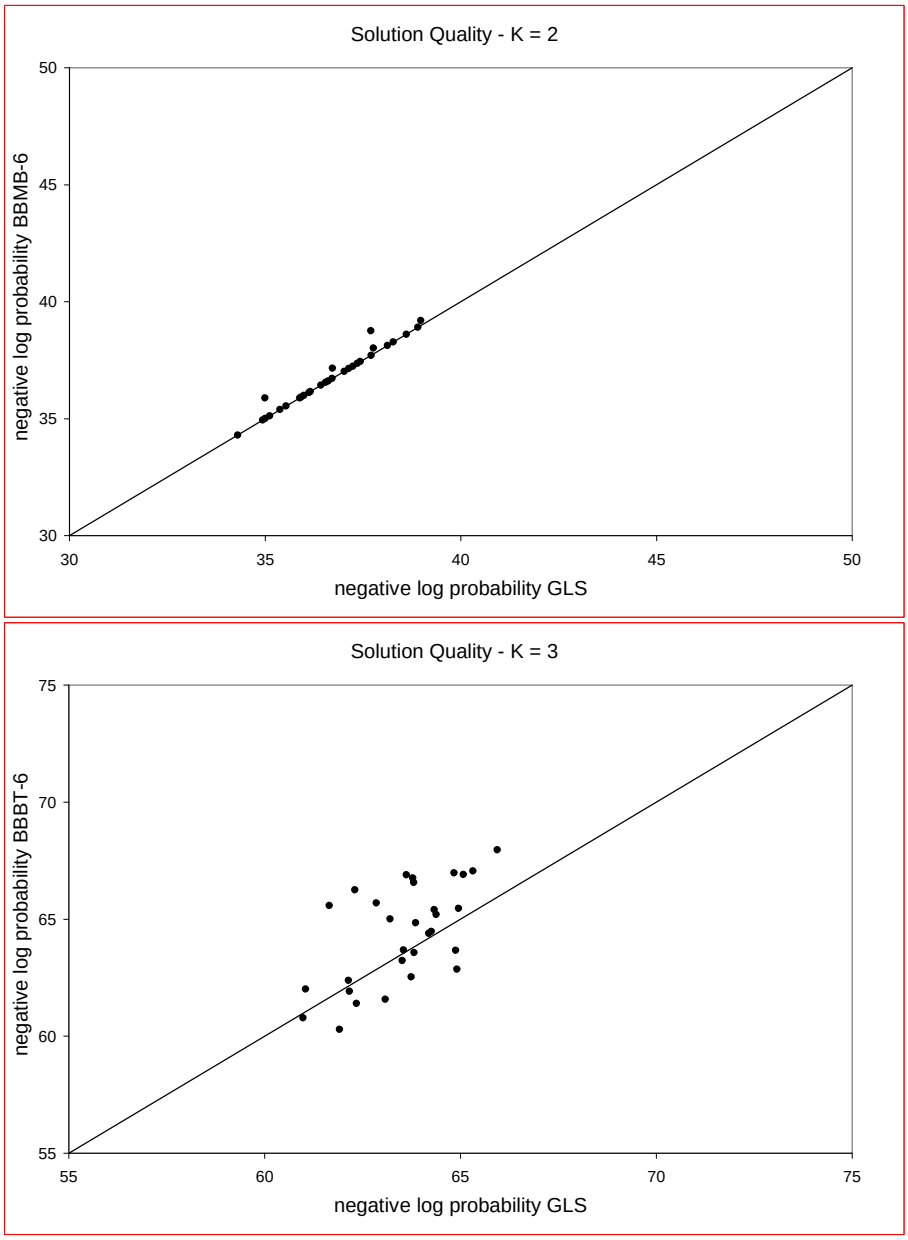


Figure 2.5: Solution quality at fixed time bound. **Random Bayesian networks** ($n = 100$, $c = 90$, k, $p = 3$). $w^* = 30$, 10 evidence, 100 samples. $k \in \{2, 3\}$

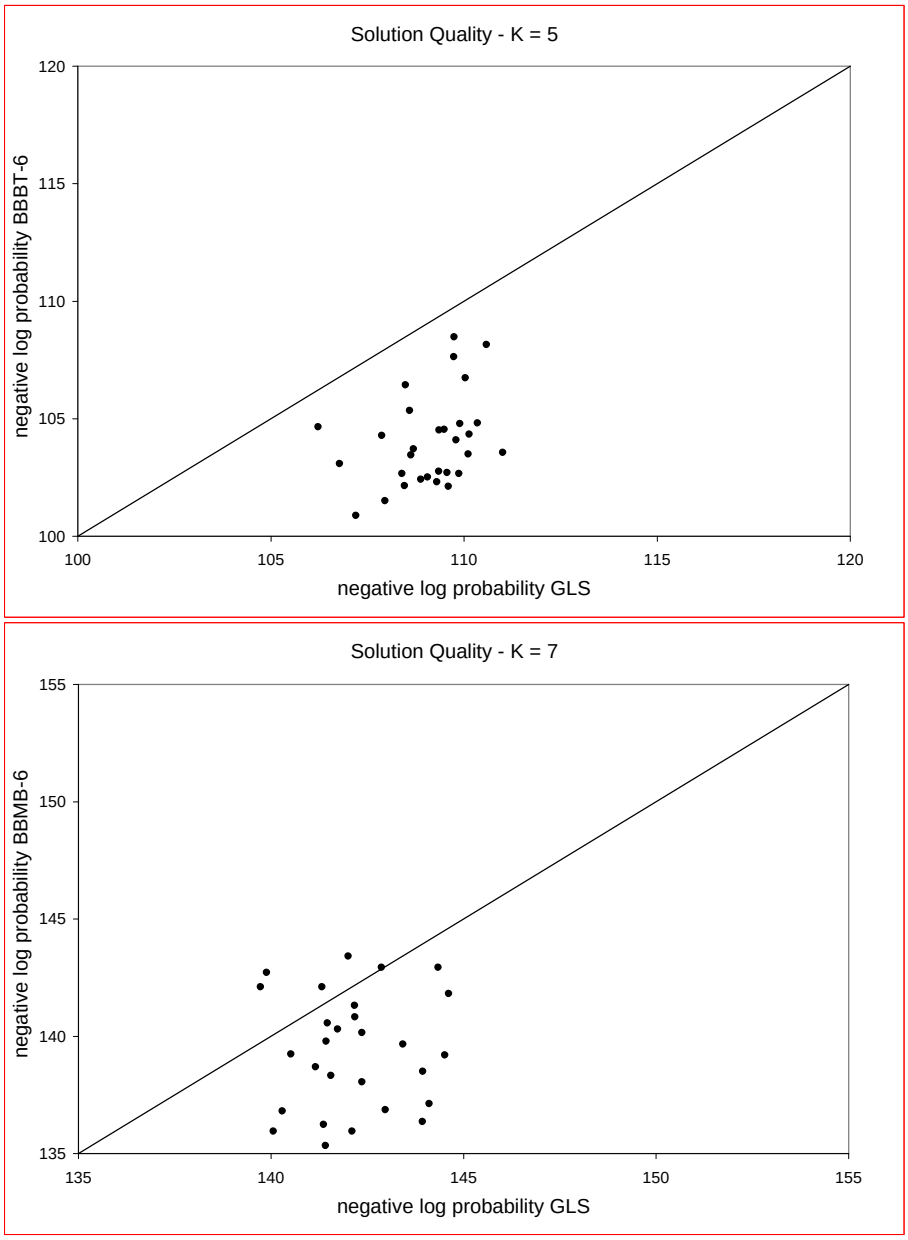


Figure 2.6: Solution quality at fixed time bound. **Random Bayesian networks** ($n = 100$, $c = 90$, k, $p = 3$). $w^* = 30$, 10 evidence, 100 samples. $k \in \{5, 7\}$

k	BBBT / BBMB i=2 # wins # nodes	BBBT / BBMB i=3 # wins # nodes	BBBT / BBMB i=4 # wins # nodes	BBBT / BBMB i=5 # wins # nodes
2	20/10 15.3K/13.8M	12/18 14.5K/16.2M	18/12 12.3K/15.9M	18/12 9.4K/11.8M
3	27/3 19.9K/16.3M	26/4 13.8K/16.8M	29/1 12.3K/15.9M	28/2 4.8K/14.2
5	29/1 18.3K/10.5M	30/0 9.1K/13.8M	30/0 3.5K/13.2M	27/3 0.9K/12.6M
7	24/6 7.7K/8.3M	26/4 3.4K/10.6M	14/16 114/10.9M	10/20 8/9.6M

Table 2.4: BBBT(i) vs. BBMB(i). **Random Bayesian networks** ($n = 100$, $c = 90$, $p = 3$). 10 evidence, 30 samples, 30 seconds.

domain sizes ($k \in \{5, 7\}$), the power of BBBT(i) is more pronounced at smaller i-bounds, whereas BBMB(i) is more efficient at larger i-bounds. Figures 2.5 and 2.6 show, pictorially, the quality of the solutions produced by GLS against the ones produced by BBBT(i) and BBMB(i). For each plot, corresponding to a different value of the domain size k, the X axis represents the negative log probability of the solutions found by GLS and the Y axis represents the negative log probability of the solutions found by BBBT(i) and BBMB(i). The superiority of Branch-and-Bound based methods for larger domain sizes is significant since these are algorithms that can prove optimality when given enough time, unlike local search methods.

Table 2.4 shows comparatively the performance of BBBT(i) as compared to BBMB(i). Each entry in the table shows the number of times BBBT(i) produced a better solution than BBMB(i) (# wins) as well as the average number of search tree nodes visited by both algorithms. We notice again the superiority of BBBT(i) at relatively small i-bounds.

2.5.2 Random Grid Networks

In grid networks, n is a square number and each CPT is generated uniformly randomly or as a Noisy-OR function. We experimented with a synthetic set of 10-by-10 grid networks. We report results on three different domain sizes. For each value of k, we generate 100

k	BBBT / BBMB i=2 %[time]	BBBT / BBMB i=4 %[time]	BBBT / BBMB i=6 %[time]	BBBT / BBMB i=8 %[time]	BBBT / BBMB i=10 %[time]	GLS %[time]	DLM %[time]	SLS %[time]
2	51[17.7]	99[2.62]	100[0.66]	100[0.48]	**100[0.42]**			
	1[29.9]	13[23.7]	93[2.16]	92[0.08]	95[0.02]	100[1.54]	0[30.01]	0[30.01]
3	3[58.7]	28[47.4]	80[19.5]	93[14.8]	**94[23.2]**			
	0[60.01]	1[58.9]	25[50.9]	89[8.63]	92[0.73]	4[58.7]	0[60.01]	0[60.01]
4	1[118.8]	12[108.3]	46[78.4]	61[88.5]	33[136]			
	0[120]	0[120]	6[113.4]	72[46.4]	**85[9.91]**	0[120]	0[120]	0[120]

Table 2.5: Average accuracy and time. **Random grid networks** ($n = 100$). $w^* = 15$, 10 evidence, 100 samples.

σ	BBBT BBMB IJGP i=2 BER[time]	BBBT BBMB IJGP i=4 BER[time]	BBBT BBMB IJGP i=6 BER[time]	BBBT BBMB IJGP i=8 BER[time]	BBBT BBMB IJGP i=10 BER[time]	IBP GLS SLS BER[time]
0.32	0.0056[3.18]	0.0104[2.87]	0.0072[1.75]	0.0034[0.72]	0.0034[0.59]	**0.0034[0.01]**
	0.0034[0.07]	0.0034[0.08]	0.0034[0.03]	0.0034[0.01]	0.0034[0.02]	0.2344[60.01]
	0.0034[0.16]	0.0034[0.18]	0.0034[0.33]	0.0034[0.92]	0.0034[3.02]	0.4980[60.01]
0.40	0.0642[19.4]	0.0400[12.8]	0.0262[6.96]	0.0148[4.52]	0.0190[4.34]	**0.0108[0.01]**
	0.0114[0.63]	0.0114[0.53]	0.0114[0.12]	0.0114[0.05]	0.0114[0.04]	0.2084[60.01]
	0.0114[0.16]	0.0138[0.18]	0.0118[0.33]	0.0116[0.91]	0.0120[3.02]	0.5128[60.01]
0.52	0.1920[48.1]	0.1790[42.0]	0.1384[31.3]	0.1144[21.4]	0.1144[19.7]	**0.0894[0.01]**
	0.0948[1.35]	0.0948[1.47]	0.0948[0.36]	0.0948[0.11]	0.0948[0.05]	0.2462[60.02]
	0.1224[0.08]	0.1242[0.09]	0.1256[0.16]	0.1236[0.47]	0.1132[1.54]	0.5128[60.01]

Table 2.6: Average BER and time. **Random coding networks** ($n = 200$, $p = 4$). $w^* = 22$, 60 seconds, 100 samples.

problem instances. Each algorithm was allowed a time limit of 30, 60 and 120 seconds, depending on the domain size.

Table 2.5 shows the average accuracy and running time for each algorithm. Once again, in terms of accuracy, GLS is highly competitive with the systematic search algorithms at the lowest domain size ($k = 2$). Even though the difficulty of the problem instances was relatively small ($i.e.$, the induced width was around 15), the degradation in GLS's performance is more pronounced as the domain size increases. Comparing BBBT(i) to BBMB(i), we notice again that BBBT(i) was superior to BBMB(i) for smaller i-bounds.

Network (n, w*) (avg, max)	BBBT/ BBMB/ IJGP i=2 %[time]	BBBT/ BBMB/ IJGP i=3 %[time]	BBBT/ BBMB/ IJGP i=4 %[time]	BBBT/ BBMB/ IJGP i=5 %[time]	BBBT/ BBMB/ IJGP i=6 %[time]	BBBT/ BBMB/ IJGP i=7 %[time]	BBBT/ BBMB/ IJGP i=8 %[time]	BBBT/ BBMB/ IJGP i=10 %[time]	GLS % [time]	DLM % [time]	SLS % [time]
Barley	-	-	90[6.33]	100[4.28]	100[3.29]	100[2.81]	**100[2.91]**	-	0	0	0
(48, 8)	-	-	25[12.8]	40[2.32]	65[0.43]	90[0.85]	**100[2.41]**	-	[30.01]	[30.01]	[30.01]
(8, 67)	67[0.99]	67[1.11]	63[1.49]	70[5.32]	80[17.9]	-	-	-			
Diabetes*	0[120]	0[123]	0[127]	**90[21.1]**	-	-	-	-	0	0	0
(413, 5)	0[120]	0[120]	5[114]	**100[2.01]**	-	-	-	-	[120.01]	[120.01]	[120.01]
(11, 21)	3[8.60]	3[11.2]	43[86.0]	97[311.1]	100[384.6]	-	-	-			
Mildew	100[0.28]	**100[0.17]**	100[0.56]	-	-	-	-	-	15	0	90
(35, 4)	30[10.5]	65[7.5]	95[0.18]	-	-	-	-	-	[30.02]	[30.02]	[30.02]
(17, 100)	90[3.59]	87[3.68]	97[33.3]	100[53.2]	-	-	-	-			
Munin1	90[6.13]	-	**100[6.48]**	-	40[23.8]	-	75[13.4]	80[43.1]	10	0	0
(189, 11)	0[30]	-	5[27.2]	-	20[24.1]	-	70[6.77]	100[9.03]	[30.02]	[30.02]	[30.02]
(5, 21)	90[0.45]	90[0.49]	97[1.10]	93[4.28]	93[14.5]	97[70.2]	100[191.9]	-			
Munin2	95[1.65]	95[1.73]	95[1.65]	95[1.99]	95[2.32]	95[2.48]	**100[1.97]**	-	0	0	0
(1003, 7)	95[30.3]	95[31.7]	95[30.5]	95[31.8]	95[31.3]	100[30.5]	**100[1.84]**	-	[30.01]	[30.01]	[30.01]
(5, 21)	95[2.44]	95[2.94]	95[5.17]	100[20.3]	95[64.9]	-	-	-			
Munin3	0[30.8]	0[30.9]	0[31.3]	5[31.7]	0[40.9]	90[4.72]	**100[2.2]**	-	0	0	0
(1044, 7)	0[30.2]	0[31]	0[32.3]	5[29.9]	0[32.7]	95[2.14]	**100[1.01]**	-	[30.02]	[30.02]	[30.02]
(5, 21)	80[1.47]	95[1.72]	85[3.10]	85[10.8]	90[38.9]	-	-	-			
Munin4	0[31]	0[31]	0[31.9]	0[37.7]	0[44.5]	0[58.8]	0[170.4]	-	0	0	0
(1041, 8)	0[30.2]	0[31.4]	0[31.6]	0[32]	0[30.3]	30[22.1]	**85[3.4]**	-	[30.02]	[30.02]	[30.02]
(5, 21)	85[1.52]	75[1.66]	90[4.15]	95[15.6]	95[43.6]	-	-	-			
Pigs	90[15.2]	-	100[3.73]	-	100[2.36]	-	100[0.58]	**100[0.56]**	10	0	0
(441, 12)	0[30.01]	-	60[4.85]	-	80[0.02]	-	95[0.04]	95[0.12]	[30.02]	[30.02]	[30.02]
(3, 3)	80[0.31]	73[0.37]	77[0.53]	83[0.86]	80[1.43]	80[2.49]	83[6.27]	93[27.3]			
Water	**100[0.01]**	-	100[0.02]	-	100[0.03]	-	100[0.04]	100[0.09]	100	75	100
(32, 11)	55[4.51]	-	60[4.5]	-	75[0.01]	-	100[0.02]	100[0.06]	[30.02]	[30.02]	[30.02]
(3, 4)	97[0.09]	97[0.09]	97[0.10]	97[0.14]	100[0.26]	100[0.45]	100[1.12]	100[5.94]			
CPCS54	100[0.35]	-	100[0.18]	-	100[0.11]	-	100[0.09]	**100[0.06]**	100	0	100
(54, 15)	35[0.02]	-	60[0.01]	-	50[0.01]	-	55[0.004]	60[0.003]	[30.02]	[30.02]	[30.02]
(2, 2)	67[0.06]	77[0.06]	67[0.06]	70[0.07]	63[0.09]	70[0.11]	63[0.16]	73[0.38]			
CPCS179	100[1.69]	-	100[1.01]	-	**100[0.05]**	-	100[0.11]	-	100	30	30
(179, 8)	80[0.02]	-	80[0.02]	-	**100[0.02]**	-	100[0.07]	-	[30.02]	[30.02]	[30.02]
(2, 4)	100[2.50]	100[2.52]	100[2.99]	100[3.37]	100[6.49]	100[8.63]	100[36.9]	-			
CPCS360b	100[0.17]	-	100[0.27]	-	100[0.21]	-	**100[0.19]**	100[0.32]	100	100	100
(360, 20)	100[0.04]	-	**100[0.03]**	-	**100[0.03]**	-	**100[0.03]**	100[0.04]	[30.02]	[30.02]	[30.02]
(2, 2)	100[10.6]	100[10.4]	100[10.5]	100[10.1]	100[9.82]	100[8.19]	100[8.59]	100[12.5]			
CPCS422b	65[52.6]	-	70[48.7]	-	70[47.2]	-	90[21.5]	95[12.9]	100	65	65
(422, 23)	100[0.5]	-	100[0.49]	-	100[0.49]	-	**100[0.47]**	**100[0.47]**	[120.01]	[120.01]	[120.01]
(2, 2)	83[88.0]	83[86.8]	87[86.4]	90[84.3]	83[85.3]	87[77.7]	87[77.1]	90[70.9]			

Table 2.7: Results for experiments on 13 **real world networks**. Average accuracy and time.

2.5.3 Random Coding Networks

Our coding networks fall within the class of *linear block codes* [65]. They can be represented as four-layer belief networks having K nodes in each layer. The decoding algorithm takes the coding network as input and the observed channel output and computes the MPE assignment. The performance of the decoding algorithm is usually measured by the Bit Error Rate (BER), which is simply the observed fraction of information bit errors.

We tested random coding networks with K=50 input bits and various levels of channel noise σ. For each value of σ we generate 100 problem instances. Each algorithm was allowed a time limit of 60 seconds. Table 2.6 reports the average Bit Error Rate, as well as the average running time of the algorithms. We see that BBBT(i) and BBMB(i) outperformed considerably GLS. On the other hand, only BBMB(i) is competitive to IBP, which is the

best performing algorithm for coding networks.

2.5.4 Real-World Networks

Our realistic domain contained 9 Bayesian networks from the Bayesian Network Repository, as well as 4 CPCS networks derived from the Computer-Based Care Simulation system. For each network, we ran 20 test cases. Each test case had 10 randomly selected evidence variables, ensuring that the probability of evidence was positive. Each algorithm was allowed a 30 second time limit.

Table 2.7 summarizes the results. For each network, we list the number of variables, the average and maximum domain size for its variables, as well as the induced width. We also provide the percentage of exactly solved problem instances and the average running time for each algorithm.

In terms of accuracy, we notice a significant dominance of the systematic algorithms over the local search methods, especially for networks with large domains ($e.g.$, Barley, Mildew, Diabetes, Munin). For networks with relatively small domain sizes ($e.g.$, Pigs, Water, CPCS networks) the non-systematic algorithms, in particular GLS, solved almost as many problem instances as the Branch-and-Bound algorithms. Nevertheless, the running time of BBBT(i) and BBMB(i) was much better in this case, because GLS had to run until exceeding the time limit, even though it might have found the optimal solution within the first few iterations. BBBT(i) and BBMB(i) on the other hand terminated, hence proving optimality.

We also used for comparison the IJGP(i) algorithm, set up for 30 iterations. In terms of average accuracy, we notice the stable performance of the algorithm in almost all test cases. For networks with large domain sizes, IJGP(i) significantly dominated the local search algorithms and in some cases it even outperformed the BBBT(i) and BBMB(i) algorithms ($e.g.$, Barley, Mildew, Munin).

2.6 Conclusion to Chapter 2

We investigated the performance of two Branch-and-Bound search algorithms, BBBT(i) and BBMB(i), against a number of state-of-the-art stochastic local search and generalized belief propagation algorithms for the problem of solving the MPE task in Bayesian networks. Both BBBT(i) and BBMB(i) use the idea of partion-based approximation of inference for heuristic computation, but in different ways: while BBMB(i) uses a static pre-computed heuristic function, BBBT(i) computes it dynamically at each step. We observed over a wide range of problem classes, both random and real-world benchmarks, that BBBT(i) and BBMB(i) do not dominate each other. While BBBT(i) can sometimes improve significantly over BBMB(i), in many other instances its (quite significant) pruning power does not outweigh its time overhead. Both algorithms are powerful in different cases. In general, when large i-bounds are effective, BBMB(i) is more powerful, however, when space is restricted, BBBT(i) with small i-bounds is often more powerful. More significantly, we also showed that the SLS algorithms are overall inferior to BBBT(i) and BBMB(i), except when the domain size is small, in which case they are competitive. This is in stark contrast with the performance of systematic versus non-systematic on CSP/SAT problems, where SLS algorithms often significantly outperform complete methods. An additional advantage of BBBT(i) and BBMB(i) is that as complete algorithms they can prove optimality if given enough time, unlike SLS.

When designing algorithms to solve an NP-hard task, one cannot hope to develop a single algorithm that would be superior across all problem classes. Our experiments show that BBBT(i) and BBMB(i), when viewed as a collection of algorithms parameterized by i, show robust performance over a wide range of MPE problem classes, because for each problem instance there is a value of i, such that the performance of BBBT(i) as well as BBMB(i) dominates that of SLS.

Chapter 3

AND/OR Branch-and-Bound Search for Graphical Models

3.1 Introduction

Graphical models such as belief networks, constraint networks, Markov networks or influence diagrams are a widely used representation framework for reasoning with probabilistic and deterministic information. These models use graphs to capture conditional independencies between variables, allowing a concise representation of the knowledge as well as efficient graph-based query processing algorithms. Optimization problems such as finding the most likely state of a belief network or finding a solution that violates the least number of constraints in a constraint network can be defined within this framework and they are typically tackled with either *inference* or *search* algorithms.

Inference-based algorithms (*e.g.*, Variable Elimination, Tree Clustering) were always known to be good at exploiting the independencies captured by the underlying graphical model. They provide worst case time guarantees exponential in the treewidth of the underlying graph. Unfortunately, any method that is time-exponential in the treewidth is also space exponential in the treewidth or separator width, therefore not practical for models with large treewidth.

Search-based algorithms (*e.g.*, Branch-and-Bound search) traverse the model's search space where each path represents a partial or full solution. The linear structure of such

traditional search spaces does not retain the independencies represented in the underlying graphical models and, therefore, search-based algorithms may not be nearly as effective as inference-based algorithms in using this information and therefore do not accommodate informative performance guarantees. This situation has changed in the past few years with the introduction of AND/OR search spaces for graphical models [38]. In addition, search methods can accommodate *implicit* specifications of the functional relationships ($i.e.$, procedural or functional form) while inference schemes often rely on an *explicit* tabular representation over the (discrete) variables. For these reasons, search-based algorithms are the only choice available for models with large treewidth and with implicit representation.

The AND/OR search space for graphical models [38] is a new framework that is sensitive to the independencies in the model, often resulting in substantially reduced complexities. It is guided by a *pseudo tree* [48, 106] that captures independencies in the graphical model, resulting in a search space exponential in the depth of the pseudo tree, rather than in the number of variables.

Contribution

In this chapter we present a new generation of AND/OR Branch-and-Bound algorithms (AOBB) that explore the AND/OR search tree in a depth-first manner for solving optimization problems in graphical models. As in traditional Branch-and-Bound search, the efficiency of these algorithms depends heavily also on their guiding heuristic function. A class of partitioning-based heuristic functions, based on the Mini-Bucket approximation [42] and known as *static mini-bucket heuristics* was shown to be powerful for optimization problems in the context of OR search spaces [65]. The Mini-Bucket algorithm provides a scheme for extracting heuristic information automatically from the functional specification of the graphical model, which is applicable to any graphical model. The accuracy of the Mini-Bucket algorithm is controlled by a bounding parameter, called i-bound, which allows varying degrees of heuristics accuracy and results in a spectrum of search algorithms

that can trade off heuristic strength and search [65]. We show here how the pre-computed mini-bucket heuristic as well as any other heuristic information can be incorporated in AND/OR search, then we introduce *dynamic mini-bucket heuristics*, which are computed dynamically at each node of the search tree.

Since variable orderings can influence dramatically the search performance, we also introduce a collection of *dynamic* AND/OR Branch-and-Bound algorithms that extend AOBB by combining the AND/OR decomposition principle with dynamic variable ordering heuristics. There are three approaches to incorporating dynamic orderings into AOBB. The first one improves AOBB by applying an independent semantic variable ordering heuristic whenever the partial order dictated by the static decomposition principle allows. The second, is an orthogonal approach that gives priority to the semantic variable ordering heuristic and applies problem decomposition as a secondary principle. Since the structure of the problem may change dramatically during search we introduce a third approach that uses a dynamic decomposition method coupled with semantic variable ordering heuristics.

We apply the depth-first AND/OR Branch-and-Bound approach to two common optimization problems in graphical models: finding the Most Probable Explanation (MPE) in Bayesian networks [104] and solving Weighted CSPs [9]. We experiment with both random models and real-world benchmarks. Our results show conclusively that the new depth-first AND/OR Branch-and-Bound algorithms improve dramatically over traditional ones exploring the OR search space, especially when the heuristic estimates are inaccurate and the algorithms rely primarily on search and cannot prune the search space efficiently.

The research presented in this chapter is based in part on [79, 81].

Chapter Outline

The chapter is organized as follows. Section 3.2 describes the AND/OR representation of the search space. In Section 3.3 we introduce the new depth-first AND/OR Branch-and-Bound algorithm (AOBB) for searching AND/OR trees. Section 3.4 presents several general

purpose heuristic functions that can guide the search focusing on the mini-bucket heuristics. In Section 3.5 we describe AOBB's extension with dynamic variable ordering heuristics. Section 3.6 is dedicated to an extensive empirical evaluation, Section 3.7 overviews related work, and Section 3.8 provides concluding remarks.

3.2 AND/OR Search Trees For Graphical Models

In this section we overview the AND/OR search space for graphical models, which forms the core of our work in this chapter.

The usual way to do search in graphical models is to instantiate variables in turn, following a static/dynamic variable ordering. In the simplest case, this process defines a search tree (called here OR search tree), whose state nodes represent partial variable assignments. Since this search space does not capture the structure of the underlying graphical model an AND/OR search space recently introduced for general graphical models [38] can be used instead. The AND/OR search space is defined using a backbone *pseudo tree* [48, 106].

DEFINITION **21 (pseudo tree, extended graph)** *Given an undirected graph $G = (\mathbf{V}, \mathbf{E})$, a directed rooted tree $\mathcal{T} = (\mathbf{V}, \mathbf{E}')$ defined on all its nodes is called* pseudo tree *if any arc of G which is not included in E' is a back-arc, namely it connects a node to an ancestor in $\mathcal{T}$. Given a pseudo tree $\mathcal{T}$ of G, the* extended graph *of G relative to $\mathcal{T}$ is defined as $G^{\mathcal{T}} = (\mathbf{V}, \mathbf{E} \cup \mathbf{E}')$ (see Example 6 ahead).*

We next define the notion of AND/OR search tree for a graphical model.

DEFINITION **22 (AND/OR search tree [38])** *Given a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, its primal graph G and a backbone pseudo tree $\mathcal{T}$ of G, the associated AND/OR search tree, denoted $S_{\mathcal{T}}(\mathcal{R})$, has alternating levels of AND and OR nodes. The OR nodes are labeled X_i and correspond to the variables. The AND nodes are labeled $\langle X_i, x_i \rangle$ (or simply x_i) and correspond to value assignments in the domains of the variables. The structure of*

the AND/OR search tree is based on the underlying backbone pseudo tree $\mathcal{T}$. The root of the AND/OR search tree is an OR node labeled with the root of $\mathcal{T}$. A path from the root of the search tree $S_{\mathcal{T}}(\mathcal{R})$ to a node n is denoted by π_n. If n is labeled X_i or x_i the path will be denoted $\pi_n(X_i)$ or $\pi_n(x_i)$, respectively. The assignment sequence along path π_n, denoted $asgn(\pi_n)$, is the set of value assignments associated with the AND nodes along π_n:

$$asgn(\pi_n(X_i)) = \{\langle X_1, x_1\rangle, \langle X_2, x_2\rangle, ..., \langle X_{i-1}, x_{i-1}\rangle\}$$

$$asgn(\pi_n(x_i)) = \{\langle X_1, x_1\rangle, \langle X_2, x_2\rangle, ..., \langle X_i, x_i\rangle\}$$

The set of variables associated with OR nodes along the path π_n is denoted by $var(\pi_n)$: $var(\pi_n(X_i)) = \{X_1, ..., X_{i-1}\}$, $var(\pi_n(x_i)) = \{X_1, ..., X_i\}$. The parent-child relationship between nodes in the search space are defined as follows:

1. *An OR node, n, labeled by X_i has a child AND node labeled $\langle X_i, x_i\rangle$ iff $\langle X_i, x_i\rangle$ is consistent with $asgn(\pi_n)$, relative to the hard constraints.*

2. *An AND node, n, labeled by $\langle X_i, x_i\rangle$ has a child OR node labeled Y iff Y is a child of X_i in the backbone pseudo tree $\mathcal{T}$. Each OR arc, emanating from an OR to an AND node is associated with a weight to be defined shortly.*

Clearly, if a node n is labeled X_i (OR node) or x_i (AND node), $var(\pi_n)$ is the set of variables mentioned on the path from the root to X_i in the backbone pseudo tree, denoted by $path_{\mathcal{T}}(X_i)$.

Semantically, the OR states in the AND/OR search tree represent alternative ways of solving a problem, whereas the AND states represent problem decomposition into independent subproblems, conditioned on the assignment above them, all of which need to be solved.

Following the general definition of a solution tree for AND/OR search graphs [97] we have here that:

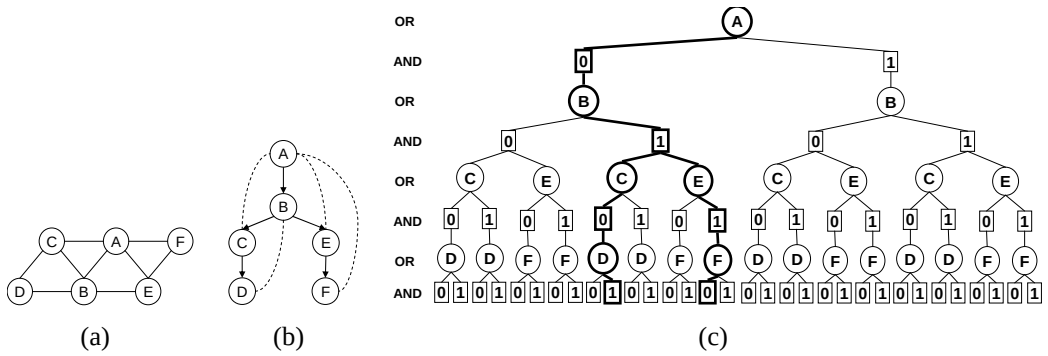


Figure 3.1: AND/OR search spaces for graphical models.

DEFINITION **23 (solution tree)** *A* solution tree *of an AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ is an AND/OR subtree T such that:*

1. *It contains the root of $S_{\mathcal{T}}(\mathcal{R})$, s;*
2. *If a non-terminal AND node $n \in S_{\mathcal{T}}(\mathcal{R})$ is in T then all of its children are in T;*
3. *If a non-terminal OR node $n \in S_{\mathcal{T}}(\mathcal{R})$ is in T then exactly one of its children is in T;*
4. *All its leaf (terminal) nodes are consistent.*

Example 6 *Figure 3.1(a) shows the primal graph of cost network with 6 bi-valued variables A, B, C, D, E and F, and 9 binary cost functions. Figure 3.1(b) displays a pseudo tree together with the back-arcs (dotted lines). Figure 3.1(c) shows the AND/OR search tree based on the pseudo tree. A solution tree is highlighted. Notice that once variables A and B are instantiated, the search space below the AND node $\langle B, 0\rangle$ decomposes into two independent subproblems, one that is rooted at C and one that is rooted at E, respectively.*

The virtue of an AND/OR search tree representation is that its size may be far smaller than the traditional OR search tree. It was shown that:

THEOREM **4 (size of AND/OR search trees [38])** *Given a graphical model $\mathcal{R}$ and a backbone pseudo tree $\mathcal{T}$, its AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$ is sound and complete, and its size is*

$O(l \cdot k^m)$ *where* m *is the depth of the pseudo tree,* l *bounds its number of leaves, and* k *bounds the domain size.*

Given a *tree decomposition* of the primal graph G having n nodes, whose treewidth is w^*, it is known that there exists a pseudo tree $\mathcal{T}$ of G whose depth, m, satisfies: $m \leq w^* \cdot logn$ [11, 5]. Therefore,

THEOREM **5 ([38])** *A graphical model that has a treewidth* w^* *has an AND/OR search tree whose size is* $O(n \cdot k^{w^* \cdot logn})$*, where* k *bounds the domain size and* n *is the number of variables.*

Weights of OR-AND Arcs

The arcs in the AND/OR trees are associated with weights that are defined based on the graphical model's functions and the combination operator. We next define arc weights for any graphical model using the notion of *buckets of functions* [38].

DEFINITION **24 (buckets relative to a pseudo tree)** *Given a graphical model* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ *and a backbone pseudo tree* $\mathcal{T}$*, the bucket of* X_i *relative to* $\mathcal{T}$*, denoted* $B_{\mathcal{T}}(X_i)$*, is the set of functions whose scopes contain* X_i *and are included in* $path_{\mathcal{T}}(X_i)$*, which is the set of variables from the root to* X_i *in* $\mathcal{T}$*. Namely,*

$$B_{\mathcal{T}}(X_i) = \{f \in \mathbf{F} | X_i \in scope(f), scope(f) \subseteq path_{\mathcal{T}}(X_i)\}$$

For simplicity and without loss of generality we consider in the remainder of the chapter a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ for which the combination and elimination operators are *summation* and *minimization*, respectively.

DEFINITION **25 (weights)** *Given an AND/OR search tree* $S_{\mathcal{T}}(\mathcal{R})$*, of a graphical model* $\mathcal{R}$*, the weight* $w_{(n,m)}(X_i, x_i)$ *(or simply* $w(X_i, x_i)$*) of arc* (n, m)*, where* X_i *labels* n *and* x_i

labels m, is the combination *of all the functions in $B_{\mathcal{T}}(X_i)$ assigned by values along π_m. Formally,*

$$w(X_i, x_i) = \begin{cases} 0 & \text{, if } B_{\mathcal{T}}(X_i) = \emptyset \\ \sum_{f \in B_{\mathcal{T}}(X_i)} f(asgn(\pi_m)) & \text{, otherwise} \end{cases}$$

DEFINITION **26 (cost of a solution tree)** *Given a weighted AND/OR search tree $S_{\mathcal{T}}(\mathcal{R})$, of a graphical model $\mathcal{R}$, and given a solution tree T having OR-to-AND set of arcs $arcs(T)$, the cost of T is defined by $f(T) = \sum_{e \in arcs(T)} w(e)$.*

Let $f(T_n)$ the cost of a solution tree rooted at node n. Then $f(T_n)$ can be computed recursively, as follows:

1. *If T_n consists only of a terminal AND node n, then $f(T_n) = 0$.*
2. *If n is an OR node having an AND child m in T_n, then $f(T_n) = w(n, m) + f(T_m)$, where T_m is the solution subtree of T_n that is rooted at m.*
3. *If n is an AND node having OR children $m_1, ..., m_k$ in T_n, then $f(T_n) = \sum_{i=1}^{k} f(T_{m_i})$, where T_{m_i} is the solution subtree of T_n rooted at m_i.*

Example 7 *Figure 3.2 shows the primal graph of a cost network with functions $\{f(A, B)$, $f(A, C)$, $f(A, B, E)$, $f(B, C, D)\}$, a pseudo tree that drives its weighted AND/OR search tree, and a portion of the AND/OR search tree with appropriate weights on the arcs expressed symbolically. In this case the bucket of E contains the function $f(A, B, E)$, the bucket of C contains two functions $f(A, C)$ and $f(B, C, D)$ and the bucket of B contains the function $f(A, B)$. We see indeed that the weights on the arcs from the OR node E to any of its AND value assignments include only the instantiated function $f(A, B, E)$, while the weights on the arcs connecting C to its AND child nodes are the sum of the two functions in its bucket instantiated appropriately. Notice that the buckets of A and D are empty and therefore the weights associated with the respective arcs are 0.*

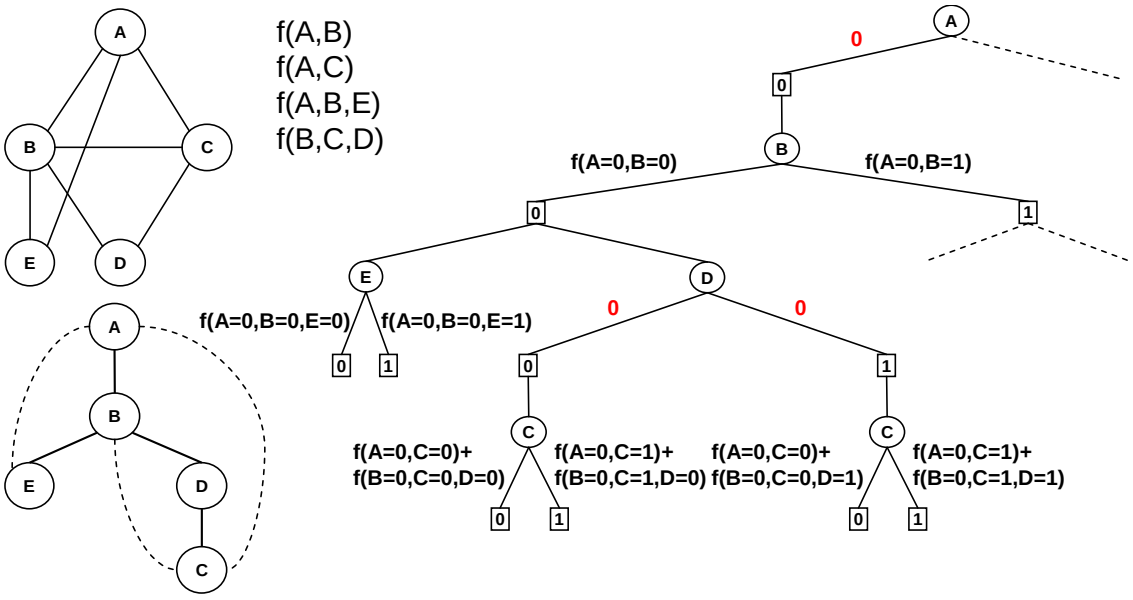


Figure 3.2: Arc weights for a cost network with 5 variables and 4 cost functions.

Node Value

With each node n of the search tree we can associate a value $v(n)$ which stands for the answer to the particular query restricted to the subproblem below n [38].

DEFINITION **27 (node value)** *Given an optimization problem $\mathcal{P} = \langle \mathcal{R}, min \rangle$ over a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$, the* value *of a node n in the AND/OR search tree $S_T(\mathcal{R})$ is the optimal cost to the subproblem below n.*

The value of a node can be computed recursively, as follows: it is 0 for terminal AND nodes and ∞ for terminal OR nodes, respectively. The value of an internal OR node is obtained by combining (summing) *the value of each AND child node with the weight on its incoming arc and then* optimize (minimize) *over all AND children. The value of an internal AND node is the* combination (summation) *of values of its OR children. Formally, if $succ(n)$ denotes the children of the node n in the AND/OR search tree, then:*

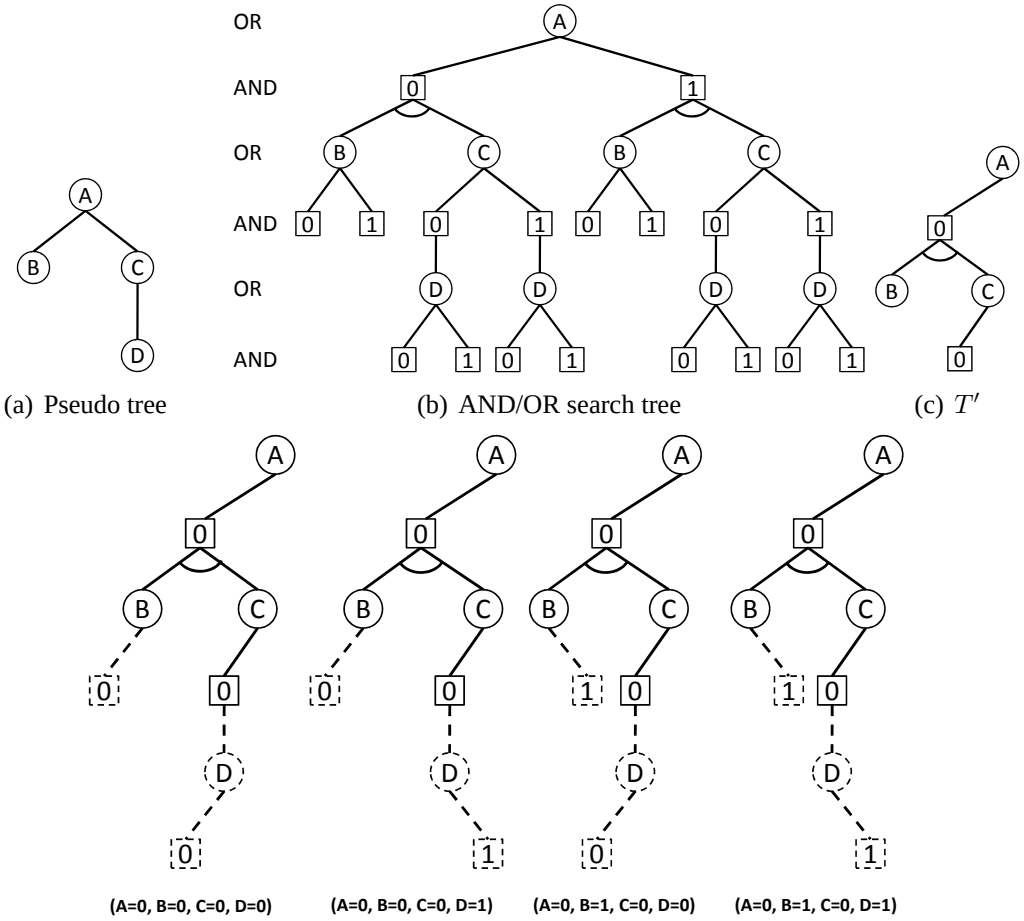
$$v(n) = \begin{cases} 0 & \text{, if } n = \langle X, x \rangle \text{ is a terminal AND node} \\ \infty & \text{, if } n = X \text{ is a terminal OR node} \\ \sum_{m \in succ(n)} v(m) & \text{, if } n = \langle X, x \rangle \text{ is an AND node} \\ \min_{m \in succ(n)} (w(n,m) + v(m)) & \text{, if } n = X \text{ is an OR node} \end{cases} \tag{3.1}$$

If n is the root of $S_{\mathcal{T}}(\mathcal{R})$, then $v(n)$ is the minimal cost solution to the initial problem. Alternatively, the value $v(n)$ can also be interpreted as the minimum of the costs of the solution trees rooted at n. Therefore, search algorithms that traverse the AND/OR search space can compute the value of the root node yielding the answer to the problem. It can be immediately inferred from Theorems 4 and 5 that:

THEOREM **6 (complexity [38])** *A depth-first search algorithm traversing an AND/OR tree for finding the minimal cost solution is time $O(n \cdot k^m)$, where n is the number of variables, k bounds the domain size and m is the depth of the pseudo tree, and may use linear space. If the primal graph has a tree decomposition with treewidth w^*, there there exists a pseudo tree $\mathcal{T}$ for which the time complexity is $O(n \cdot k^{w^* \cdot log n})$.*

3.3 AND/OR Branch-and-Bound Search

This section introduces the main contribution of the chapter which is an AND/OR Branch-and-Bound algorithm for AND/OR search spaces for graphical models. Traversing AND/OR search spaces by best-first algorithms or depth-first Branch-and-Bound was described as early as [97, 103, 62]. Here we adapt these algorithms to graphical models. We will revisit next the notion of partial solution trees [97] to represent sets of solution trees which will be used in our description.



(a) Pseudo tree (b) AND/OR search tree (c) T'

(d) Solution trees represented by T': $extension(T')$

Figure 3.3: A partial solution tree and possible extensions to solution trees.

DEFINITION **28 (partial solution tree)** *A* partial solution tree T' *of an AND/OR search tree* $S_{\mathcal{T}}$ *is a subtree which: (1) contains the root node* s *of* $S_{\mathcal{T}}$*; (2) if* n *in* T' *is an OR node then it contains at most one of its AND child nodes in* $S_{\mathcal{T}}$*, and if* n *is an AND node then it contains all its OR children in* $S_{\mathcal{T}}$ *or it has no child nodes. A node in* T' *is called a* tip *node if it has no children in* T'*. A tip node is either a* terminal *node (if it has no children in* $S_{\mathcal{T}}$*), or a* non-terminal *node (if it has children in* $S_{\mathcal{T}}$*).*

A partial solution tree can be extended (possibly in several ways) to a full solution tree. It represents $extension(T')$, the set of all full solution trees which can extend it. Clearly, a partial solution tree all of whose tip nodes are terminal in $S_{\mathcal{T}}$ is a solution tree.

Example 8 *Figure 3.3(c) shows a partial solution tree* T' *of the AND/OR search tree of*

Algorithm 5: AO: Depth-first AND/OR tree search

Input: An optimization problem $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum, min \rangle$, pseudo-tree $\mathcal{T}$ rooted at X_1.
Output: Minimal cost solution to $\mathcal{P}$ and an optimal solution tree.

```
1  v(s) ← ∞; ST(s) ← ∅; OPEN ← {s}                                  // Initialize the root node
2  while OPEN ≠ ∅ do
3      n ← top(OPEN); remove n from OPEN                                          // EXPAND
4      succ(n) ← ∅
5      if n is an OR node, labeled X_i then
6          foreach x_i ∈ D_i do
7              create an AND node n' labeled by ⟨X_i, x_i⟩
8              v(n') ← 0; ST(n') ← ∅
9              w(n, n') ← Σ_{f∈B_T(X_i)} f(asgn(π_n))          // Compute the OR-to-AND arc weight
10             succ(n) ← succ(n) ∪ {n'}
11     else if n is an AND node, labeled ⟨X_i, x_i⟩ then
12         foreach X_j ∈ children_T(X_i) do
13             create an OR node n' labeled by X_j
14             v(n') ← ∞; ST(n') ← ∅
15             succ(n) ← succ(n) ∪ {n'}
16     Add succ(n) on top of OPEN
                                                                              // PROPAGATE
17     while succ(n) == ∅ do
18         let p be the parent of n
19         if n is an OR node, labeled X_i then
20             if X_i == X_1 then
21                 return (v(n), ST(n))                                  // Search terminates
22             v(p) ← v(p) + v(n)                                         // Update AND value
23             ST(p) ← ST(p) ∪ ST(n)                  // Update solution tree below AND node
24         else if n is an AND node, labeled ⟨X_i, x_i⟩ then
25             if v(p) > (w(p, n) + v(n)) then
26                 v(p) ← w(p, n) + v(n)                                  // Update OR value
27                 ST(p) ← ST(n) ∪ {(X_i, x_i)}           // Update solution tree below OR node
28         remove n from succ(p)
29         n ← p
```

Figure 3.3(b) relative to the pseudo tree displayed in Figure 3.3(a). The set of solution trees represented by T' is given in Figure 3.3(d) and corresponds to the following assignments: $(A = 0, B = 0, C = 0, D = 0)$, $(A = 0, B = 0, C = 0, D = 1)$, $(A = 0, B = 1, C = 0, D = 0)$ *and* $(A = 0, B = 1, C = 0, D = 1)$.

Brute-force Depth-First AND/OR Tree Search

A simple depth-first search algorithm, called AO, that traverses the AND/OR search tree is described in Algorithm 5. The algorithm maintains the current partial solution being explored and will compute the value of each node (see Definition 27) in a depth-first manner. The value of the root node is the optimal cost. The algorithm also returns the optimal solu-

tion tree. It interleaves a forward expansion of the current partial solution tree (EXPAND) with a cost revision step (PROPAGATE) that updates the node values. The search stack is maintained by the OPEN list, n denotes the current node and p its parent in the search tree. Each node n in the search tree maintains its current value $v(n)$, which is updated based on the values of its children. For OR nodes, the current $v(n)$ is an upper bound on the optimal solution cost below n. Initially, $v(n)$ is set to ∞ if n is OR, and 0 if n is AND, respectively. A data structure $ST(n)$ maintains the actual best solution found in the subtree of n (as a list of value assignments to the variables in the respective subtree).

EXPAND selects a tip node n of the current partial solution tree and expands it by generating its successors. If n is an OR node, labeled X_i, then its successors are AND nodes represented by the values x_i in variable X_i's domain (lines 5–10). Each OR-to-AND arc is associated with the appropriate weight (see Definition 25). Similarly, if n is an AND node, labeled $\langle X_i, x_i \rangle$, then its successors are OR nodes labeled by the child variables of X_i in $\mathcal{T}$ (lines 11–15). There are no weights associated with AND-to-OR arcs.

PROPAGATE propagates node values bottom up in the search tree. Is is triggered when a node has an empty set of descendants (note that as each successor is evaluated, it is removed from the set of successors in line 28). This means that all its children have been evaluated, and their final values are already determined. If the current node is the root, then the search terminates with its value and an optimal solution tree (line 21). If n is an OR node, then its parent p is an AND node, and p updates its current value $v(p)$ by summation with the value of n (line 22). An AND node n propagates its value to its parent p in a similar way, by minimization (lines 25–27). Finally, the current node n is set to its parent p (line 29), because n was completely evaluated. Each node in the search tree also records the current best assignment to the variables of the subproblem below it and when the algorithm terminates it contains an optimal solution tree. Specifically, if n is an AND node, then $ST(n)$ is the union of the optimal solution trees propagated from n's OR children (line 23). If n is an OR node and n' is its AND child such that

$n' = argmin_{m \in succ(n)}(w(n,m) + v(m))$, then $ST(n)$ is obtained from the label of n' combined with the optimal solution tree below n' (line 27). Search continues either with a *propagation* step (if conditions are met) or with an *expansion* step.

Heuristic Lower Bounds on Partial Solution Trees

A regular OR Branch-and-Bound algorithm traverses the space of partial assignments in a depth-first manner and discards any partial assignment that cannot lead to a superior solution than the current best one found so far. This is normally achieved by using an evaluation function that underestimates (for minimization tasks) the best possible extension of the current partial path. Thus, when the estimated lower bound, called also heuristic evaluation function, is higher than the best current solution (upper bound), search terminates below this path.

We will now extend the brute-force AO algorithm into a Branch-and-Bound scheme, guided by a lower bound heuristic evaluation function. For that, we first define the exact evaluation function of a partial solution tree, and will then derive the notion of a lower bound for it. Like in OR search, we assume a given heuristic evaluation function $h(n)$ associated with each node n in the AND/OR search tree such that $h(n) \leq h^*(n)$, where $h^*(n)$ is the best cost extension of the subproblem below n (namely, $h^*(n) = v(n)$). We call $h(n)$ a *node-based heuristic function.*

DEFINITION **29 (exact evaluation function of a partial solution tree)** *The* exact evaluation function $f^*(T')$ *of a partial solution tree T' is the minimum of the costs of all solution trees represented by T', namely:*

$$f^*(T') = min\{f(T) \mid T \in extension(T')\}$$

We define $f^*(T'_n)$ the exact evaluation function of a partial solution tree rooted at node n. Then $f^*(T'_n)$ can be computed recursively, as follows:

1. *If T'_n consists of a single node n, then $f^*(T'_n) = v(n)$.*

2. *If n is an OR node having the AND child m in T'_n, then $f^*(T'_n) = w(n,m) + f^*(T'_m)$, where T'_m is the partial solution subtree of T'_n that is rooted at m.*

3. *If n is an AND node with OR children $m_1, ..., m_k$ in T'_n, then $f^*(T'_n) = \sum_{i=1}^{k} f^*(T'_{m_i})$, where T'_{m_i} is the partial solution subtree of T'_n rooted at m_i.*

Clearly, we are interested to find the $f^*(T')$ of a partial solution tree T' rooted at the root s. If each non-terminal tip node n of T' is assigned a heuristic lower bound estimate $h(n)$ of $v(n)$, then it induces a heuristic evaluation function on the minimal cost extension of T', as follows.

DEFINITION **30 (heuristic evaluation function of a partial solution tree)** *Given a node-based heuristic function $h(m)$ which is a lower bound on the optimal cost below any node m, namely $h(m) \leq v(m)$, and given a partial solution tree T'_n rooted at node n in the AND/OR search tree S_T, the* tree-based heuristic evaluation function *$f(T'_n)$ of T'_n, is defined recursively by:*

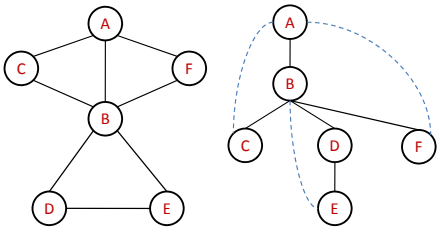
1. *If T'_n consists of a single node n then $f(T'_n) = h(n)$.*

2. *If n is an OR node having the AND child m in T'_n, then $f(T'_n) = w(n,m) + f(T'_m)$, where T'_m is the partial solution subtree of T'_n that is rooted at m.*

3. *If n is an AND node having OR children $m_1, ..., m_k$ in T'_n, then $f(T'_n) = \sum_{i=1}^{k} f(T'_{m_i})$, where T'_{m_i} is the partial solution subtree of T'_n rooted at m_i.*

Clearly, by definition:

PROPOSITION **1** *For any node n, $f(T'_n) \leq f^*(T'_n)$. If n is the root of the AND/OR search tree, then $f(T') \leq f^*(T')$.*



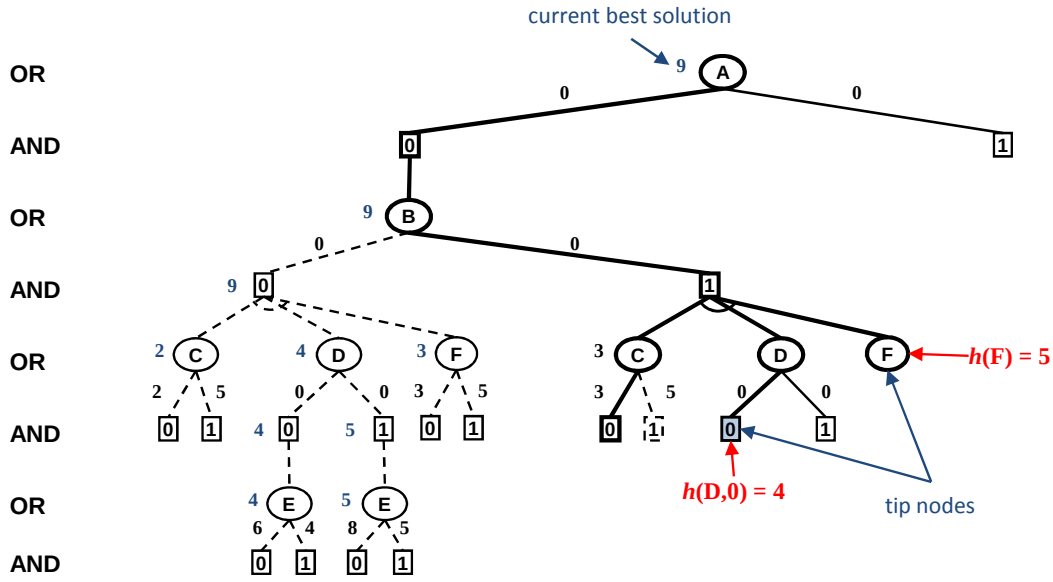
(a) Cost network and pseudo tree

A	B	C	f_1(ABC)
0	0	0	2
0	0	1	5
0	1	0	3
0	1	1	5
1	0	0	9
1	0	1	3
1	1	0	7
1	1	1	2

A	B	F	f_2(ABF)
0	0	0	3
0	0	1	5
0	1	0	1
0	1	1	4
1	0	0	6
1	0	1	5
1	1	0	6
1	1	1	5

B	D	E	f_3(BDE)
0	0	0	6
0	0	1	4
0	1	0	8
0	1	1	5
1	0	0	9
1	0	1	3
1	1	0	7
1	1	1	4

(b) Cost functions



(c) Partial solution tree

Figure 3.4: Cost of a partial solution tree.

Example 9 *Consider the cost network with bi-valued variables* A, B, C, D, E *and* F *in Figure 3.4(a). The cost functions* $f_1(A, B, C)$*,* $f_2(A, B, F)$ *and* $f_3(B, D, E)$ *are given in Figure 3.4(b). A partially explored AND/OR search tree relative to the pseudo tree from Figure 3.4(a) is displayed in Figure 3.4(c). The current partial solution tree* T' *is highlighted. It contains the nodes:* A*,* $\langle A, 0\rangle$*,* B*,* $\langle B, 1\rangle$*,* C*,* $\langle C, 0\rangle$*,* D*,* $\langle D, 0\rangle$ *and* F*. The nodes labeled by* $\langle D, 0\rangle$ *and by* F *are non-terminal tip nodes and their corresponding heuristic estimates are* $h(\langle D, 0\rangle) = 4$ *and* $h(F) = 5$*, respectively. The node labeled by* $\langle C, 0\rangle$ *is a terminal tip node of* T'*. The subtree rooted at* $\langle B, 0\rangle$ *along the path* $(A, \langle A, 0\rangle, B, \langle B, 0\rangle)$ *is fully explored, yielding the current best solution cost found so far equal to* 9*. We assume that the search is currently at the tip node labeled by* $\langle D, 0\rangle$ *of* T'*. The heuristic evaluation*

function of T' is computed recursively as follows:

$$\begin{aligned}
f(T') &= w(A,0) + f(T'_{\langle A,0\rangle}) \\
&= w(A,0) + f(T'_B) \\
&= w(A,0) + w(B,1) + f(T'_{\langle B,1\rangle}) \\
&= w(A,0) + w(B,1) + f(T'_C) + f(T'_D) + f(T'_F) \\
&= w(A,0) + w(B,1) + w(C,0) + f(T'_{\langle C,0\rangle}) + w(D,0) + f(T'_{\langle D,0\rangle}) + h(F) \\
&= w(A,0) + w(B,1) + w(C,0) + 0 + w(D,0) + h(\langle D,0\rangle) + h(F) \\
&= 0 + 0 + 3 + 0 + 0 + 4 + 5 \\
&= 12
\end{aligned}$$

Notice that if the pseudo tree $\mathcal{T}$ is a chain, then a partial tree T' is also a chain and corresponds to the partial assignment $\bar{x}^p = (x_1, ..., x_p)$. In this case, $f(T')$ is equivalent to the classical definition of the heuristic evaluation function of $\bar{x}^p$. Namely, $f(T')$ is the sum of the cost of the partial solution $\bar{x}^p$, $g(\bar{x}^p)$, and the heuristic estimate of the optimal cost extension of $\bar{x}^p$ to a complete solution.

During search we maintain an upper bound $ub(s)$ on the optimal solution $v(s)$ as well as the heuristic evaluation function of the current partial solution tree $f(T')$, and we can prune the search space by comparing these two measures, as is common in Branch-and-Bound search. Namely, if $f(T') \geq ub(s)$, then searching below the current tip node t of T' is guaranteed not to reduce $ub(s)$ and therefore, the search space below t can be pruned.

Example 10 *For illustration, consider again the partially explored AND/OR search tree from Example 9 (see Figure 3.4(c)). In this case, the current best solution found after exploring the subtree below $\langle B,0\rangle$, which ends the path $(A, \langle A,0\rangle, B, \langle B,0\rangle)$, is 9. Since we computed $f(T') = 12$ for the current partial solution tree highlighted in Figure 3.4(c), then exploring the subtree rooted at $\langle D,0\rangle$, which is the current tip node, cannot yield a*

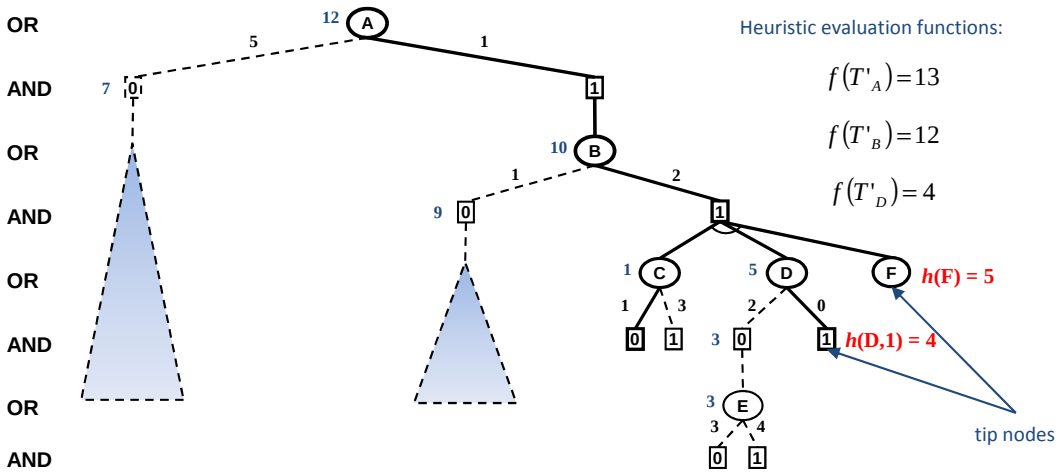


Figure 3.5: Illustration of the pruning mechanism.

better solution and search can be pruned.

Up until now we considered the case when the best solution found so far is maintained at the root node of the search tree. It is also possible to maintain the current best solutions for all the OR nodes along the active path between the tip node t of T' and s. Then, if $f(T'_m) \geq ub(m)$, where m is an OR ancestor of t in T' and T'_m is the subtree of T' rooted at m, it is also safe to prune the search tree below t. This provides an efficient mechanism to discover that the search space below a node can be pruned more quickly.

Example 11 *Consider the partially explored weighted AND/OR search tree in Figure 3.5, relative to the pseudo tree from Figure 3.4(a). The current partial solution tree T' is highlighted. It contains the following nodes: A, $\langle A,1\rangle$, B, $\langle B,1\rangle$, C, $\langle C,0\rangle$, D, $\langle D,1\rangle$ and F. The nodes labeled by $\langle D,1\rangle$ and by F are non-terminal tip nodes and their corresponding heuristic estimates are $h(\langle D,1\rangle) = 4$ and $h(F) = 5$, respectively. The subtrees rooted at the AND nodes labeled $\langle A,0\rangle$, $\langle B,0\rangle$ and $\langle D,0\rangle$ are fully evaluated, and therefore the current upper bounds of the OR nodes labeled A, B and D, along the active path, are $ub(A) = 12$, $ub(B) = 10$ and $ub(D) = 5$, respectively. Moreover, the heuristic evaluation functions of the partial solution subtrees rooted at the OR nodes along the current path can be computed recursively based on Definition 30, namely $f(T'_A) = 13$, $f(T'_B) = 12$*

and $f(T'_D) = 4$, respectively. Notice that while we could prune below $\langle D, 1\rangle$ because $f(T'_A) > ub(A)$, we could discover this pruning earlier by looking at node B only, because $f(T'_B) > ub(B)$. Therefore, the partial solution tree T'_A need not be consulted in this case.

Algorithm 6: AOBB: Depth-first AND/OR Branch-and-Bound search

Input: An optimization problem $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum, min\rangle$, pseudo-tree $\mathcal{T}$ rooted at X_1, heuristic function $h(n)$.
Output: Minimal cost solution to $\mathcal{P}$ and an optimal solution tree.

```
1  v(s) ← ∞; ST(s) ← ∅; OPEN ← {s}                                   // Initialize the root node
2  while OPEN ≠ ∅ do
3      n ← top(OPEN); remove n from OPEN                              // EXPAND
4      succ(n) ← ∅
5      if n is an OR node, labeled X_i then
6          foreach x_i ∈ D_i do
7              create an AND node n' labeled by ⟨X_i, x_i⟩
8              v(n') ← 0; ST(n') ← ∅
9              w(n, n') ← Σ_{f∈B_T(X_i)} f(asgn(π_n))              // Compute the OR-to-AND arc weight
10             succ(n) ← succ(n) ∪ {n'}
11     else if n is an AND node, labeled ⟨X_i, x_i⟩ then
12         deadend ← false
13         foreach OR ancestor m of n do
14             f(T'_m) ← evalPartialSolutionTree(T'_m)
15             if f(T'_m) ≥ v(m) then
16                 deadend ← true          // Pruning the subtree below the current tip node
17                 break
18         if deadend == false then
19             foreach X_j ∈ children_T(X_i) do
20                 create an OR node n' labeled by X_j
21                 v(n') ← ∞; ST(n') ← ∅
22                 succ(n) ← succ(n) ∪ {n'}
23         else
24             p ← parent(n)
25             succ(p) ← succ(p) − {n}
26     Add succ(n) on top of OPEN
                                                                      // PROPAGATE
27     while succ(n) == ∅ do
28         let p be the parent of n
29         if n is an OR node, labeled X_i then
30             if X_i == X_1 then
31                 return (v(n), ST(n))                              // Search terminates
32             v(p) ← v(p) + v(n)                                    // Update AND value
33             ST(p) ← ST(p) ∪ ST(n)                  // Update solution tree below AND node
34         if n is an AND node, labeled ⟨X_i, x_i⟩ then
35             if v(p) > (w(p, n) + v(n)) then
36                 v(p) ← w(p, n) + v(n)                             // Update OR value
37                 ST(p) ← ST(n) ∪ {(X_i, x_i)}          // Update solution tree below OR node
38         remove n from succ(p)
39         n ← p
```

Algorithm 7: Recursive computation of the heuristic evaluation function.

```
function: evalPartialSolutionTree(T'_n)
Input: Partial solution tree T'_n rooted at node n.
Output: Return heuristic evaluation function f(T'_n).
1  if succ(n) == ∅ then
2      return h(n)
3  else
4      if n is an AND node then
5          let m_1, ..., m_k be the OR children of n
6          return Σ_{i=1}^{k} evalPartialSolutionTree(T'_{m_i})
7      else if n is an OR node then
8          let m be the AND child of m
9          return w(n, m) + evalPartialSolutionTree(T'_m)
```

Depth-First AND/OR Branch-and-Bound Tree Search

The *AND/OR Branch-and-Bound* algorithm, AOBB, for searching AND/OR trees for graphical models, is described by Algorithm 6. Like AO, it interleaves a forward expansion of the current partial solution tree with a backward propagation step that updates the nodes values. The fringe of the search is maintained by a stack called OPEN, the current node is n, its parent p, and the current path π_n. As before, $ST(n)$ accumulates the current best solution tree below n. The node-based heuristic function $h(n)$ of $v(n)$ is assumed to be available to the algorithm, either retrieved from a cache or computed during search.

Before expanding the current AND node n, labeled $\langle X_i, x_i \rangle$, the algorithm computes the heuristic evaluation function for every partial solution subtree rooted at the OR ancestors of n along the path from the root (lines 11–17). The search below n is terminated if, for some OR ancestor m, $f(T'_m) \geq v(m)$, where $v(m)$ is the current best upper bound on the optimal cost below m. The recursive computation of $f(T'_m)$ based on Definition 30 is described in Algorithm 7. Notice also that for any OR node n, labeled X_i in the search tree, $v(n)$ is trivially initialized to ∞ and is updated in line 36.

The node values are updated by the propagation step, in the usual way (lines 24–40): OR nodes by minimization, while AND nodes by summation. The search terminates when the root node is evaluated in line 32.

THEOREM **7 (complexity)** *The time complexity of the depth-first AND/OR Branch-and-*

Bound algorithm (AOBB) is $O(n \cdot k^m)$, where m is the depth of the pseudo tree, k bounds the domain size and n is the number of variables, and it can use linear space.

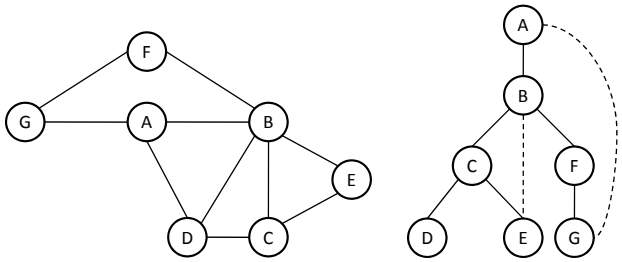
Proof. The time complexity follows immediately from the size of the AND/OR search tree explored (see Theorem 4). Since only the current partial solution tree needs to be stored in memory, the algorithm can operate in linear space. □

3.4 Lower Bound Heuristics for AND/OR Search

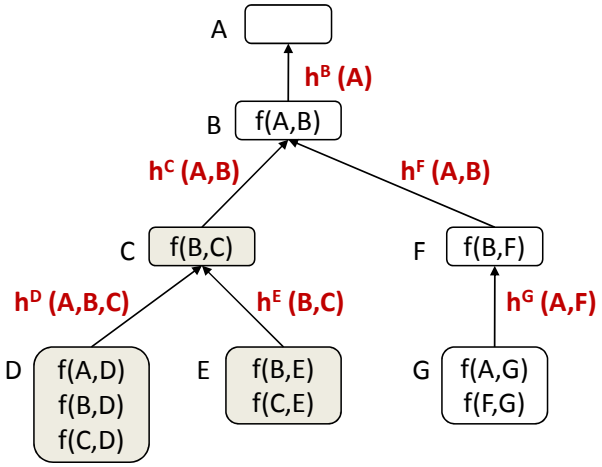
The effectiveness of any Branch-and-Bound search strategy greatly depends on the quality of the heuristic evaluation function. Naturally, more accurate heuristic estimates may yield a smaller search space, possibly at a much higher computational cost for computing the lower bound heuristic function. The right tradeoff between the computational overhead and the pruning power exhibited during search may be hard to predict. One of the primary heuristics we used is the Mini-Bucket heuristic introduced in [65] for OR search spaces. In the following subsections we discuss its extension to AND/OR search spaces. We also extend the local consistency based lower bound developed in [71, 72, 25] to AND/OR search spaces. Both of these heuristic functions were used in our experiments.

3.4.1 Static Mini-Bucket Heuristics

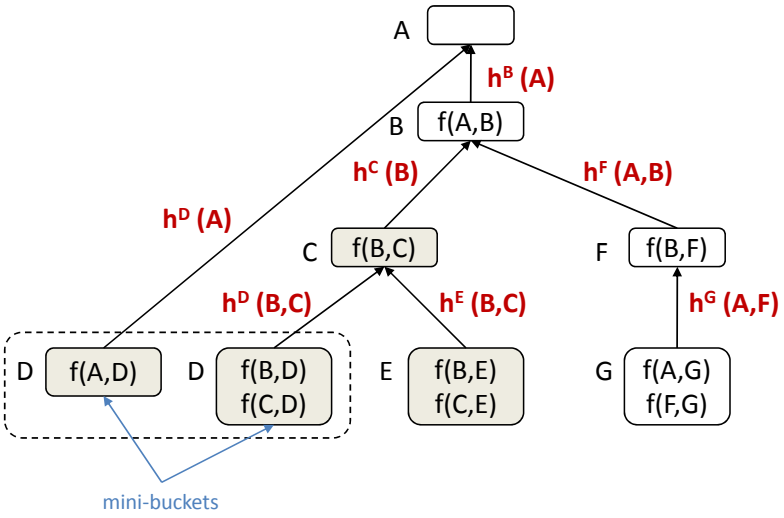
Consider the cost network and pseudo tree shown in Figures 3.6(a) and 3.6(b), respectively, and consider also the variable ordering $d = (A, B, C, D, E, F, G)$ and the bucket and mini-buckets configuration in the output as displayed in Figures 3.6(c) and 3.6(d), respectively (see also Chapter 1, Sections 1.3.1 and 1.3.2 for a more details). For clarity, we display the execution of the Bucket and Mini-Bucket Elimination algorithms along the bucket tree corresponding to the given elimination ordering. The bucket tree is also a pseudo tree [38].



(a) Cost network (b) Pseudo tree



(c) Bucket Elimination



(d) Mini-Bucket Elimination MBE(3)

Figure 3.6: Static mini-bucket heuristics for $i = 3$.

The functions denoted on the arcs are those messages sent from a bucket node to its parent in the tree.

Let us assume, without loss of generality, that variables A and B have been instantiated during search. Let $h^*(a, b, c)$ be the minimal cost solution of the subproblem rooted at node C in the pseudo tree, conditioned on $(A = a, B = b, C = c)$. In the AND/OR search tree, this is represented by the subproblem rooted at the AND node labeled $\langle C, c\rangle$, ending the path $\{A, \langle A, a\rangle, B, \langle B, b\rangle, C, \langle C, c\rangle\}$. By definition,

$$h^*(a, b, c) = min_{d,e}(f(c, e) + f(b, e) + f(a, d) + f(c, d) + f(b, d)) \tag{3.2}$$

Notice that we restrict ourselves to the subproblem over variables D and E only. Therefore, we obtain:

$$\begin{aligned} h^*(a, b, c) &= min_d(f(a, d) + f(c, d) + f(b, d) + min_e(f(c, e) + f(b, e))) \\ &= min_d(f(a, d) + f(c, d) + f(b, d)) + min_e(f(c, e) + f(b, e)) \\ &= h^D(a, b, c) + h^E(b, c) \end{aligned}$$

where,

$$\begin{aligned} h^D(a, b, c) &= min_d(f(a, d) + f(c, d) + f(b, d)) \\ h^E(b, c) &= min_e(f(c, e) + f(b, e)) \end{aligned}$$

Notice that the functions $h^D(a, b, c)$ and $h^E(b, c)$ are produced by the bucket elimination algorithm shown in Figure 3.6(c). Specifically, the function $h^D(a, b, c)$, generated in bucket of D by bucket elimination, is the result of a minimization operation over variable D. In practice, however, this function may be too hard to compute as it requires processing a

function on four variables. It can be replaced by a partition-based approximation ($e.g.$, the minimization is split into two parts). This yields a lower bound approximation, denoted by $h(a, b, c)$, namely:

$$\begin{aligned} h^*(a,b,c) &= min_d(f(a,d) + f(c,d) + f(b,d)) + h^E(b,c) \\ &\geq min_d f(a,d) + min_d(f(c,d) + f(b,d)) + h^E(b,c) \\ &= h^D(a) + h^D(b,c) + h^E(b,c) \\ &\triangleq h(a,b,c) \end{aligned}$$

where,

$$\begin{aligned} h^D(a) &= min_d f(a,d) \\ h^D(c,b) &= min_d(f(c,d) + f(b,d)) \end{aligned}$$

The functions $h^D(a)$ and $h^D(b, c)$ are the ones computed by the Mini-Bucket algorithm MBE(3), shown in Figure 3.6(d). Therefore, the function $h(a, b, c)$ can be constructed during search from the pre-compiled mini-buckets, yielding a lower bound on the minimal cost of the respective subproblem.

For OR nodes, such as n, labeled by C, ending the path $\{A, \langle A, a\rangle, B, \langle B, b\rangle, C\}$, $h(n)$ can be obtained by minimizing over the values $c \in D_C$ the sum between the weight $w(n, m)$ and the heuristic estimate $h(m)$ below the AND child m of n, labeled $\langle C, c\rangle$. Namely, $h(n) = min_m(w(n, m) + h(m))$.

In summary, similarly to [65], we can show that the mini-bucket heuristic associated with any node in the AND/OR search tree can be obtained from the the pre-compiled mini-bucket functions.

DEFINITION **31 (static mini-bucket heuristic)** *Given an ordered set of augmented buck-*

ets $\{B(X_1), ..., B(X_n)\}$ *generated by the Mini-Bucket algorithm MBE(*i*) along the bucket tree* $\mathcal{T}$*, and given a node* n *in the AND/OR search tree, the* static mini-bucket heuristic *function* $h(n)$ *is computed as follows:*

1. *If* n *is an AND node, labeled by* $\langle X_p, x_p \rangle$*, then:*

$$h(n) = \sum_{h_j^k \in \{B(X_p) \cup B(X_p^1 .. X_p^q)\}} h_j^k$$

Namely, it is the sum of the intermediate functions h_j^k *that satisfy the following two properties:*

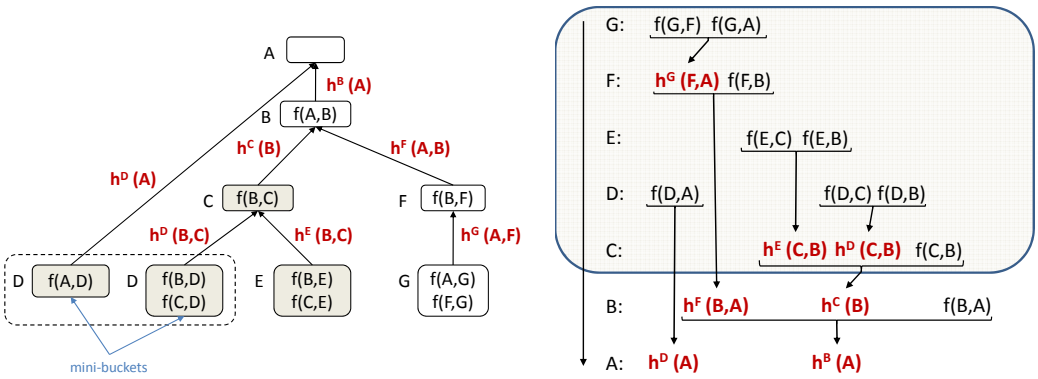
- *They are generated in buckets* $B(X_k)$*, where* X_k *is any descendant of* X_p *in the bucket tree* $\mathcal{T}$*,*
- *They reside in bucket* $B(X_p)$ *or the buckets* $B(X_p^1 .. X_p^q) = \{B(X_p^1), ..., B(X_p^q)\}$ *that correspond to the ancestors* $\{X_p^1, ..., X_p^q\}$ *of* X_p *in* $\mathcal{T}$*.*

2. *If* n *is an OR node, labeled by* X_p*, then:*

$$h(n) = min_m(w(n, m) + h(m))$$

where m *is the AND child of* n *labeled with value* x_p *of* X_p*.*

Example 12 *Figure 3.6(d) shows the bucket tree for the cost network in Figure 3.6(a) together with the intermediate functions generated by MBE(3) along the ordering* $d = (A, B, C, D, E, F, G)$*. The static mini-bucket function* $h(a', b', c')$ *associated with the AND node labeled* $\langle C, c' \rangle$ *ending the path* $(A = a', B = b', C = c')$ *in the AND/OR search tree is by definition* $h(a', b', c') = h^D(a') + h^D(c', b') + h^E(b', c')$*. The intermediate functions* $h^D(c', b')$ *and* $h^E(b', c')$ *are generated in buckets* D *and* E*, respectively, and reside in bucket* C*. The function* $h^D(a')$ *is also generated in bucket* D*, but it resides in bucket* A*, which is an ancestor of bucket* C *in the bucket tree.*



(a) AND/OR static mini-buckets (b) OR static mini-buckets

Figure 3.7: AND/OR versus OR static mini-bucket heuristics for $i = 3$.

We see that the computation of the static mini-bucket heuristic of a node n in the AND/OR search tree is identical to the OR case (see Definition 15 in Chapter 1), except that it only considers the intermediate functions generated by the buckets corresponding to the current conditioned subproblem rooted at n.

Example 13 *For example, consider again the cost network in Figure 3.6(a). Figures 3.7(a) (which repeats Figure 3.6(d)) and 3.7(b) show the compiled bucket structure obtained by MBE(3) along the given elimination order* $d = (A, B, C, D, E, F, G)$*, for the AND/OR and OR spaces, respectively. The static mini-bucket heuristic function underestimating the minimal cost extension of the partial assignment* $(A = a', B = b', C = c')$ *in the OR search space is* $h(a', b', c') = h^D(a') + h^D(c', b') + h^E(b', c') + h^F(b', a')$*. Namely, it involves the extra function* $h^F(b', a')$ *which was generated in bucket* F *and resides in bucket* B*, as shown in Figure 3.7(b). This is because, in the OR space, variables* F *and* G *are part of the subproblem rooted at* C*, unlike the AND/OR search space.*

3.4.2 Dynamic Mini-Bucket Heuristics

It is also possible to generate the mini-bucket heuristic information dynamically during search, as we show next. The idea is to compute MBE(i) conditioned on the current partial assignment.

DEFINITION **32 (dynamic mini-bucket heuristics)** *Given a bucket tree $\mathcal{T}$ with buckets $\{B(X_1), ..., B(X_n)\}$, a node n in the AND/OR search tree and given the current partial assignment $asgn(\pi_n)$ along the path to n, the* dynamic mini-bucket heuristic *function $h(n)$ is computed as follows:*

1. *If n is an AND node labeled by $\langle X_p, x_p \rangle$, then:*

$$h(n) = \sum_{h_j^k \in B(X_p)} h_j^k$$

Namely, it is the sum of the intermediate functions h_j^k that reside in bucket $B(X_p)$ and were generated by MBE(i), conditioned on $asgn(\pi_n)$, in buckets $B(X_p^1)$ through $B(X_p^q)$, where $\{X_p^1, ..., X_p^q\}$ are the descendants of X_p in $\mathcal{T}$.

2. *If n is an OR node labeled by X_p, then:*

$$h(n) = min_m(w(n,m) + h(m))$$

where m is the AND child of n labeled with value x_p of X_p.

Given an i-bound, the dynamic mini-bucket heuristic implies a much higher computational effort compared with the static version. However, the bounds generated dynamically may be far more accurate since some of the variables are assigned and will therefore yield smaller functions and less partitioning. More importantly, the dynamic mini-bucket heuristic can be used with dynamic variable ordering heuristics, unlike the pre-compiled one, which restricts search to be conducted in an order that respects a static pseudo tree structure.

Example 14 *Figure 3.8 shows the bucket tree structure corresponding to the binary cost network displayed in Figure 3.6(a), along the elimination ordering (A, B, C, D, E, F, G). The dynamic mini-bucket heuristic estimate $h(a', b', c')$ of the AND node labeled $\langle C, c' \rangle$*

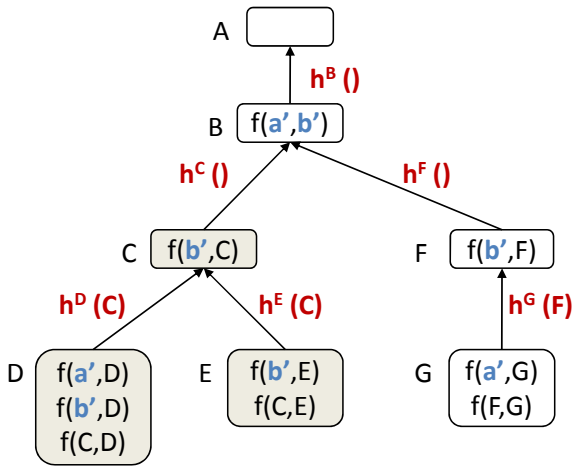


Figure 3.8: Dynamic mini-bucket heuristics for $i = 3$.

ending the path $\{A, \langle A, a' \rangle, B, \langle B, b' \rangle, C, \langle C, c' \rangle\}$ *is computed by MBE(*3*) on the subproblem represented by the buckets* D *and* E*, conditioned on the partial assignment* $(A = a', B = b', C = c')$*. Namely, MBE(*3*) processes buckets* D *and* E *by eliminating the respective variables, and generates two new functions:* $h^D(c')$ *and* $h^E(c')$*, as illustrated in Figure 3.8. These new functions are in fact constants since variables* A*,* B *and* C *are assigned in the scopes of the input functions that constitute the conditioned subproblem:* $f(a', D)$*,* $f(b', D)$*,* $f(c', D)$*,* $f(b', E)$ *and* $f(c', E)$*, respectively. Therefore* $h(a', b', c') = h^D(c') + h^E(c')$ *and it equals the exact* $h^*(a', b', c')$ *in this case.*

3.4.3 Local Consistency Based Heuristics for AND/OR Search

Another class of heuristic lower bounds developed for guiding Branch-and-Bound search for solving binary Weighted CSPs is based on exploiting local consistency algorithms for cost functions. We discuss next its extension to AND/OR trees.

Extension of Local Consistency to AND/OR Search Spaces

As mentioned in Chapter 1, the zero-arity constraint $C_\emptyset$ which is obtained by enforcing local consistency, can be used as a heuristic function to guide Branch-and-Bound search. The extension of this heuristic to AND/OR search spaces is fairly straightforward and is similar

to the extension of the mini-bucket heuristics from OR to AND/OR spaces. Consider P_n, the subproblem rooted at the AND node n, labeled $\langle X_i, x_i \rangle$, in the AND/OR search tree defined by a pseudo tree $\mathcal{T}$. The heuristic function $h(n)$ underestimating $v(n)$ is the zero-arity cost function $C_\emptyset^n$ resulted from enforcing soft arc consistency over P_n only, subject to the current partial instantiation of the variables along the path from the root of the search tree. Note that P_n is defined by the variables and cost functions corresponding to the subtree rooted at X_i in $\mathcal{T}$. If n is an OR node labeled X_i then $h(n)$ is computed in the usual way, namely $h(n) = min_m(w(n,m) + h(m))$, where m is the AND child of n, labeled with value x_i of X_i. Notice that in this case the weights associated with the OR-to-AND arcs are computed now relative to the equivalent subproblem resulted from enforcing arc consistency.

3.5 Dynamic Variable Orderings

The depth-first AND/OR Branch-and-Bound algorithm introduced in Section 3.3 assumed a static variable ordering determined by the underlying pseudo tree of the primal graph. In classical CSPs, dynamic variable ordering is known to have a significant impact on the size of the search space explored [34]. Well known variable ordering heuristics, such as *min-domain* [54], *min-dom/ddeg* [8], *brelaz* [12] and *min-dom/wdeg* [44, 14] were shown to improve dramatically the performance of systematic search algorithms. In this section we discuss some strategies that allow dynamic variable orderings in AND/OR search.

We distinguish two classes of variable ordering heuristics:

1. *Graph*-based heuristics ($e.g.$, pseudo tree) that try to maximize problem decomposition, and

2. *Semantic*-based heuristics ($e.g.$, min-domain) that aim at shrinking the search space, based on context and current value assignment.

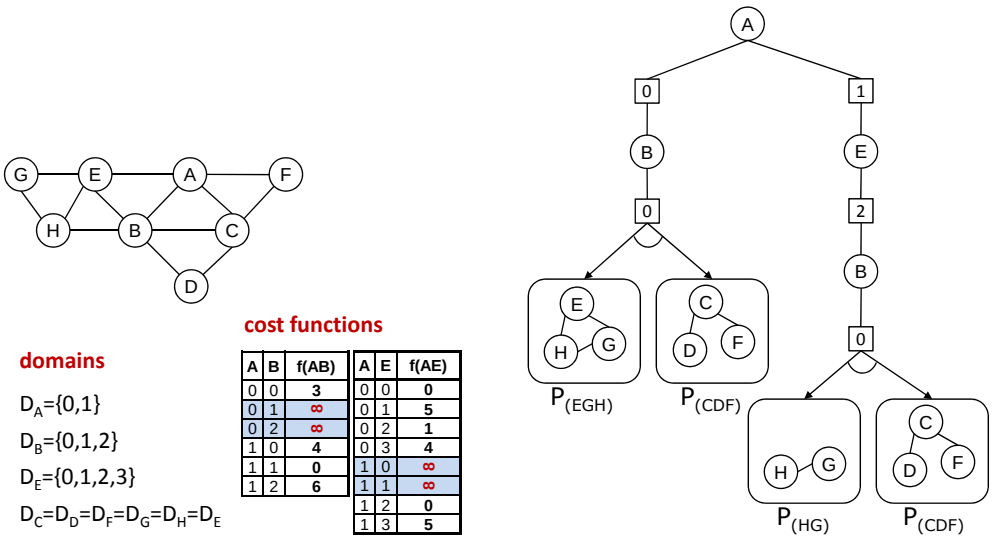
These two approaches are orthogonal, namely we can use one as the primary guide and break ties based on the other. We present three schemes of combining these heuristics. For simplicity and without loss of generality we consider the *min-domain* as our semantic variable ordering heuristic. It selects the next variable to instantiate as the one having the smallest current domain among the uninstantiated (future) variables. Clearly, it can be replaced by any other heuristic.

3.5.1 Partial Variable Ordering (PVO)

The first approach, called *AND/OR Branch-and-Bound with Partial Variable Ordering* and denoted by AOBB+PVO uses the static graph-based decomposition given by a pseudo tree with a dynamic semantic ordering heuristic applied over chain portions of the pseudo tree. It is an adaptation of the ordering heuristics developed in [56, 76] which were used for solving large-scale SAT problem instances.

Consider the pseudo tree from Figure 3.1(a) inducing the following variable groups (or chains): $\{A, B\}$, $\{C, D\}$ and $\{E, F\}$, respectively. This implies that variables $\{A, B\}$ should be considered before $\{C, D\}$ and $\{E, F\}$. The variables in each group can be dynamically ordered based on a second, independent heuristic. Notice that once variables $\{A, B\}$ are instantiated, the problem decomposes into independent components that can be solved separately.

AOBB+PVO can be derived from Algorithm 6 with some simple modifications. As usual, the algorithm traverses an AND/OR search tree in a depth-first manner, guided by a pre-computed pseudo tree $\mathcal{T}$. When the current AND node n, labeled $\langle X_i, x_i \rangle$ is expanded in the forward step (line 9), the algorithm generates its OR successor, labeled by X_j, based on the semantic variable ordering heuristic (line 12). Specifically, the OR node m, labeled X_j corresponds to the uninstantiated variable with the smallest current domain in the current pseudo tree chain. If there are no uninstantiated variables left in the current chain, namely variable X_i was instantiated last, then the OR successors of n are labeled by the variables



(a) Cost network (b) AND/OR search tree

Figure 3.9: Full dynamic variable ordering for AND/OR Branch-and-Bound search.

with the smallest domain from the variable groups rooted by X_i in $\mathcal{T}$.

3.5.2 Full Dynamic Variable Ordering (DVO)

A second, orthogonal approach to partial variable orderings, called *AND/OR Branch-and-Bound with Full Dynamic Variable Ordering* and denoted by DVO+AOBB, gives priority to the dynamic semantic variable ordering heuristic and applies static problem decomposition as a secondary principle during search. This idea was also explored in [6] for model counting, and more recently in [119] for weighted model counting.

For illustration, consider the cost network with 8 variables $\{A, B, C, D, E, F, G, H\}$, 13 binary cost functions, and the domains given in Figure 3.9(a), as follows: $D_A = \{0, 1\}$, $D_B = \{0, 1, 2\}$, and $D_C = D_D = D_E = D_F = D_G = D_H = \{0, 1, 2, 3\}$, respectively. Each of the cost functions $f(A, B)$ and $f(A, E)$ assigns an ∞ cost to two of their corresponding tuples, whereas the remaining 11 functions do not contain such tuples.

During search, variables are instantiated in min-domain order. However, after each variable assignment we test for problem decomposition and solve the remaining subproblems independently. Figure 3.9(b) shows the partial AND/OR search tree obtained after several

variable instantiations based on the min-degree ordering heuristic. Notice that, depending on the order in which the variables are instantiated, the primal graph may decompose into independent components *higher* or *deeper* in the search tree. For instance, after instantiating A to 0, the values $\{1, 2\}$ can be removed from the domain of B, because the corresponding tuples have cost ∞ in the cost function $f(A, B)$ (see Figure 3.9(a)). Therefore, B is the next variable to be instantiated, at which point the problem decomposes into independent components, as shown in Figure 3.9(b). Similarly, when A is instantiated to 1, values $\{0, 1\}$ can also be removed from the domain of E, because of the cost function $f(A, E)$. Then, variable E, having 2 values left in its domain, is selected next in the min-domain order, followed by B with domain size 3.

DVO+AOBB can be expressed by modifying Algorithm 6 as follows. It instantiates the variables dynamically using the min-domain ordering heuristic while maintaining the current graph structure. Specifically, after the current AND node n, labeled $\langle X_i, x_i \rangle$, is expanded, DVO+AOBB tentatively removes from the primal graph all nodes corresponding to the instantiated variables together with their incoming arcs. If disconnected components are detected, their corresponding subproblems are then solved separately and the results combined in an AND/OR manner. In this case a variable selection may yield a significant impact on tightening the search space, yet, it may not yield a good decomposition for the remaining problem.

3.5.3 Dynamic Separator Ordering (DSO)

The third approach, called *AND/OR Branch-and-Bound with Dynamic Separator Ordering* and denoted by AOBB+DSO, exploits constraint propagation which can be used for dynamic graph-based decomposition with a dynamic semantic variable ordering, giving priority to the first. At each AND node we apply a lookahead procedure hoping to detect singleton variables ($i.e.$, with only one feasible value left in their domains). When the value of a variable is known, it can be removed from the corresponding subproblem, yielding a

stronger decomposition of the simplified primal graph.

AOBB+DSO defined on top of Algorithm 6 creates and maintains a separator S of the current primal graph. A graph separator can be computed using the hypergraph partitioning method presented in [76]. The next variable is chosen dynamically from S by the min-domain ordering heuristic until S is fully instantiated and the current problem decomposes into several independent subproblems, which are then solved separately. The separator of each component is created from a simplified subgraph resulted from previous constraint propagation steps and it may differ for different value assignments. Clearly, if no singleton variables are discovered by the lookahead steps this approach is computationally identical to AOBB+PVO, although it may have a higher overhead due to the dynamic generation of the separators.

3.6 Experimental Results

We have conducted a number of experiments on two common optimization problem classes in graphical models: finding the Most Probable Explanation in Bayesian networks and solving Weighted CSPs. We implemented our algorithms in C++ and carried out all experiments on a 1.8GHz dual-core Athlon64 with 2GB of RAM running Ubuntu Linux 7.04.

3.6.1 Overview and Methodology

Bayesian Networks. For the MPE task, we tested the performance of the AND/OR Branch-and-Bound algorithms on the following types of problems: random Bayesian networks, random coding networks, grid networks, Bayesian networks derived from the ISCAS'89 digital circuit benchmark, genetic linkage analysis networks, networks from the Bayesian Network Repository, and Bayesian networks from the UAI'06 Inference Evaluation Dataset.

The detailed outline of the experimental evaluation for Bayesian networks is given

Benchmarks	static mini-buckets BB+SMB(i) AOBB+SMB(i)	dynamic mini-buckets BB+DMB(i) AOBB+DMB(i)	min-fill vs. hypergraph pseudo trees	constraint propagation	SamIam	Superlink
Random BN	√	√	√	-	-	-
Coding	√	√	√	-	√	-
Grids	√	√	√	√	√	-
Linkage	√	-	√	√	√	√
ISCAS'89	√	√	√	√	√	-
UAI'06 Dataset	√	-	√	-	√	-
BN Repository	√	√	-	-	√	-

Table 3.1: Detailed outline of the experimental evaluation for Bayesian networks.

in Table 3.1. We evaluated the two classes of depth-first AND/OR Branch-and-Bound search algorithms, guided by the static and dynamic mini-bucket heuristics, denoted by `AOBB+SMB(`i`)` and `AOBB+DMB(`i`)`, respectively. We compare these algorithms against traditional depth-first OR Branch-and-Bound algorithms with static and dynamic mini-bucket heuristics introduced in [65, 86], denoted by `BB+SMB(`i`)` and `BB+DMB(`i`)`, respectively, which were among the best-performing complete search algorithms for this domain at the time. The parameter i represents the mini-bucket i-bound and controls the accuracy of the heuristic. The pseudo trees that guide AND/OR search algorithms were generated using the min-fill and hypergraph partitioning heuristics, described later in this section. We also consider an extension of the AND/OR Branch-and-Bound that exploits the determinism present in the Bayesian network by constraint propagation.

Since the pre-compiled mini-bucket heuristics require a static variable ordering, the corresponding OR and AND/OR search algorithms used the variable ordering as well derived from a depth-first traversal of the guiding pseudo tree. When we applied dynamic variable orderings with dynamic mini-bucket heuristics we observed that the computational overhead was prohibitively large compared with the static variable ordering setup. We therefore do not report these results. We note however that the `AOBB+SMB(`i`)` and `AOBB+DMB(`i`)` algorithms support a restricted form of dynamic variable and value ordering. Namely, there is a dynamic internal ordering of the successors of the node just expanded, before placing them onto the search stack. Specifically, in line 26 of Algorithm 6, if the current node n is AND, then the independent subproblems rooted by its OR children can be solved in de-

Benchmarks	static mini-buckets BB+SMB(i) AOBB+SMB(i)	dynamic mini-buckets BB+DMB(i) AOBB+DMB(i)	min-fill vs. hypergraph pseudo trees	EDAC heuristics BBEDAC AOEDAC, PVO, DVO, DSO	toolbar
SPOT5	√	√	√	√	√
ISCAS'89	√	√	√	√	√
Mastermind	√	-	√	√	√
CELAR	-	-	-	√	√

Table 3.2: Detailed outline of the experimental evaluation for Weighted CSPs.

creasing order of their corresponding heuristic estimates (variable ordering). Alternatively, if n is OR, then its AND children corresponding to domain values can also be sorted in decreasing order of their heuristic estimates (value ordering).

We compared our algorithms with the SAMIAM 2.3.2 software package[1]. SAMIAM is a public implementation of Recursive Conditioning [24] which can also be viewed as an AND/OR search algorithm. The algorithm uses a context-based caching mechanism that records the optimal solution of the subproblems and retrieves the saved values when the same subproblems are encountered again during search. This version of recursive conditioning traverses a context minimal AND/OR search graph [38], rather than a tree, and its space complexity is exponential in the treewidth. Note that when we use mini-bucket heuristics with high values of i, we use space exponential in i for the heuristic calculation and storing. Our search regime however does not consume any additional space.

Weighted CSPs. For WCSPs we evaluated the performance of the depth-first AND/OR Branch-and-Bound algorithms on: random binary WCSPs, scheduling problems from the SPOT5 benchmark, networks derived from the ISCAS'89 digital circuits, radio link frequency assignment problems and instances of the Mastermind game.

The outline of the experimental evaluation for Weighted CSPs is detailed in Table 3.2. In addition to the mini-bucket heuristics, we also consider a heuristic evaluation function that is computed by maintaining Existential Directional Arc-Consistency (EDAC) [25]. `AOBB` with this heuristic is called `AOEDAC`. We also consider the extension of

[1]Available at http://reasoning.cs.ucla.edu/samiam. We used the `batchtool 1.5` provided with the package.

AOEDAC that incorporates dynamic variable orderings heuristics described earlier yielding: AOEDAC+PVO (partial variable ordering), DVO+AOEDAC (full dynamic variable ordering) and AOEDAC+DSO (dynamic separator ordering). For comparison, we report results obtained with our implementation of the classic OR Branch-and-Bound with EDAC, denoted here by BBEDAC.

For reference, we also ran the state-of-the-art solver called `toolbar`[2], which is the implementation of the OR Branch-and-Bound maintaining EDAC introduced in [25]. toolbar is currently one of the best performing solver for binary Weighted CSPs.

The semantic-based dynamic variable ordering heuristic used by the OR and AND/OR Branch-and-Bound algorithms with EDAC based heuristics was the *min-dom/ddeg* heuristic, which selects the variable with the smallest ratio of the current domain size divided by the future degree. Ties were broken lexicographically.

Measures of Performance. In all our experiments we report the average CPU time in seconds and the number of nodes visited, required for proving optimality. We also report problem's parameters as the number of variables (n), number of evidence variables (e), maximum domain size (k), the depth of the pseudo tree (h) and the induced width of the graph (w^*). When evidence is asserted in the network, w^* and h are computed after the evidence nodes are removed from the graph. We also report the time required by the Mini-Bucket algorithm MBE(i) to pre-compile the heuristic information. The best performance points are highlighted. In each table, "-" denotes that the respective algorithm exceeded the time limit. Similarly, "out" stands for exceeding the 2GB memory limit.

3.6.2 Finding Good Pseudo Trees

The performance of the AND/OR Branch-and-Bound search algorithms is influenced by the quality of the guiding pseudo tree. Finding the minimal depth/induced width pseudo

[2] Available at: http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP

Network	hypergraph		min-fill		Network	hypergraph		min-fill	
	width	depth	width	depth		width	depth	width	depth
barley	7	13	7	23	spot_5	47	152	39	204
diabetes	7	16	4	77	spot_28	108	138	79	199
link	21	40	15	53	spot_29	16	23	14	42
mildew	5	9	4	13	spot_42	36	48	33	87
munin1	12	17	12	29	spot_54	12	16	11	33
munin2	9	16	9	32	spot_404	19	26	19	42
munin3	9	15	9	30	spot_408	47	52	35	97
munin4	9	18	9	30	spot_503	11	20	9	39
water	11	16	10	15	spot_505	29	42	23	74
pigs	11	20	11	26	spot_507	70	122	59	160

Table 3.3: Bayesian Networks Repository (left); SPOT5 benchmarks (right).

tree is a hard problem [48, 11, 106]. We describe next two heuristics for generating pseudo trees with relatively small depths/induced-widths which we used in our experiments.

Min-Fill Heuristic

Min-Fill [67] is one of the best and most widely used heuristics for creating small induced width elimination orders. An ordering is generated by placing the variable with the smallest *fill set* ($i.e.$, number of induced edges that need be added to fully connect the neighbors of a node) at the end of the ordering, connecting all of its neighbors and then removing the variable from the graph. The process continues until all variables have been eliminated.

Once an elimination order is given, the pseudo tree can be extracted as a depth-first traversal of the min-fill induced graph, starting with the variable that initiated the ordering, always preferring as successor of a node the earliest adjacent node in the induced graph. An ordering uniquely determines a pseudo tree. This approach was first used by [106].

To improve orderings, we can run the min-fill ordering several times by randomizing the tie breaking. In our experiments, we ran the min-fill heuristic just once and broke the ties lexicographically.

Hypergraph Decomposition Heuristic

An alternative heuristic for generating a low height balanced pseudo tree is based on the recursive decomposition of the dual hypergraph associated with the graphical model.

DEFINITION **33 (dual hypergraph)** *The* dual hypergraph *of a graphical model* $\langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$*, is a pair* $\mathcal{H} = (\mathbf{V}, \mathbf{E})$*, where each function in* $\mathbf{F}$ *is a vertex* $v_i \in \mathbf{V}$ *and each variable in* $\mathbf{X}$ *is an edge* $e_j \in \mathbf{E}$ *connecting all the functions (vertices) in which it appears.*

DEFINITION **34 (hypergraph separators)** *Given a dual hypergraph* $\mathcal{H} = (\mathbf{V}, \mathbf{E})$ *of a graphical model, a* hypergraph separator decomposition *is a triple* $\langle \mathcal{H}, \mathcal{S}, \alpha \rangle$ *where:*

1. $\mathcal{S} \subset \mathbf{E}$*, and the removal of* $\mathcal{S}$ *separates* $\mathcal{H}$ *into* k *disconnected components;*

2. α *is a relation over the size of the disjoint subgraphs (i.e., balance factor).*

It is well known that the problem of finding the minimal size hypergraph separator is hard. However heuristic approaches were developed over the years. A good approach is packaged in `hMeTiS`[3].

We will use this software as a basis for our pseudo tree generation. Following [24], generating a pseudo tree $\mathcal{T}$ for $\mathcal{R}$ using `hMeTiS` is fairly straightforward. The vertices of the hypergraph are partitioned into two balanced (roughly equal-sized) parts, denoted by $\mathcal{H}_{left}$ and $\mathcal{H}_{right}$ respectively, while minimizing the number of hyperedges across. A small number of crossing edges translates into a small number of variables shared between the two sets of functions. $\mathcal{H}_{left}$ and $\mathcal{H}_{right}$ are then each recursively partitioned in the same fashion, until they contain a single vertex. The result of this process is a tree of hypergraph separators which can be shown to also be a pseudo tree of the original model where each separator corresponds to a subset of variables chained together.

Since the hypergraph partitioning heuristic uses a non-deterministic algorithm ($i.e.$, `hMeTiS`), the depth and induced width of the resulting pseudo tree may vary significantly

[3] Available at: http://www-users.cs.umn.edu/karypis/metis/hmetis

from one run to the next. In our experiments we picked the pseudo tree with the smallest depth out of 10 independent runs.

In Table3.3 we illustrate the induced width and depth of the pseudo tree obtained with the hypergraph and min-fill heuristics for 10 belief networks from the Bayesian Networks Repository[4] and 10 constraint networks derived from the SPOT5 benchmark [7]. From this and the experiments presented in the remaining of this section, we observe that the min-fill heuristic generates lower induced width pseudo trees, while the hypergraph heuristic produces much smaller depth pseudo trees. Therefore, perhaps the hypergraph based pseudo trees appear to be favorable for tree search algorithms guided by heuristics that are not sensitive to the treewidth ($e.g.$, local consistency based heuristics), while the min-fill pseudo trees, which minimize the treewidth, are more appropriate for search algorithms whose guiding heuristic is sensitive to the treewidth ($e.g.$, mini-bucket heuristics).

3.6.3 Results for Empirical Evaluation on Bayesian Networks

In this section we focus on mini-bucket heuristics and static variable orderings.

Random Bayesian Networks

The random Bayesian networks were generated using parameters (n, k, c, p), where n is the number of variables, k is the domain size, c is the number of conditional probability tables (CPTs) and p is the number of parents in each CPT. The structure of the network is created by randomly picking c variables out of n and, for each, randomly picking p parents from their preceding variables, relative to some ordering. The remaining $n - c$ variables are called *root* nodes. The entries of each probability table are generated randomly using a uniform distribution, and the table is then normalized.

Table 3.4 shows detailed results for solving a class of random belief networks using min-fill and hypergraph partitioning based pseudo trees. The columns are indexed by the

[4]Available at: http://www.cs.huji.ac.il/labs/compbio/Repository

k	(w*, h)	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=2		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=4		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=6		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=8		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=10		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=12	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
minfill pseudo tree													
		0.43		0.43		0.44		0.43		0.44		0.45	
		174.86	2,109,890	89.33	1,088,420	38.19	488,197	3.28	41,539	0.90	12,918	1.06	15,021
2	(14, 25)	12.23	308,536	1.01	25,706	0.70	17,124	0.17	4,273	0.07	1,666	**0.06**	1,103
		25.86	62,466	3.13	10,737	2.75	10,289	2.71	10,653	2.91	11,570	2.59	10,153
		3.07	11,023	0.50	1,365	0.24	635	0.15	489	0.17	450	0.18	347
		0.43		0.43		0.44		0.47		0.69		2.12	
		-	-	-	-	122.00	1,061,530	37.44	344,128	7.23	67,299	3.55	21,341
3	(14, 25)	-	-	100.47	1,950,280	40.54	722,818	19.78	384,609	2.37	39,318	**2.26**	13,957
		163.72	208,945	31.09	24,603	23.00	19,753	23.50	19,293	28.24	17,787	44.43	18,994
		137.61	357,485	24.93	34,127	16.17	6,283	16.40	1,613	20.85	702	34.96	478
		0.50		0.50		0.52		0.80		3.93		39.22	
		-	-	-	-	251.01	1,724,330	107.49	742,803	**20.31**	137,357	43.14	42,869
4	(14, 25)	-	-	283.61	4,585,420	188.38	2,922,760	85.19	1,326,610	23.38	303,695	41.27	51,276
		-	-	162.86	48,281	157.93	31,620	170.88	28,508	218.89	27,731	323.48	13,235
		-	-	155.49	85,964	146.72	7,891	161.38	1,367	211.84	697	317.11	218
		0.49		0.49		0.58		2.20		33.18			
		-	-	-	-	298.49	1,645,150	174.05	998,579	**116.31**	572,171		
5	(14, 25)	-	-	-	-	267.68	3,804,650	185.49	2,540,320	127.26	1,218,160		
		-	-	277.68	51,702	288.91	42,167	293.88	38,522	-	-		
		-	-	270.10	69,453	282.30	5,623	291.07	1,054	-	-		
hypergraph pseudo tree													
		0.43		0.43		0.44		0.43		0.44		0.45	
		178.94	2,076,390	143.48	1,739,470	121.20	1,495,580	67.72	858,691	24.85	319,742	7.63	99,539
2	(14, 20)	18.87	453,372	2.37	44,796	0.83	9,181	0.73	7,135	0.54	2,415	**0.50**	1,242
		120.80	203,392	8.83	15,798	3.65	9,299	3.47	9,134	3.41	9,013	3.47	9,163
		3.64	11,524	0.85	899	0.63	480	0.58	363	0.60	336	0.66	294
		0.43		0.43		0.44		0.47		0.69		2.12	
		-	-	-	-	-	-	172.16	1,508,000	119.81	1,066,200	81.45	717,941
3	(14, 20)	178.35	3,965,780	137.11	2,558,520	67.95	1,078,460	14.27	198,026	5.10	68,847	**2.94**	13,396
		-	-	67.56	53,725	29.66	24,415	21.68	20,004	29.79	19,347	49.22	17,425
		129.58	490,813	16.66	9,164	10.57	1,409	8.39	640	16.64	469	35.47	349
		0.50		0.50		0.52		0.80		3.93		39.22	
		-	-	-	-	-	-	-	-	243.82	1,685,500	157.19	848,755
4	(14, 20)	-	-	284.29	4,679,600	176.11	2,478,050	89.32	1,196,610	**35.50**	409,701	41.73	30,918
		-	-	167.98	52,789	141.18	32,760	164.00	30,774	213.91	31,316	300.53	13,787
		287.64	666,192	142.71	18,706	125.39	2,834	139.73	785	196.69	502	303.70	195
		0.49		0.49		0.58		2.20		33.18			
		-	-	-	-	-	-	-	-	295.99	1,524,180		
5	(14, 20)	-	-	-	-	257.71	2,955,420	152.83	1,365,200	**102.25**	586,760		
		-	-	287.11	59,292	289.47	40,179	-	-	-	-		
		-	-	254.74	30,200	253.84	1,933	279.00	645	-	-		

Table 3.4: CPU time in seconds and number of nodes explored for solving **random belief networks** with $n = 100$ nodes, $p = 2$ parents per CPT, $c = 90$ CPTs and domain sizes $k \in \{2, 3, 4, 5\}$. Each test case had $e = 10$ variables chosen randomly as evidence. The time limits are 180 seconds for $k \in \{2, 3\}$ and 300 seconds for $k \in \{4, 5\}$, respectively. Pseudo trees generated by min-fill and hypergraph heuristics.

mini-bucket i-bound. For each domain size we generated 20 random instances and in each test case $e = 10$ variables were chosen randomly as evidence.

We observe that `AOBB+SMB(`i`)` is better than `BB+SMB(`i`)` at relatively small i-bounds ($i.e.$, $i \in \{2, 4, 6\}$) when the heuristic is weak. This demonstrates the benefit of AND/OR over classical OR search when the heuristic estimates are relatively weak and the algorithms rely primarily on search rather than on pruning via the heuristic evaluation function. As the i-bound increases ($e.g.$, $i \geq 8$) and the heuristic estimates become strong enough to cut the

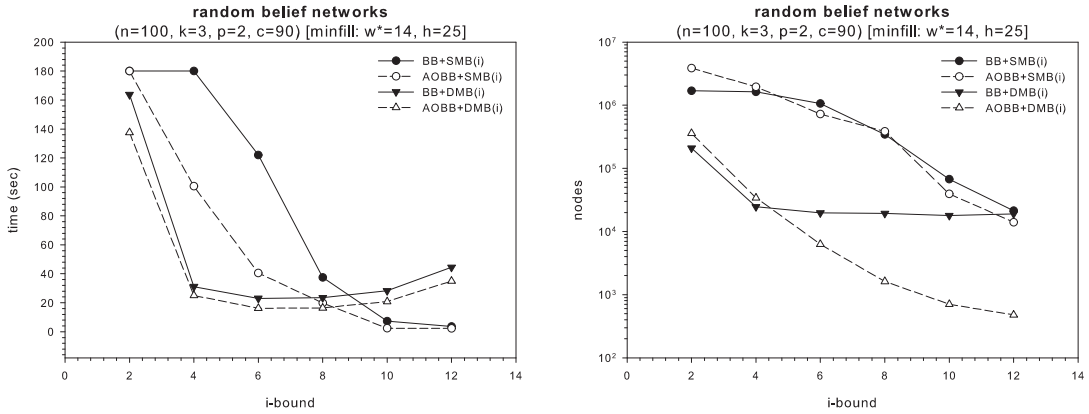


Figure 3.10: Comparison of the impact of static and dynamic mini-bucket heuristics on **random belief networks** with parameters $(n = 100, k = 3, p = 2, c = 90)$ from Table 3.4.

search space substantially, the difference between the AND/OR and OR Branch-and-Bound decreases.

When focusing on dynamic mini-bucket heuristics, we observe that AOBB+DMB(i) is better than BB+DMB(i) at relatively small i-bounds, but the difference is not that prominent as in the static case. This is probably because these heuristics are far more accurate compared with the pre-compiled version and the savings in number of nodes caused by traversing the AND/OR search tree do not translate into additional time savings. When comparing the static and dynamic mini-bucket heuristics, we see that the latter is competitive only for relatively small i-bounds, because of the high overhead of the dynamic mini-bucket. This may be significant because small i-bounds usually require restricted space. At higher levels of the i-bound the accuracy of the dynamic mini-bucket heuristic does not outweigh its overhead.

In some exceptional cases the OR Branch-and-Bound explored fewer nodes than the AND/OR counterpart. For example, on problem class displayed in the third horizontal block of Table 3.4, the search space explored by AOBB+DMB(4) was almost two times larger that that explored by BB+DMB(4). Similarly, AOBB+SMB(8) expanded almost two times more nodes than BB+SMB(8) on this problem class. This can be explained by the internal dynamic ordering used by AND/OR Branch-and-Bound to solve independent

subproblems rooted at the AND nodes in the search tree, which did not pay off in this case. We also see that even though BB+SMB(i) (resp. BB+DMB(i)) traversed a smaller search space than AOBB+SMB(i) (resp. AOBB+DMB(i)), the runtime of the AND/OR algorithms was actually better. This is because the computational overhead of the mini-bucket heuristics was much smaller for AND/OR search than for OR search, and, therefore, the AND/OR algorithms were able to overcome the increase in size of the search space.

Figure 3.10 plots the running time and number of nodes visited by AOBB+SMB(i) and AOBB+DMB(i) (resp. BB+SMB(i) and BB+DMB(i)) as a function of the mini-bucket i-bound for solving the random belief networks with parameters $(n = 100, k = 3, p = 2, c = 90)$ ($i.e.$, corresponding to the second horizontal block from Table 3.4). It shows explicitly how the performance of Branch-and-Bound changes with the mini-bucket strength for both types of heuristics. We see that i-bound of 6 is most cost effective for dynamic mini-buckets, while i-bound of 12 yields best performance for static mini-buckets. We see clearly that the dynamic mini-bucket heuristic is more accurate yielding smaller search spaces. It also demonstrates that the dynamic mini-bucket heuristics are cost effective at small i-bounds, whereas the pre-compiled version is more powerful for larger i-bounds. This behavior is typical for all instances presented in the subsequent sections.

When comparing the min-fill versus hypergraph heuristics for generating pseudo trees, we observe that the hypergraph based pseudo trees have smaller depths. However, min-fill trees appear to be favorable to AOBB+SMB(i). This may be explained by the fact that pre-compiling the mini-bucket heuristic using a min-fill based elimination ordering tends to generate more accurate estimates. For AOBB+DMB(i) the picture is sometimes reversed, but not in a significant way.

Random Coding Networks

The purpose of *channel coding* is to provide reliable communication through a noisy channel. A systematic error-correcting encoding [91] maps a vector of K *information bits*

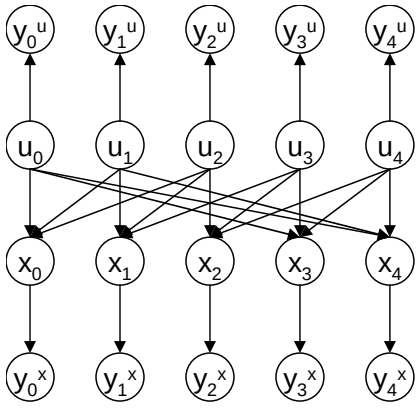


Figure 3.11: Belief network for structured (10,5) block code with parent size $p = 3$.

$u = (u_1, ..., u_k)$, $u_i \in \{0, 1\}$ into an N-bit *codeword* $c = (u, x)$, where $N - K$ additional bits $x = (x_1, ..., x_{N-K})$, $x_j \in \{0, 1\}$ add redundancy to the information source in order to decrease the decoding error. The codeword, called the channel input, is transmitted through a noisy channel. A commonly used Additive White Noise (AWGN) channel model implies that independent Gaussian noise with variance σ^2 is added to each transmitted bit, producing the channel output y. Given a real-valued vector y, the decoding task is to restore the input information vector u [91, 68, 78]. An alternative approach, not considered here, is to round y to a 0/1 vector before decoding.

Our random coding networks fall within the class of linear block codes. They can be represented as four-layer belief networks (Figure 3.11). The second and third layers (from top) correspond to input information bits and parity check bits respectively. Each parity check bit represents an XOR function of input bits u_i. The first and last layers correspond to transmitted information and parity check bits respectively. Input information and parity check nodes are binary, while the output nodes are real-valued. In our experiments, each layer has the same number of nodes because we use code rate of $R = \frac{K}{N} = \frac{1}{2}$, where K is the number of input bits and N is the number of transmitted bits.

Given a number of input bits K, number of parents P for each XOR bit, and channel noise variance σ^2, a coding network structure is generated by randomly picking parents for each XOR node. Then we simulate an input signal by assuming a uniform random

minfill pseudo tree												
(K, N)	(w*, h)	SamIam	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=4		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=8		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=12		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=16		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=20	
			time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
			0.02		0.02		0.07		0.68		8.33	
			-	-	19.71	203,028	0.09	184	0.71	153	8.51	153
(64, 128)	(27, 40)	-	287.10	5,052,010	6.58	119,289	**0.08**	152	0.68	129	8.34	129
$\sigma^2 = 0.22$			23.42	9,932	0.43	232	1.43	153	12.76	153	121.90	153
			23.62	20,008	0.35	185	1.37	129	12.77	129	121.12	129
			0.02		0.02		0.07		0.68		8.32	
			-	-	82.60	850,665	1.16	12,190	**0.81**	1,463	8.35	227
(64, 128)	(27, 40)	-	277.41	5,250,380	47.80	834,680	1.23	22,406	0.84	3,096	8.33	160
$\sigma^2 = 0.36$			48.81	19,489	5.38	1,504	5.71	618	15.70	240	123.76	192
			48.71	44,734	5.17	1,864	5.53	512	15.53	164	122.90	144
hypergraph pseudo tree												
			0.32		0.33		0.38		1.02		8.91	
			-	-	24.29	287,699	0.59	2,259	1.06	156	8.97	156
(64, 128)	(27, 34)	-	-	-	4.76	61,426	**0.40**	381	1.03	142	8.92	129
$\sigma^2 = 0.22$			35.71	20,678	0.77	263	1.71	163	12.02	163	107.08	163
			31.46	17,224	0.59	160	1.60	129	11.69	129	102.38	129
			0.32		0.33		0.38		1.05		9.39	
			-	-	113.04	1,391,480	22.26	275,844	1.74	9,039	9.40	295
(64, 128)	(27, 34)	-	-	-	34.73	489,614	1.82	19,040	**1.69**	9,494	9.40	295
$\sigma^2 = 0.36$			92.76	50,006	3.34	1,134	3.67	408	14.80	307	105.92	185
			54.25	26,031	5.55	1,312	7.91	472	12.52	143	105.76	142

Table 3.5: CPU time and nodes visited for solving **random coding networks** with 64 bits and 4 parents per XOR bit. Time limit 300 seconds. Pseudo trees generated by min-fill and hypergraph heuristics. SAMIAM was not able to solve any of the test instances.

distribution of information bits, compute the corresponding values of the parity check bits, and generate an assignment to the output nodes by adding Gaussian noise to each information and parity check bit. The decoding algorithm takes as input the coding network and the observed real-valued output assignment and recovers the original input bit-vector by computing or approximating an MPE assignment.

Tables 3.5 and 3.6 display the results using min-fill and hypergraph based pseudo trees for solving two classes of random coding networks with $K = 64$ and $K = 128$ input bits, respectively. The number of parents for each XOR bit was $P = 4$ and we chose the channel noise variance $\sigma^2 \in \{0.22, 0.36\}$. We see that `AOBB+SMB(`$i$`)` and `AOBB+DMB(`$i$`)` are slightly faster than `BB+SMB(`i`)` and `BB+DMB(`i`)`, respectively, only for relatively small i-bounds. In several test cases, however, the search space explored by the AND/OR algorithms was larger than the corresponding OR space. For instance, on the problem class with $K = 128$ and $\sigma^2 = 0.36$ shown in the second horizontal block of Table 3.6, `AOBB+SMB(12)` expanded almost 2 times more nodes than `BB+SMB(12)`. This was caused again by the internal dynamic variable ordering used by the AND/OR algorithms.

minfill pseudo tree												
(K, N)	(w*, h)	SamIam	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=4		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=8		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=12		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=16		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=20	
			time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
			0.05		0.06		0.18		1.80		25.65	
					257.42	1,581,950	52.69	345,028	3.53	12,513	25.75	2,065
(128, 256)	(53, 71)	-	-	-	229.02	3,227,110	16.67	206,004	**3.51**	22,644	25.87	3,081
$\sigma^2 = 0.22$			196.64	41,359	48.80	4,178	17.86	726	130.95	588	-	-
			195.82	121,822	48.17	9,391	17.15	500	129.38	388	-	-
			0.05		0.06		0.18		1.80		25.39	
			-	-	-	-	271.29	1,717,770	211.88	1,452,980	**99.14**	598,738
(128, 256)	(53, 71)	-	-	-	291.61	4,309,160	240.74	3,409,580	188.44	2,617,880	110.89	1,137,120
$\sigma^2 = 0.36$			289.06	65,591	230.23	22,617	234.33	6,857	276.40	1,957	-	-
			289.09	223,938	229.91	46,768	233.96	7,947	276.31	953	-	-
hypergraph pseudo tree												
			0.73		0.74		0.86		2.49		27.13	
			-	-	285.82	1,765,300	184.90	1,264,890	94.43	677,488	31.72	36,604
(128, 256)	(53, 63)	-	-	-	238.91	3,070,670	125.01	1,252,930	38.12	404,160	**27.28**	1,658
$\sigma^2 = 0.22$			277.94	133,702	152.10	21,264	27.63	942	90.89	376	-	-
			282.15	126,614	84.82	6,358	73.46	1,307	166.75	409	-	-
			0.73		0.74		0.86		2.51		25.95	
			-	-	-	-	296.69	1,948,930	285.70	2,009,240	210.16	1,360,710
(128, 256)	(53, 63)	-	-	-	-	-	296.02	3,583,930	251.96	2,969,470	**142.85**	1,340,740
$\sigma^2 = 0.36$			-	-	287.30	32,456	269.73	5,269	292.08	2,308	-	-
			-	-	261.00	58,212	269.14	4,614	282.24	823	-	-

Table 3.6: CPU time and nodes visited for solving **random coding networks** with 128 bits and 4 parents per XOR bit. Time limit 300 seconds. Pseudo trees generated by min-fill and hypergraph heuristics. SAMIAM was not able to solve any of the test instances.

We also see that the overhead of the mini-bucket heuristic was smaller in the AND/OR than the OR case, which paid off in some test cases.

When looking at the impact of the min-fill versus the hypergraph based pseudo trees we see that, even though the hypergraph trees were shallower than the min-fill ones, the mini-bucket heuristics generated relative to min-fill orderings were more accurate than those corresponding to hypergraph partitioning based orderings. In some cases this translated into significant time savings. For example, on the problem class with $K = 128$ and $\sigma^2 = 0.22$, the min-fill pseudo tree causes an 8-fold speedup over the hypergraph tree, for `AOBB+SMB(12)`. A similar behavior can be observed for dynamic mini-bucket heuristics.

Figure 3.12 plots the running time and number of nodes visited by `AOBB+SMB(`i`)` and `AOBB+DMB(`i`)` (resp. `BB+SMB(`i`)` and `BB+DMB(`i`)`), for solving the coding networks with parameters $(K = 128, \sigma^2 = 0.22)$ ($i.e.$, corresponding to the first horizontal block from Table 3.6). We see that as the i-bound increases, the mini-bucket heuristics become more accurate and the performance of Branch-and-Bound improves. For example, i-bound of 14 yields the best performance for `AOBB+SMB(`i`)`, whereas `AOBB+DMB(i)` achieves

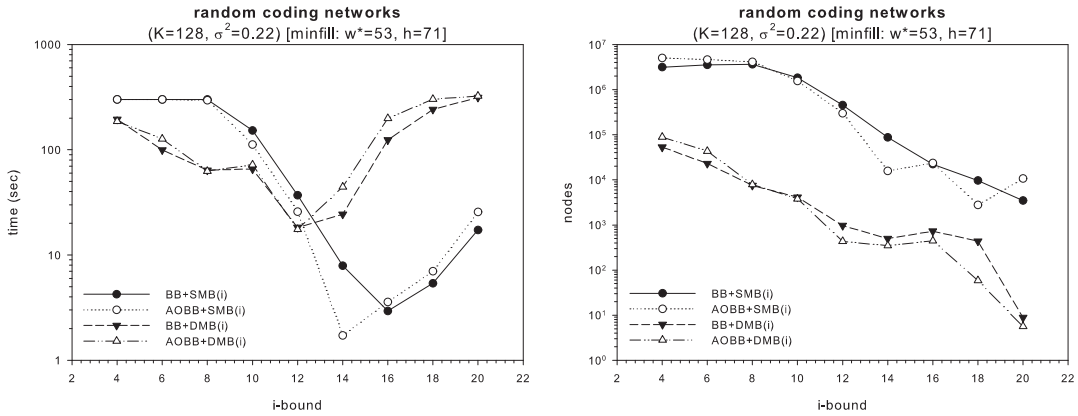


Figure 3.12: Comparison of the impact of static and dynamic mini-bucket heuristics on **random coding networks** with parameters $(K = 128, \sigma^2 = 0.22)$ from Table 3.6.

the best performance at $i = 12$. For even larger i-bounds however, the overhead of both the pre-compiled and dynamic heuristics deteriorates the performance of the algorithms. The dynamic mini-bucket heuristics are better for relatively small i-bounds, whereas relatively larger i-bounds are cost effective for the pre-compiled heuristics.

Grid Networks

In random grid networks, the nodes are arranged in an $N \times N$ square and each CPT is generated uniformly at random. We experimented with problem instances having bi-valued variables that were initially developed in [112] for the task of weighted model counting. For these problems N ranges between 10 and 38, and, for each instance, 90% of the CPTs are deterministic (having only 0 and 1 probability entries).

Table 3.7 displays the results for experiments with 8 grids of increasing difficulty, using mini-fill based pseudo trees. For each test instance we ran a single MPE query with e evidence variables picked randomly. We see again the superiority of `AOBB+SMB(`i`)` over the OR counterpart, especially on the harder instances. For example, on the `90-30-1` grid, `AOBB+SMB(20)` finds the MPE in about 87 seconds, whereas `BB+SMB(20)` exceeds the 1 hour time limit. The AND/OR Branch-and-Bound algorithms with dynamic mini-bucket heuristics as well as `SamIam` are able to solve relatively efficiently only the first 3 test

minfill pseudo tree											
grid (w*, h) (n, e)	SamIam	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=8		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=10		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=12		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=14		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=16	
	time	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.01		0.02		0.04		0.07		0.07	
90-10-1		0.12	3,348	**0.04**	424	0.05	153	0.07	153	0.08	153
(13, 39)	0.13	0.17	8,080	0.06	2,052	0.05	101	0.07	101	0.08	101
(100, 0)		0.87	543	0.57	250	0.48	153	0.54	153	0.54	153
		0.34	344	0.33	241	0.32	101	0.39	101	0.39	101
		0.02		0.04		0.11		0.22		0.72	
90-14-1		75.71	1,235,366	71.98	1,320,090	1.07	18,852	0.54	5,035	0.90	2,826
(22, 66)	11.97	4.27	130,619	3.44	100,696	0.61	17,479	**0.32**	3,321	0.81	2,938
(196, 0)		149.44	16,415	52.34	2,894	12.46	537	13.71	211	19.22	199
		65.74	31,476	33.57	4,137	7.50	397	12.00	211	17.65	199
		0.03		0.05		0.14		0.46		1.01	
90-16-1		-	-	-	-	23.74	347,479	1.85	18,855	**1.44**	6,098
(24, 82)	147.19	362.66	10,104,350	91.03	2,600,690	7.53	193,440	1.89	39,825	1.78	23,421
(256, 0)		771.73	43,366	553.08	13,363	172.14	2,011	166.61	1,169	65.15	414
		1114.19	462,180	410.87	47,121	109.11	3,227	80.57	719	40.68	260

		i=12		i=14		i=16		i=18		i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.33		0.89		2.69		7.61		31.26	
90-24-1		-	-	-	-	-	-	-	-	-	-
(33, 111)	-	-	-	1500.66	24,117,151	921.96	18,238,983	**93.73**	1,413,764	111.46	1,308,009
(576, 20)		-	-	-	-	-	-	-	-	-	-
		-	-	-	-	1367.38	2,739	1979.42	1,228	2637.71	598
		0.37		1.02		3.39		11.74		36.16	
90-26-1		-	-	-	-	324.30	2,234,558	-	-	70.53	327,859
(36, 113)	-	206.93	2,903,489	242.37	3,205,257	**7.43**	59,055	21.48	165,182	36.49	5,777
(676, 40)		-	-	-	-	-	-	-	-	-	-
		-	-	-	-	1514.18	2,545	2889.49	1,191	-	-
		0.53		1.35		4.36		13.34		50.53	
90-30-1		-	-	-	-	-	-	-	-	-	-
(43, 150)	-	742.51	9,445,224	239.08	3,324,942	215.56	3,039,966	101.10	1,358,569	**87.68**	485,300
(900, 60)		-	-	-	-	-	-	-	-	-	-
		-	-	-	-	-	-	-	-	-	-
		0.66		1.60		5.35		18.42		62.17	
90-34-1		-	-	-	-	-	-	-	-	-	-
(45, 153)	-	-	-	-	-	-	-	-	-	**257.14**	1,549,829
(1154, 80)		-	-	-	-	-	-	-	-	-	-
		-	-	-	-	-	-	-	-	-	-
		0.82		2.16		6.43		20.46		72.10	
90-38-1		-	-	-	-	-	-	-	-	-	-
(47, 163)	-	936.65	6,835,745	1858.99	12,321,175	341.05	2,850,393	252.67	2,079,146	**199.44**	1,038,065
(1444, 120)		-	-	-	-	-	-	-	-	-	-
		-	-	-	-	-	-	-	-	-	-

Table 3.7: CPU time in seconds and nodes visited for solving **grid networks**. Time limit 1 hour.

instances.

Figure 3.13 plots the running time and number of nodes visited by `AOBB+SMB(`i`)` and `AOBB+DMB(`i`)` (resp. `BB+SMB(`i`)` and `BB+DMB(`i`)`), for solving the `90-14-1` grid network ($i.e.$, corresponding to the second horizontal block from Table 3.7). Focusing on `AOBB+SMB(`i`)` (resp. `BB+SMB(`i`)`) we see that its running time, as a function of i, forms a U-shaped curve. At first ($i = 4$) it is high, then as the i-bound increases the total time decreases (when $i = 10$ the time is 3.44 for `AOBB+SMB(10)` and 71.98 for `BB+SMB(10)`, respectively), but then as i increases further the time starts to increase again. The same behavior can be observed in the case of `AOBB+DMB(`i`)` (resp. `BB+DMB(`i`)`) as well.

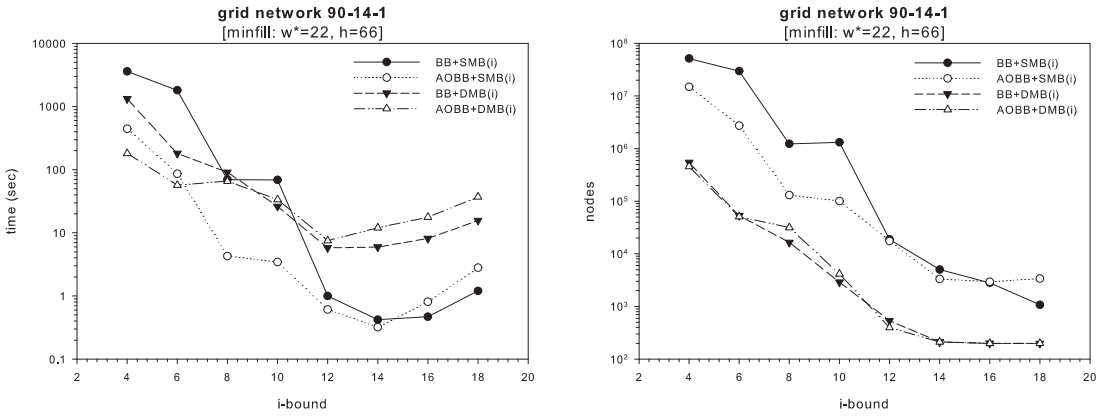


Figure 3.13: Comparison of the impact of static and dynamic mini-bucket heuristics on the **`90-14-1` grid network** from Table 3.7.

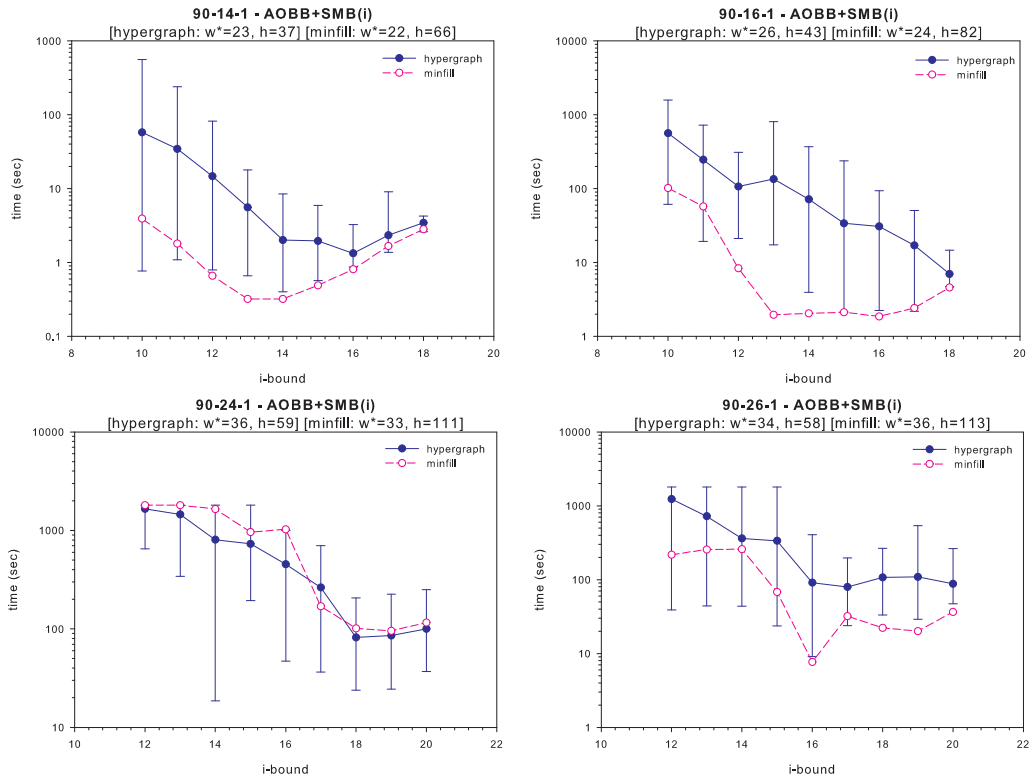


Figure 3.14: Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. CPU time in seconds for solving **grid networks** with `AOBB+SMB(`i`)`.

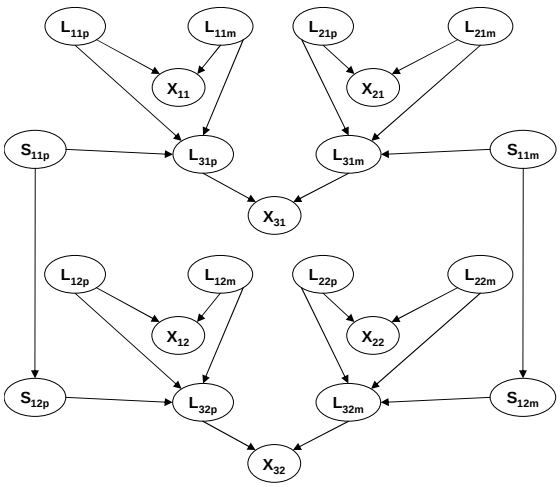


Figure 3.15: A fragment of a belief network used in genetic linkage analysis.

Figure 3.14 displays the runtime distribution of AOBB+SMB(i) using hypergraph based pseudo trees for 4 grid networks. For each reported i-bound, the corresponding data point and error bar reports the average as well as the minimum and maximum runtime obtained over 20 independent runs of the algorithm with a 30 minute time limit. We also record the average induced width and depth obtained for the hypergraph pseudo trees (see the header of each plot in Figure 3.14). As observed earlier, the hypergraph based pseudo trees are significantly shallower compared with the min-fill ones, and in some cases they are able to improve performance dramatically, especially at relatively small i-bounds. For example, on the grid 90-24-1, AOBB+SMB(14) guided by a hypergraph pseudo tree is about 2 orders of magnitude faster than AOBB+SMB(14) using a min-fill pseudo tree. At larger i-bounds, the pre-compiled mini-bucket heuristic benefits from the small induced width which normally is obtained with the min-fill ordering. Therefore AOBB+SMB(i) using min-fill based trees is generally faster than AOBB+SMB(i) guided by hypergraph based trees ($e.g.$, 90-26-1).

Genetic Linkage Analysis

In human genetic linkage analysis [98], the *haplotype* is the sequence of alleles at different loci inherited by an individual from one parent, and the two haplotypes (maternal and pater-

min-fill pseudo tree											
pedigree (n, k) (w*, h)	Superlink SamIam	MBE(i) BB+SMB(i) AOBB+SMB(i) i=6		MBE(i) BB+SMB(i) AOBB+SMB(i) i=8		MBE(i) BB+SMB(i) AOBB+SMB(i) i=10		MBE(i) BB+SMB(i) AOBB+SMB(i) i=12		MBE(i) BB+SMB(i) AOBB+SMB(i) i=14	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
ped1		0.05		0.05		0.11		0.31		0.97	
(299, 5)	54.73	-	-	-	-	6.34	37,657	7.33	42,447	8.30	41,134
(15, 61)	5.44	24.30	416,326	13.17	206,439	**1.58**	24,361	1.84	25,674	1.89	15,156
ped38		0.12		0.45		2.20		60.97		out	
(582, 5)	**28.36**	-	-	-	-	-	-	-	-		
(17, 59)	out	-	-	8120.58	85,367,022	-	-	3040.60	35,394,461		
ped50		0.11		0.74		5.38		37.19		out	
(479, 5)	-	-	-	-	-	-	-	-	-		
(18, 58)	out	-	-	-	-	476.77	5,566,578	**104.00**	748,792		
		i=10		i=12		i=14		i=16		i=18	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
ped23		0.42		2.33		11.33		274.75		out	
(310, 5)	9146.19	-	-	-	-	3176.72	14,044,797	343.52	358,604		
(27, 71)	out	498.05	6,623,197	**15.45**	154,676	16.28	67,456	286.11	117,308		
ped37		0.67		5.16		21.53		58.59		out	
(1032, 5)	**64.17**	-	-	-	-	-	-	-	-		
(21, 61)	out	273.39	3,191,218	1682.09	25,729,009	1096.79	15,598,863	128.16	953,061		
		i=12		i=14		i=16		i=18		i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
ped18		0.51		1.42		4.59		12.87		19.30	
(1184, 5)	139.06	-	-	-	-	-	-	-	-	-	-
(21, 119)	157.05	-	-	2177.81	28,651,103	270.96	2,555,078	100.61	682,175	**20.27**	7,689
ped20		1.42		5.11		37.53		410.96		out	
(388, 5)	**14.72**	-	-	-	-	-	-	-	-		
(24, 66)	out	3793.31	54,941,659	1293.76	18,449,393	1259.05	17,810,674	1080.05	9,151,195		
ped25		0.34		0.72		2.27		6.56		29.30	
(994, 5)	-	-	-	-	-	-	-	-	-	-	-
(34, 89)	out	-	-	-	-	9399.28	111,301,168	3607.82	34,306,937	**2965.60**	28,326,541
ped30		0.42		0.83		1.78		5.75		21.30	
(1016, 5)	13095.83	-	-	-	-	-	-	-	-	-	-
(23, 118)	out	-	-	-	-	-	-	214.10	1,379,131	**91.92**	685,661
ped33		0.58		2.31		7.84		33.44		112.83	
(581, 4)	-	-	-	-	-	-	-	-	-	-	-
(37, 165)	out	2804.61	34,229,495	737.96	9,114,411	3896.98	50,072,988	**159.50**	1,647,488	2956.47	35,903,215
ped39		0.52		2.32		8.41		33.15		81.27	
(1272, 5)	322.14	-	-	-	-	-	-	-	-	-	-
(23, 94)	out	-	-	-	-	4041.56	52,804,044	386.13	2,171,470	**141.23**	407,280
ped42		4.20		31.33		206.40		out		out	
(448, 5)	**561.31**	-	-	-	-	-	-				
(25, 76)	out	-	-	-	-	-	-				

Table 3.8: CPU time and nodes visited for solving **genetic linkage networks**. Time limit 3 hours.

nal) of an individual constitute this individual's *genotype*. When genotypes are measured by standard procedures, the result is a list of unordered pairs of alleles, one pair for each locus. The *maximum likelihood haplotype* problem consists of finding a joint haplotype configuration for all members of the pedigree which maximizes the probability of data.

The pedigree data can be represented as a belief network with three types of random variables: *genetic loci* variables which represent the genotypes of the individuals in the pedigree (two genetic loci variables per individual per locus, one for the paternal allele and one for the maternal allele), *phenotype* variables, and *selector* variables which are auxiliary variables used to represent the gene flow in the pedigree. Figure 3.15 shows a fragment of a network that describes parents-child interactions in a simple 2-loci analysis. The ge-

netic loci variables of individual i at locus j are denoted by $L_{i,jp}$ and $L_{i,jm}$. Variables $X_{i,j}$, $S_{i,jp}$ and $S_{i,jm}$ denote the phenotype variable, the paternal selector variable and the maternal selector variable of individual i at locus j, respectively. The conditional probability tables that correspond to the selector variables are parameterized by the *recombination ratio* θ [47]. The remaining tables contain only deterministic information. It can be shown that given the pedigree data, the haplotyping problem is equivalent to computing the Most Probable Explanation (MPE) of the corresponding belief network [47, 46].

Table 3.8 shows results with 12 genetic linkage networks[5]. For comparison, we include results obtained with SUPERLINK 1.6. SUPERLINK [47, 46] which is currently one of the most efficient solvers for genetic linkage analysis, uses a combination of variable elimination and conditioning, and takes advantage of the determinism in the network. We did not run AOBB+DMB(i) on this domain because of its prohibitively high computational overhead associated with relatively large i-bounds.

We observe again that AOBB+SMB(i) is the best performing algorithm, outperforming its competitors on 8 out of the 12 test networks. For example, on the ped23 instance, AOBB+SMB(16) is 3 orders of magnitude faster than SUPERLINK, whereas SAMIAM and BB+SMB(i) exceed the 2GB memory bound and the 3 hour time limit, respectively. Similarly, on the ped30 instance, AOBB+SMB(20) outperforms SUPERLINK with about 2 orders of magnitude, while neither SAMIAM nor BB+SMB(16) are able to solve the problem instance. Notice also that the ped42 instance is solved only by SUPERLINK.

Figure 3.16 displays the runtime distribution of AOBB+SMB(i) with hypergraph based pseudo trees over 20 independent runs, for 4 linkage instances. Again, we see that the hypergraph partitioning heuristic generates pseudo trees having average depths almost two times smaller than those of the min-fill based ones. Therefore, using hypergraph based pseudo trees improves sometimes significantly the performance for relatively small i-bounds ($e.g.$, ped23, ped33).

[5] Available at http://bioinfo.cs.technion.ac.il/superlink/. The corresponding belief network of the pedigree data was extracted using the export feature of the SUPERLINK 1.6 program.

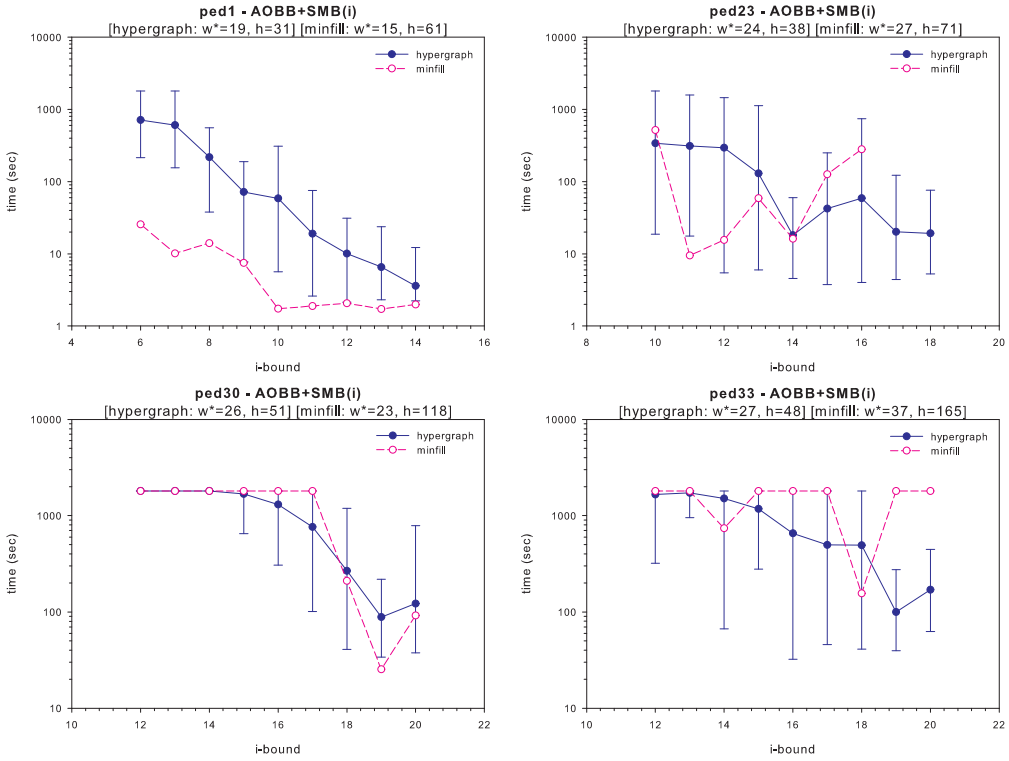


Figure 3.16: Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. CPU time in seconds for solving **linkage networks** with AOBB+SMB(i).

ISCAS'89 Circuits (BN)

ISCAS'89 circuits[6] are a common benchmark used in formal verification and diagnosis. For our purpose, we converted each of these circuits into a belief network by removing flip-flops and buffers in a standard way, creating a deterministic conditional probabilistic tables for each gate and putting uniform distributions on the input signals.

Table 3.9 shows the results for experiments with 10 circuits, using min-fill based pseudo trees. As usual, for each test instance we generated a single MPE query without any evidence. When comparing the algorithms using static mini-bucket heuristics we observe again the superiority of the AND/OR over OR Branch-and-Bound search in almost all test cases, across i-bounds. For instance, on the c880 circuit, AOBB+SMB(4) proves optimality in less than a second, while BB+SMB(4) exceeds the 30 minute time limit. Similarly,

[6] Available at http://www.fm.vslib.cz/kes/asic/iscas/

minfill pseudo tree											
iscas89 (w*, h) (n, d)	SamIam	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=6		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=8		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=10		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=12		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=14	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.04		0.05		0.08		0.12		0.26	
c432		-	-	-	-	-	-	0.29	432	0.42	432
(27, 45)	out	-	-	-	-	605.79	20,751,699	**0.13**	432	0.28	432
(432, 2)		-	-	132.19	21,215	2.23	432	3.44	432	5.85	432
		1422.98	4,438,597	24.03	39,711	1.15	432	2.23	432	4.52	432
		0.02		0.03		0.05		0.14		0.37	
c499		0.16	499	0.17	499	0.19	499	0.28	499	0.50	499
(23, 55)	139.89	**0.04**	499	0.05	499	0.06	499	0.15	499	0.38	499
(499, 2)		1.09	499	1.32	499	2.00	499	4.01	499	8.92	499
		0.39	499	0.63	499	1.31	499	3.32	499	8.21	499
		0.09		0.09		0.11		0.18		0.51	
c880		-	-	0.59	881	0.60	881	0.66	881	0.99	881
(27, 67)	out	**0.13**	884	**0.13**	881	0.15	881	0.21	881	0.55	881
(880, 2)		4.49	881	5.82	881	8.07	881	12.78	881	20.99	881
		0.78	881	1.14	881	2.16	881	4.98	881	13.19	881
		0.01		0.02		0.03		0.08		0.20	
s386		0.10	1,358	0.06	677	0.05	172	0.10	172	0.22	172
(19, 44)	3.66	**0.02**	257	**0.02**	257	0.03	172	0.08	172	0.21	172
(172, 2)		0.15	172	0.21	172	0.42	172	0.78	172	1.56	172
		0.09	172	0.16	172	0.36	172	0.72	172	1.50	172
		0.06		0.07		0.12		0.31		1.01	
s953		-	-	-	-	-	-	-	-	601.69	4,031,967
(66, 101)	out	715.60	9,919,295	15.25	238,780	37.11	549,181	22.83	434,481	**2.30**	21,499
(440, 2)		27.12	2,737	18.84	912	64.12	1,009	25.28	467	221.17	577
		26.48	2,738	18.30	913	63.44	1,010	24.75	468	220.97	578
		0.07		0.10		0.16		0.39		1.30	
s1196		-	-	-	-	-	-	-	-	-	-
(54, 97)	out	21.75	316,875	215.81	3,682,077	4.57	77,205	19.81	320,205	16.64	289,873
(560, 2)		2.57	580	4.34	568	49.30	924	126.85	863	582.66	1,008
		1.20	660	2.59	568	45.90	924	118.16	863	571.79	1,008
		0.07		0.09		0.17		0.42		1.26	
s1238		-	-	-	-	272.63	2,078,885	144.85	1,094,713	585.48	4,305,175
(59, 94)	out	2.63	57,355	8.32	187,499	2.14	47,340	**1.49**	25,538	2.12	20,689
(540, 2)		32.17	5,841	6.59	601	370.26	17,278	52.28	651	120.20	558
		2.04	1,089	4.02	795	17.44	1,824	40.35	849	95.84	744
		0.06		0.06		0.09		0.13		0.35	
s1423		-	-	-	-	-	-	0.46	762	0.67	749
(24, 54)	107.48	**0.14**	1,986	0.30	5,171	0.32	5,078	0.17	866	0.37	749
(748, 2)		2.95	751	3.37	749	4.05	749	5.50	749	9.62	749
		0.55	751	0.76	749	1.35	749	2.81	749	6.93	749
		0.08		0.10		0.18		0.46		1.50	
s1488		11.91	92,764	1.65	12,080	2.19	17,410	1.26	6,480	2.17	5,327
(47, 67)	out	11.83	135,563	1.48	17,170	2.29	28,420	1.25	12,285	2.26	12,370
(667, 2)		2.31	670	3.14	670	5.43	668	13.11	667	41.43	667
		0.83	670	1.64	670	3.92	668	11.67	667	40.17	667
		0.07		0.09		0.17		0.49		1.57	
s1494		8.64	64,629	524.05	3,410,547	130.92	815,326	**4.43**	33,373	43.54	268,421
(48, 69)	out	9.63	158,070	28.14	476,874	7.09	118,372	11.87	198,912	2.75	21,137
(661, 2)		6.29	873	6.23	711	9.81	681	26.60	680	93.29	686
		4.88	873	4.77	711	8.36	681	25.10	680	91.70	686

Table 3.9: CPU time and nodes visited for solving belief networks derived from **ISCAS'89 circuits**. Time limit 30 minutes.

on the `s953` circuit, `AOBB+SMB(14)` is 300 times faster than `BB+SMB(14)` and explores a search space 180 times smaller. Using the dynamic mini-bucket heuristics does pay off in some test cases. For example, on the `s1196` circuit, `AOBB+DMB(4)` causes a speedup of 2 over `BB+DMB(4)` and 45 over `AOBB+SMB(4)`, while `BB+SMB(4)` exceeds the time limit. The overall impact of the AND/OR algorithms versus the OR ones can be explained by the relatively shallow pseudo-trees. In summary, the dynamic mini-bucket heuristics were inferior to the corresponding static ones for large i-bounds, however, smaller i-bound

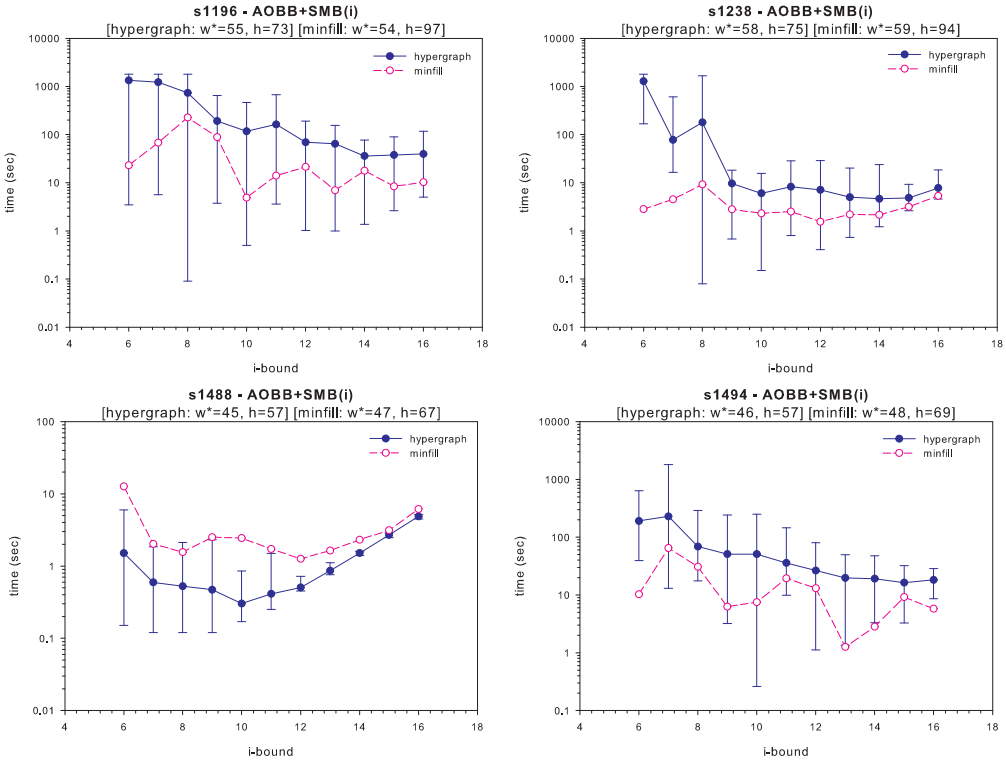


Figure 3.17: Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. CPU time in seconds for solving **ISCAS'89 networks** with AOBB+SMB(i).

dynamic mini-buckets were overall more cost-effective. Notice that SAMIAM is able to solve only 2 out of 10 test instances.

Figure 3.17 shows the runtime distribution of AOBB+SMB(i) with hypergraph pseudo trees, over 20 independent runs. We observe again that in several cases (*e.g.*, s1196, s1238) the hypergraph pseudo trees are able to improve performance with up to 3 orders of magnitude, at relatively small i-bounds.

UAI'06 Evaluation Dataset

The UAI 2006 Evaluation Dataset[7] contains a collection of random as well as real-world belief networks that were used during the first UAI 2006 Inference Evaluation contest.

Table 3.10 shows the results for experiments with 14 networks, using min-fill based pseudo trees. Instances BN_31 through BN_41 are random grid networks with determin-

[7] http://ssli.ee.washington.edu/bilmes/uai06InferenceEvaluation

min-fill pseudo tree											
bn (w*, h) (n, d)	SamIam	MBE(i) BB+SMB(i) AOBB+SMB(i) i=17		MBE(i) BB+SMB(i) AOBB+SMB(i) i=18		MBE(i) BB+SMB(i) AOBB+SMB(i) i=19		MBE(i) BB+SMB(i) AOBB+SMB(i) i=20		MBE(i) BB+SMB(i) AOBB+SMB(i) i=21	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
BN_31		10.31		20.06		34.15		74.17		121.5	
(46, 160)	out	-	-	-	-	-	-	-	-	-	-
(1156, 2)		828.60	4,741,037	1229.64	7,895,304	594.36	3,988,933	646.67	4,293,760	**178.87**	380,470
BN_33		13.84		26.27		48.61		90.35		159.67	
(43, 163)	-	-	-	-	-	-	-	-	-	-	-
(1444, 2)		865.01	3,540,778	**193.79**	685,246	395.99	1,441,245	308.14	1,018,353	230.53	360,880
BN_35		14.27		24.62		47.50		77.66		124.17	
(41, 168)	-	-	-	-	-	-	-	-	-	-	-
(1444, 2)		335.43	1,755,561	390.04	1,954,720	247.73	1,108,708	**191.03**	663,784	234.97	622,551
BN_37		13.82		26.58		44.21		85.17		170.20	
(45, 159)	-	-	-	-	-	-	-	-	-	-	-
(1444, 2)		94.27	428,643	82.15	298,477	**79.99**	183,016	100.41	89,948	196.06	168,957
BN_39		12.95		26.10		51.51		87.16		148.40	
(48, 164)	-	-	-	-	-	-	-	-	-	-	-
(1444, 2)		-	-	-	-	-	-	-	-	**837.58**	3,366,427
BN_41		13.41		23.51		42.01		71.77		125.97	
(49, 164)	-	-	-	-	-	-	-	-	-	-	-
(1444, 2)		125.27	486,844	107.81	364,363	**79.23**	168,340	115.18	195,506	161.10	162,274
BN_126		6.76		13.75		24.62		49.11		98.43	
(54, 70)	-	336.88	2,101,962	871.17	6,677,492	628.26	3,717,027	97.21	350,841	105.54	71,919
(512, 2)		351.91	4,459,174	918.04	10,991,861	126.49	1,333,266	**75.40**	386,490	108.20	150,391
BN_127		7.15		14.26		30.82		56.12		98.82	
(57, 74)	out	-	-	-	-	-	-	-	-	**180.57**	639,878
(512, 2)		-	-	-	-	-	-	-	-	200.14	1,384,957
BN_128		7.77		15.38		28.49		58.08		99.85	
(48, 73)	out	8.19	3,476	15.66	2,645	34.14	36,025	58.54	831	100.29	4,857
(512, 2)		**8.11**	5,587	15.48	1,712	29.64	18,734	58.12	625	100.18	5,823
BN_129		7.39		11.83		24.96		55.28		96.60	
(52, 68)	out					**188.49**	1,605,045	1423.49	11,860,050	343.68	2,049,880
(512, 2)		827.37	11,469,012	-	-	198.24	1,999,591	1796.81	22,855,693	297.90	2,542,057
BN_130		6.29		13.24		22.63		53.68		94.78	
(54, 67)	out	**25.42**	184,439	-	-	918.48	7,317,237	-	-	105.43	110,193
(512, 2)		29.52	348,660	-	-	981.08	10,905,151	-	-	108.25	205,010
BN_131		7.16		13.72		23.36		44.94		82.36	
(48, 72)	out	**21.55**	142,487	47.11	328,560	1216.80	10,249,055	73.25	235,433	-	-
(512, 2)		26.44	296,576	58.78	677,149	1695.44	24,678,072	87.01	673,358	-	-
BN_132		6.16		11.63		22.31		52.78		91.20	
(49, 71)	out	-	-	-	-	-	-	792.42	6,596,296	**644.01**	4,829,396
(512, 2)		-	-	-	-	-	-	886.31	10,251,600	809.86	10,207,347
BN_133		7.60		14.43		27.55		56.54		106.24	
(54, 71)	out	-	-	**24.18**	105,920	46.69	174,274	157.04	932,745	110.05	32,041
(512, 2)		-	-	25.55	169,574	48.53	272,258	184.94	1,859,117	110.87	71,195

Table 3.10: CPU time and nodes visited for solving **UAI'06 instances**. Time limit 30 minutes.

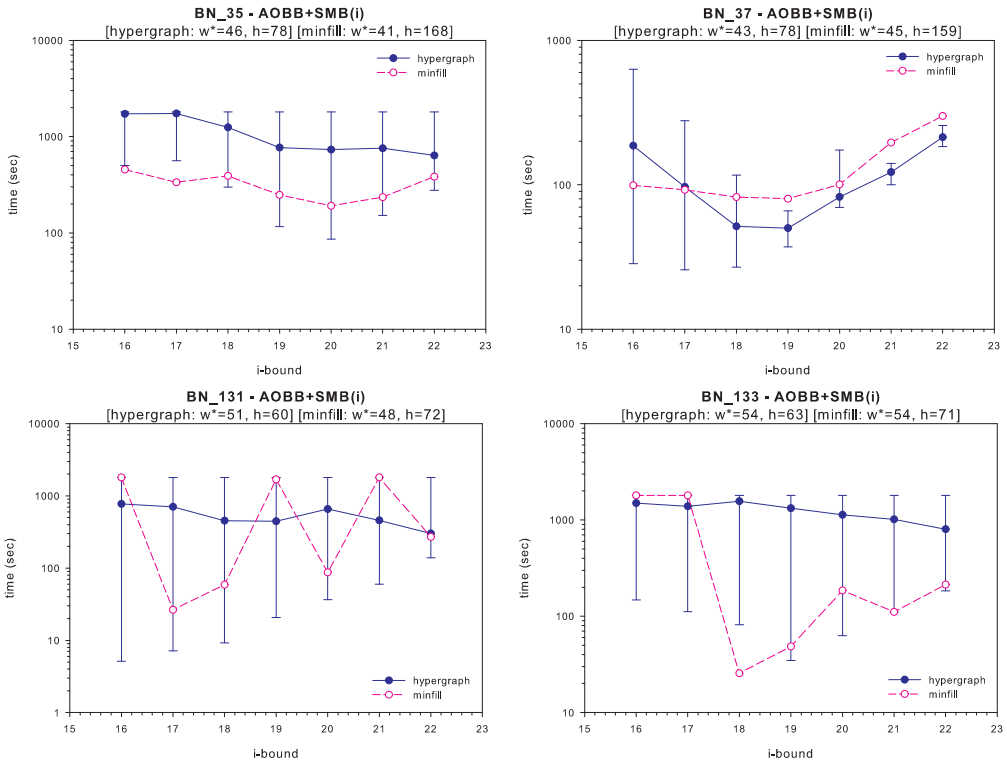


Figure 3.18: Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. CPU time in seconds for solving **UAI'06 networks** with AOBB+SMB(i).

istic CPTs, while instances BN_126 through BN_133 represent random coding networks with 128 input bits, 4 parents per XOR bit and channel noise variance $\sigma^2 = 0.40$. We report only on the Branch-and-Bound algorithms using static mini-buckets. The dynamic mini-buckets were not competitive due to their much higher computational overhead at relatively large i-bounds. We notice again that AOBB+SMB(i) clearly outperforms BB+SMB(i) at all reported i-bounds, especially on the first set of grid networks ($e.g.$, BN_31, ..., BN_41). For instance, on the BN_37, AOBB+SMB(19) finds the MPE solution in about 80 seconds, whereas its OR counterpart BB+SMB(19) exceeds the 30 minute time limit. This is in contrast to what we observe on the second set of coding networks ($e.g.$, BN_126, ..., BN_133), where the best performance is offered by the OR algorithm BB+SMB(i).

Figure 3.18 shows the runtime distribution of AOBB+SMB(i) with hypergraph pseudo trees, over 20 independent runs. We observe again that the hypergraph pseudo trees improve slightly the performance compared with min-fill ones.

Bayesian Network Repository

min-fill pseudo tree											
bn (w*, h) (n, d)	SamIam	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=2		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=3		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=4		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=5		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=6	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.01		0.01		0.01		0.01		0.01	
cpcs54		10.41	141,260	18.26	252,886	0.54	8,072	2.18	30,912	0.62	9,237
(14, 23)	0.16	0.66	16,030	0.34	8,621	0.27	6,761	0.39	10,485	**0.13**	3,672
(54, 2)		1.99	2,493	1.45	2,214	0.70	1,003	0.53	848	0.34	532
		1.48	2,339	1.16	1,889	0.86	798	0.42	419	0.29	159
		0.09		0.09		0.09		0.09		0.09	
cpcs360b		72.21	336,720	66.86	317,249	65.05	316,991	61.38	297,313	63.82	314,173
(20, 27)	18.91	0.45	10,027	0.44	9,827	0.44	9,809	**0.40**	8,947	0.43	9,771
(360, 2)		377.73	308,339	373.48	307,084	373.23	307,083	373.96	307,083	373.34	307,078
		4.36	9,383	4.15	9,309	4.06	9,313	4.20	9,285	4.18	9,181
		1.58		1.58		1.58		1.58		1.58	
cpcs422b		57.43	204,209	56.60	203,448	55.61	203,410	54.27	203,410	54.34	203,409
(23, 36)	112.78	1.80	3,557	1.78	3,409	**1.77**	3,409	**1.77**	3,409	1.78	3,568
(422, 2)		-	-	-	-	-	-	-	-	-	-
		54.48	3,140	54.41	3,142	54.98	3,094	54.98	3,029	55.03	2,998
		0.01		0.01		0.01		0.01		0.01	
Insurance		0.14	1,877	0.06	962	69.56	1,749,933	35.70	910,498	0.02	160
(7, 14)	0.08	0.04	977	0.02	453	0.02	411	**0.01**	255	**0.01**	62
(27, 5)		0.13	364	0.03	89	0.03	87	0.08	87	0.16	87
		0.11	299	0.02	36	0.03	33	0.08	33	0.15	33
		0.02		0.02		0.03		0.06		0.19	
Munin1		-	-	-	-	-	-	-	-	10.16	81,982
(12, 28)	out	6.32	102,540	2.79	44,071	**1.32**	22,934	2.00	42,484	1.79	38,669
(189, 21)		-	-	256.48	80,411	228.91	66,583	62.08	15,523	65.29	15,513
		45.76	84,788	25.46	27,217	18.15	11,230	9.45	2,557	12.30	2,547
		0.14		0.16		0.20		0.32		0.46	
Munin2		-	-	-	-	-	-	-	-	-	-
(9, 32)	4.30	-	-	-	-	137.72	712,814	30.53	174,333	**2.57**	15,978
(1003, 21)		-	-	-	-	-	-	-	-	-	-
		-	-	-	-	-	-	208.47	13,459	167.27	9,360
		0.15		0.15		0.18		0.28		0.40	
Munin3		-	-	-	-	-	-	-	-	-	-
(9, 32)	7.28	-	-	-	-	15.20	152,191	1.02	6,440	**0.63**	1,945
(1044, 21)		-	-	-	-	-	-	-	-	-	-
		-	-	345.26	146,866	28.54	2,573	12.11	1,319	10.50	1,180
		0.16		0.15		0.19		0.32		0.86	
Munin4		-	-	-	-	-	-	-	-	-	-
(9, 35)	26.19	-	-	-	-	-	-	-	-	**292.30**	3,183,146
(1041, 21)		-	-	-	-	-	-	-	-	-	-
		-	-	-	-	-	-	-	-	-	-
		0.03		0.04		0.04		0.04		0.05	
Pigs		-	-	0.50	6,060,855	0.48	6,446,055	0.48	5,956,733	0.48	81,982
(11, 26)	1.14	-	-	**0.06**	455	**0.06**	455	**0.06**	455	0.07	455
(441, 3)				7.98	1,984	8.58	1,984	8.66	1,984	8.79	1,984
				0.31	455	0.39	455	0.49	455	0.63	455
		0.01		0.01		0.01		0.02		0.03	
Water		78.53	1,658,313	78.02	1,670,307	3.47	53,784	0.34	5,202	0.45	6,769
(10, 15)	3.03	0.67	17,210	1.07	24,527	0.80	19,193	**0.14**	3,005	**0.14**	2,658
(32, 4)		344.89	697,777	4.39	1,932	0.92	535	0.67	235	0.98	468
		8.49	11,125	3.97	1,622	0.82	193	0.61	153	0.88	113

Table 3.11: CPU time in seconds and number of nodes visited for solving **Bayesian Network Repository** instances. Time limit 10 minutes.

The Bayesian Network Repository[8] contains a collection of belief networks extracted from various real-life domains which are often used for benchmarking probabilistic inference algorithms.

Table 3.11 displays the results for experiments with 15 belief networks from the repos-

[8]http://www.cs.huji.ac.il/compbio/Repository/

itory. We set the time limit to 10 minutes and for each test instance we generated a single MPE query without evidence. We observe again a considerable improvement of the new AND/OR Branch-and-Bound algorithms over the corresponding OR ones. For example, on the `cpcs360b` network, `AOBB+SMB(5)` causes a CPU speedup of 153 over `BB+SMB(5)`, while exploring a search space 33 times smaller. Similarly, `AOBB+DMB(5)` is 89 times faster than `BB+DMB(5)` and expands about 33 times less nodes. Overall, `AOBB+SMB(`i`)` is the best performing algorithm for this domain. In particular, for networks with relatively low connectivity and large domain sizes (e.g., `Munin` networks) the difference between `AOBB+SMB(`i`)` and `BB+SMB(`i`)` is up to several orders of magnitude in terms of both running time and size of the search space explored.

3.6.4 The Impact of Determinism in Bayesian Networks

In general, when the functions of the graphical model express both hard constraints and general cost functions, it is beneficial to exploit the computational power of the constraints explicitly via constraint propagation [37, 69, 1, 35]. For Bayesian networks, the hard constraints are represented by the zero probability tuples of the CPTs. We note that the use of constraint propagation via directional resolution [108] or generalized arc consistency has been explored in [37, 69], in the context of variable elimination algorithms where the constraints are also extracted based on the zero probabilities in the Bayesian network. The approach we take for handling the determinism in belief networks is based on the known technique of *unit resolution* for Boolean Satisfiability (SAT). The idea of using unit resolution during search for Bayesian networks was first explored in [1].

The CNF formula encodes the determinism in the network and is created based on the zero CPT entries, as follows.

SAT Variables

Given a belief network $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, the CNF is defined over the multi-valued variables $\{X_1, ..., X_n\}$. Its propositions are L_{X_i,x_i}, where $x_i \in D_i$. The proposition is true if X_i is assigned value $x_i \in D_i$ and is false otherwise.

SAT Clauses

The CNF is augmented with a collection of 2-CNFs for each variable X_i in the network, called *at-most-one* clauses, that forbids the assignments of more than one value to a variable. Formally,

DEFINITION **35 (at-most-one clause)** *Given $X_i \in \mathbf{X}$ with domain $D_i = \{x_{i_1}, ..., x_{i_d}\}$, its corresponding* at-most-one *clauses have the following form: $\neg L_{X_i,x_{i_p}} \vee \neg L_{X_i,x_{i_q}}$ for every pair $(x_{i_p}, x_{i_q}) \in D_i \times D_i$, where $1 \leq p < q \leq d$.*

In addition, we will add to the CNF a set of *at-least-one* clauses to ensure that each variable in the network is assigned at least one value from its domain:

DEFINITION **36 (at-least-one clause)** *Given $X_i \in \mathbf{X}$ with domain $D_i \in \{x_{i_1}, ..., x_{i_d}\}$, its corresponding* at-least-one *clause is of the following form: $L_{X_i,x_{i_1}} \vee L_{X_i,x_{i_2}} ... \vee L_{X_i,x_{i_d}}$.*

The remaining clauses are generated from the zero probability tuples in the CPTs.

DEFINITION **37 (no-good clauses)** *Given a conditional probability table $P(X_i|pa(X_i))$, each entry in the CPT having $P(x_i|x_{pa_i}) = 0$, where $pa(X_i) = \{Y_1, ..., Y_t\}$ are X_i's parents and $x_{pa_i} = (y_1, ..., y_t)$ is their corresponding value assignment, can be translated to a* no-good *clause of the form: $\neg L_{Y_1,y_1} \vee ... \vee \neg L_{Y_t,y_t} \vee \neg L_{X_i,x_i}$.*

Example 15 *Consider a belief network over variables $\{A, B, C\}$ with domains $D_A = \{1, 2\}$, $D_B = \{1, 2\}$ and $D_C = \{1, 2, 3\}$, and probability tables: $P(A)$, $P(B)$ and*

A	B	C	$P(C\|A,B)$	Clauses
1	1	1	1	
1	1	2	0	$(\neg L_{A,1} \vee \neg L_{B,1} \vee \neg L_{C,2})$
1	1	3	0	$(\neg L_{A,1} \vee \neg L_{B,1} \vee \neg L_{C,3})$
1	2	1	0	$(\neg L_{A,1} \vee \neg L_{B,2} \vee \neg L_{C,1})$
1	2	2	1	
1	2	3	0	$(\neg L_{A,1} \vee \neg L_{B,2} \vee \neg L_{C,3})$
2	1	1	.2	
2	1	2	.8	
2	1	3	0	$(\neg L_{A,2} \vee \neg L_{B,1} \vee \neg L_{C,3})$
2	2	1	.7	
2	2	2	.3	
2	2	3	0	$(\neg L_{A,2} \vee \neg L_{B,2} \vee \neg L_{C,3})$

Table 3.12: Deterministic CPT $P(C|A,B)$

$P(C|A,B)$*, respectively. The deterministic CPT* $P(C|A,B)$ *is given in Table 3.12. The corresponding CNF encoding has the following Boolean variables:* $L_{A,1}$*,* $L_{A,2}$*,* $L_{B,1}$*,* $L_{B,2}$*,* $L_{C,1}$*,* $L_{C,2}$ *and* $L_{C,3}$*. Variable* $L_{A,1}$ *is* true *if the network variable* A *takes value* 1*, and* false *otherwise.*

To generate the no-good clauses in the knowledge base, we begin by iterating through the parent instantiations of the CPT for variable C*. Whenever a state* $c \in D_C$ *has a probability of 0 we will generate a clause. This clause contains the negative literal* $\neg L_{C,c}$*, as well as the negative literals* $\{\neg L_{A,a}, \neg L_{B,b}\}$ *where* $(A = a, B = b)$ *is the corresponding parent instantiation. These clauses are given in the last column of Table 3.12.*

The remaining at-least-one and at-most-one clauses are given in the table below:

at-least-one	at-most-one
$(L_{A,1} \vee L_{A,2})$	$(\neg L_{A,1} \vee \neg L_{A,2})$
$(L_{B,1} \vee L_{B,2})$	$(\neg L_{B,1} \vee \neg L_{B,2})$
$(L_{C,1} \vee L_{C,2} \vee L_{C,3})$	$(\neg L_{C,1} \vee \neg L_{C,2})$
	$(\neg L_{C,1} \vee \neg L_{C,3})$
	$(\neg L_{C,2} \vee L_{C,3})$

We evaluated the AND/OR Branch-and-Bound algorithms with static and dynamic mini-bucket heuristics on selected classes of Bayesian networks containing deterministic conditional probability tables ($i.e.$, zero probability tuples). The algorithms exploit the determin-

min-fill pseudo tree										
grid (w*, h) (n, e)	AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) i=8		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) i=10		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) i=12		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) i=14		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) i=16	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	0.31	8,080	0.11	2,052	**0.02**	101	0.05	101	0.05	101
90-10-1	0.28	7,909	0.09	2,050	0.05	101	0.06	101	0.06	101
(16, 26)	0.31	344	0.30	241	0.24	101	0.30	101	0.30	101
(100, 0)	0.52	344	0.47	241	0.39	101	0.47	101	0.47	101
	7.84	130,619	6.42	100,696	1.03	17,479	0.34	3,321	0.61	2,938
90-14-1	2.36	45,870	2.52	46,064	0.66	11,914	**0.31**	3,286	0.61	2,922
(23,37)	62.17	31,476	25.22	4,137	5.05	397	7.61	211	10.67	199
(196, 0)	33.03	10,135	16.08	3,270	4.92	396	7.72	211	10.88	199
	646.83	10,104,350	164.02	2,600,690	13.14	193,440	2.92	39,825	2.08	23,421
90-16-1	121.24	2,209,097	78.97	1,416,247	6.99	121,595	2.25	35,376	**1.84**	22,986
(26, 42)	1030.41	462,180	316.77	47,121	75.13	3,227	52.16	719	25.63	260
(256, 0)	841.32	452,923	248.38	37,670	55.86	2,264	49.99	719	25.03	260
	i=12		i=14		i=16		i=18		i=20	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	-	-	2214.12	24,117,151	1479.15	18,238,983	132.35	1,413,764	135.72	1,308,009
90-24-1	2605.56	30,929,553	689.47	9,868,626	738.17	11,100,088	**106.00**	1,282,902	121.67	1,273,738
(36, 61)	-	-	-	-	884.41	2,739	1223.18	1,228	1634.57	598
(576, 20)	-	-	-	-	843.79	2,739	1173.48	1,228	1611.74	598
	314.88	2,903,489	382.22	3,205,257	8.42	59,055	23.14	165,182	22.22	5,777
90-26-1	103.56	1,264,309	167.27	1,805,787	**6.20**	43,798	19.36	150,345	22.11	4,935
(35, 64)	-	-	-	-	938.98	2,545	1701.64	1,191	2638.95	691
(676, 40)	1592.53	108,694	1034.26	12,819	862.38	2,545	1583.37	1,191	2478.19	691
	1125.40	9,445,224	379.14	3,324,942	339.66	3,039,966	147.99	1,358,569	93.63	485,300
90-30-1	367.41	3,723,781	190.38	2,002,447	164.39	1,734,294	107.95	1,150,182	**70.14**	387,242
(38, 68)	-	-	-	-	-	-	-	-	-	-
(900, 60)	-	-	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-	462.41	1,549,829
90-34-1	-	-	-	-	-	-	-	-	**255.08**	981,831
(43, 79)	-	-	-	-	-	-	-	-	-	-
(1154, 80)	-	-	-	-	-	-	-	-	-	-
	2007.47	6,835,745	3589.43	12,321,175	800.72	2,850,393	566.11	2,079,146	368.60	1,038,065
90-38-1	410.94	1,972,430	578.54	2,339,244	270.05	1,349,223	278.11	1,249,270	**204.56**	702,806
(47, 86)	-	-	-	-	-	-	-	-	-	-
(1444, 120)	-	-	-	-	-	-	-	-	-	-

Table 3.13: CPU time and nodes visited for solving **deterministic grid networks** using static and dynamic mini-bucket heuristics. Time limit 1 hour.

ism present in the networks by applying unit resolution over the CNF encoding of the zero-probability tuples, at each node in the search tree. They are denoted by AOBB+SAT+SMB(i) and AOBB+SAT+DMB(i), respectively. We used a unit resolution scheme similar to the one employed by zChaff, a state-of-the-art SAT solver introduced by [94]. These experiments were performed on a 2.4GHz Pentium IV with 2GB of RAM running Windows XP, and therefore the CPU times reported here may be slower than those in the previous sections.

Table 3.13 shows the results for 8 grid networks from Section 3.6.3. These networks have a high degree of determinism encoded in their CPTs. Specifically, 90% of the probability tables are deterministic, containing only 0 and 1 probability entries.

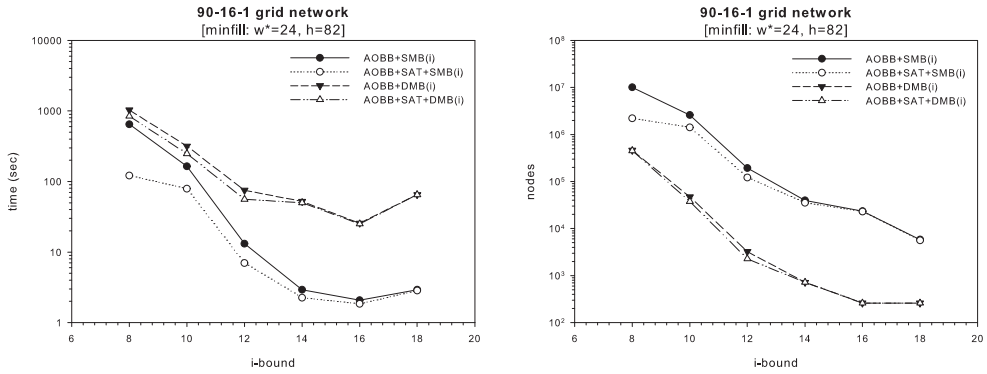


Figure 3.19: Comparison of the impact of static and dynamic mini-bucket heuristics on the **`90-16-1` deterministic grid network** from Table 3.13.

We see that `AOBB+SAT+SMB(`i`)` improves significantly over `AOBB+SMB(`i`)`, especially at relatively small i-bounds. On grid `90-26-1`, for example, `AOBB+SAT+SMB(10)` is 9 times faster than `AOBB+SMB(10)`. As the i-bound increases and the search space is pruned more effectively, the difference between `AOBB+SMB(`i`)` and `AOBB+SAT+SMB(`i`)` decreases because the heuristics are strong enough to cut the search space significantly. The mini-bucket heuristic already does some level of constraint propagation.

When comparing the AND/OR search algorithms with dynamic mini-bucket heuristics, we see that the difference between `AOBB+DMB(`i`)` and `AOBB+SAT+DMB(`i`)` is again more pronounced at small i-bounds.

Figure 3.19 displays the CPU time and number of nodes visited, as a function of the mini-bucket i-bound, on the `90-16-1` grid network ($i.e.$, corresponding to the third horizontal block from Table 3.13). We notice again the U-shaped curve of the running time for all algorithms.

Table 3.14 displays the results obtained for the 10 ISCAS'89 circuits used in Section 3.6.3. We observe that, on this domain also, constraint propagation via unit resolution does play a dramatic role rendering the search space almost backtrack-free for both static and dynamic mini-bucket heuristics and at all reported i-bounds. For instance, on the `s953` circuit, `AOBB+SAT+SMB(6)` is 3 orders of magnitude faster than `AOBB+SMB(6)` and the search space explored is about 4 orders of magnitude smaller. Similarly, on the

min-fill pseudo tree										
iscas89 (w*, h) (n, d)	AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) i=6		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) i=8		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) i=10		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) i=12		AOBB+SMB(i) AOBB+SAT+SMB(i) AOBB+DMB(i) AOBB+SAT+DMB(i) i=14	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	-	-	-	-	1079.59	20,751,699	**0.14**	432	0.24	432
c432	1658.62	37,492,131	873.71	19,423,461	4.52	89,632	0.16	432	0.23	432
(27, 45)	-	-	30.08	39,711	1.03	432	1.75	432	3.20	432
(432, 2)	0.56	434	0.69	433	1.00	432	1.70	432	3.09	432
	0.11	499	**0.09**	499	0.11	499	0.17	499	0.30	499
c499	0.10	499	**0.09**	499	0.11	499	0.17	499	0.30	499
(23, 55)	0.59	499	0.75	499	1.22	499	2.55	499	5.55	499
(499, 2)	0.59	499	0.77	499	1.19	499	2.56	499	5.59	499
	0.22	884	0.23	881	0.23	881	0.28	881	0.48	881
c880	**0.22**	881	**0.22**	881	0.25	881	0.28	881	0.47	881
(27, 67)	1.17	881	1.41	881	2.14	881	4.08	881	9.33	881
(880, 2)	1.19	881	1.35	881	2.25	881	4.03	881	9.67	881
	0.03	257	0.05	257	**0.03**	172	0.06	172	0.14	172
s386	**0.03**	172	**0.03**	172	0.05	172	0.08	172	0.14	172
(19, 44)	0.14	172	0.17	172	0.31	172	0.53	172	1.03	172
(172, 2)	0.11	172	0.16	172	0.30	172	0.52	172	1.02	172
	1019.87	9,919,295	22.50	238,780	54.77	549,181	34.74	434,481	2.61	21,499
s953	**0.19**	829	**0.19**	667	0.22	685	0.33	623	0.74	623
(66, 101)	33.03	2,738	16.52	913	48.61	1,010	17.23	468	146.66	578
(440, 2)	2.64	543	4.31	525	12.53	550	14.56	459	98.31	527
	33.00	316,875	343.50	3,682,077	7.22	77,205	31.25	320,205	26.80	289,873
s1196	**0.19**	565	0.20	565	0.23	565	0.38	565	0.92	565
(54, 97)	1.59	660	2.50	568	35.47	924	81.63	863	369.30	1,008
(560, 21)	1.17	564	2.00	563	4.61	563	13.05	563	42.02	563
	4.31	57,355	13.73	187,499	3.55	47,340	2.16	25,538	2.41	20,689
s1238	**0.20**	771	0.30	2,053	0.34	2,053	0.49	2,037	1.00	2,037
(59, 94)	2.66	1,089	3.81	795	13.77	1,824	28.03	849	62.30	744
(540, 2)	1.63	748	2.48	734	7.44	1,655	19.41	802	52.86	736
	0.27	1,986	0.47	5,171	0.48	5,078	**0.22**	866	0.34	749
s1423	0.24	1,903	0.45	4,918	0.45	4,896	**0.22**	860	0.36	749
(24, 54)	0.83	751	0.97	749	1.36	749	2.33	749	4.92	749
(748, 2)	0.81	751	0.97	749	1.37	749	2.34	749	4.92	749
	15.95	135,563	2.09	17,170	3.24	28,420	1.56	12,285	1.64	12,370
s1488	**0.22**	1,115	**0.22**	667	0.27	667	0.44	667	1.05	667
(47, 67)	1.14	670	1.67	670	3.25	668	8.11	667	25.55	667
(667, 2)	0.89	667	1.30	667	2.63	667	6.61	667	20.641	667
	15.13	158,070	43.58	476,874	11.30	118,372	17.48	198,912	3.00	21,137
s1494	**0.20**	665	0.22	665	0.25	665	0.45	665	1.11	665
(48, 69)	7.20	873	2.77	711	11.38	681	19.70	680	58.78	686
(661, 2)	1.11	665	1.75	665	3.92	665	10.41	665	31.11	665

Table 3.14: CPU time in seconds and number of nodes visited for solving belief networks corresponding to **ISCAS'89 circuits**, using static and dynamic mini-bucket heuristics. Time limit 30 minutes.

min-fill pseudo tree											
pedigree	Superlink SamIam	MBE(i) AOBB+SMB(i) AOBB+SAT+SMB(i) i=6		MBE(i) AOBB+SMB(i) AOBB+SAT+SMB(i) i=8		MBE(i) AOBB+SMB(i) AOBB+SAT+SMB(i) i=10		MBE(i) AOBB+SMB(i) AOBB+SAT+SMB(i) i=12		MBE(i) AOBB+SMB(i) AOBB+SAT+SMB(i) i=14	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
ped1		0.05		0.05		0.11		0.31		0.97	
(299, 5)	54.73	24.30	416,326	13.17	206,439	**1.58**	24,361	1.84	25,674	1.89	15,156
(15, 61)	5.44	24.72	414,239	12.97	205,887	1.59	24,361	1.86	25,674	1.89	15,156
ped38		0.12		0.45		2.20		60.97		out	
(582, 5)	**28.36**	-	-	8120.58	85,367,022	-	-	3040.60	35,394,461		
(17, 59)	out	-	-	7663.89	83,808,576	-	-	3094.33	35,394,277		
ped50		0.11		0.74		5.38		37.19		out	
(479, 5)	-	-	-	-	-	476.77	5,566,578	**104.00**	748,792		
(18, 58)	out	-	-	-	-	497.30	5,566,344	107.11	748,792		
		i=10		i=12		i=14		i=16		i=18	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
ped23		0.42		2.33		11.33		274.75		out	
(310, 5)	9146.19	498.05	6,623,197	**15.45**	154,676	16.28	67,456	286.11	117,308		
(27, 71)	out	514.33	6,618,811	15.89	154,666	17.87	67,456	270.05	117,308		
ped37		0.67		5.16		21.53		58.59		out	
(1032, 5)	**64.17**	273.39	3,191,218	1682.09	25,729,009	1096.79	15,598,863	128.16	953,061		
(21, 61)	out	282.83	3,189,847	1674.54	25,280,466	1066.79	15,372,724	131.56	953,061		
		i=12		i=14		i=16		i=18		i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
ped18		0.51		1.42		4.59		12.87		19.30	
(1184, 5)	139.06	-	-	2177.81	28,651,103	270.96	2,555,078	100.61	682,175	**20.27**	7,689
(21, 119)	157.05	-	-	2199.44	28,651,103	285.03	2,555,078	103.89	682,175	20.41	7,689
ped20		1.42		5.11		37.53		410.96		out	
(388, 5)	**14.72**	3793.31	54,941,659	1293.76	18,449,393	1259.05	17,810,674	1080.05	9,151,195		
(24, 66)	out	3953.23	54,941,659	1349.51	18,449,393	1301.26	17,810,674	1112.49	9,151,195		
ped25		0.34		0.72		2.27		6.56		29.30	
(994, 5)	-	-	-	-	-	9399.28	111,301,168	3607.82	34,306,937	**2965.60**	28,326,541
(34, 89)	out	-	-	-	-	9690.70	111,301,168	3427.79	34,306,937	2987.50	28,326,541
ped30		0.42		0.83		1.78		5.75		21.30	
(1016, 5)	13095.83	-	-	-	-	-	-	214.10	1,379,131	**91.92**	685,661
(23, 118)	out	-	-	-	-	-	-	225.67	1,379,131	96.16	685,661
ped33		0.58		2.31		7.84		33.44		112.83	
(581, 4)	-	2804.61	34,229,495	737.96	9,114,411	3896.98	50,072,988	**159.50**	1,647,488	2956.47	35,903,215
(37, 165)	out	3051.15	34,218,037	796.58	9,113,615	4290.28	50,071,828	171.31	1,647,488	3216.04	35,884,557
ped39		0.52		2.32		8.41		33.15		81.27	
(1272, 5)	322.14	-	-	-	-	4041.56	52,804,044	386.13	2,171,470	**141.23**	407,280
(23, 94)	out	-	-	-	-	4242.59	52,804,044	405.08	2,171,470	145.03	407,280
ped42		4.20		31.33		206.40		out		out	
(448, 5)	**561.31**	-	-	-	-	-	-				
(25, 76)	out	-	-	-	-	-	-				

Table 3.15: CPU time and nodes visited for solving **genetic linkage networks** using static mini-bucket heuristics. Time limit 3 hours.

same network, AOBB+SAT+DMB(6) is 12 times faster than AOBB+DMB(4) and explores about 5 times fewer nodes. Notice that in the case of dynamic mini-bucket heuristics, the difference between AOBB+SAT+DMB(i) and AOBB+DMB(i) is not too prominent as in the static case, because the heuristic estimates prune the search space quite effectively.

Table 3.15 shows the results obtained for the 12 linkage analysis networks from Section 3.6.3. In this case, we observe that applying unit resolution was not cost effective.

minfill pseudo tree										
spot5 (w*, h) (n, k, c)	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=6		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=8		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=12		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=14		AOEDAC toolbar	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	0.03		0.34		21.72		147.66			
29	-	-	-	-	25.69	5,095	148.27	632	613.79	8,997,894
(14, 42)	4.83	45,509	**0.64**	2,738	21.74	246	147.69	481	4.56	218,846
(83, 4, 476)	131.64	9,713	57.22	541	678.22	507	1758.78	507		
	65.91	11,850	53.72	364	630.09	330	1675.74	330		
	0.11		0.50		28.81		223.14			
42b	-	-	2154.64	9,655,444	148.11	712,685	228.17	12,255	-	-
(18, 62)	-	-	1790.76	9,606,846	**131.34**	689,402	223.64	4,189	-	-
(191, 4, 1341)	-	-	-	-	-	-	-	-		
	-	-	-	-	-	-	-	-		
	0.02		0.09		1.25		1.23			
54	2.98	27,383	0.59	4,996	1.28	921	1.52	921	31.34	823,326
(11, 33)	1.50	17,757	0.34	3,616	1.28	329	1.27	329	**0.31**	21,939
(68, 4, 283)	52.44	2,469	38.63	921	464.58	921	465.35	921		
	27.27	2,188	21.91	329	266.55	329	265.89	329		
	0.02		0.09		1.09		4.03			
404	-	-	-	-	4009.57	32,763,223	1827.05	15,265,025	255.83	3,260,610
(19, 42)	146.05	1,373,846	14.08	144,535	**1.39**	3,273	4.06	367	151.11	6,215,135
(100, 4, 710)	-	-	-	-	-	-	1964.20	2,015		
	272.46	39,144	215.17	5,612	565.06	1,327	167.90	220		
	0.08		0.31		8.30		35.22			
408b	-	-	-	-	-	-	-	-	-	-
(24, 59)	-	-	-	-	682.12	4,784,407	**124.67**	567,407	-	-
(201, 4, 1847)	-	-	-	-	-	-	-	-		
	-	-	-	-	-	-	-	-		
	0.03		0.14		0.39		0.39			
503	-	-	-	-	1.22	5,229	1.22	5,229	-	-
(9, 39)	412.63	5,102,299	397.77	4,990,898	**0.44**	641	**0.44**	641	-	-
(144, 4, 639)	-	-	-	-	690.44	5,229	694.86	5,229		
	-	-	-	-	64.02	641	64.52	641		
	0.01		0.12		48.20		372.27			
505b	-	-	-	-	-	-	-	-	-	-
(16, 98)	-	-	-	-	-	-	**392.08**	143,371	-	-
(240, 4, 1721)	-	-	-	-	-	-	-	-		
	-	-	-	-	-	-	-	-		

Table 3.16: CPU time and nodes visited for solving **SPOT5 networks**. Time limit 2 hours.

3.6.5 Results for Empirical Evaluation on Weighted CSPs

In this section we focus on both mini-bucket and EDAC heuristics when problems are solved in a static variable ordering. We also evaluate the impact of dynamic variable orderings when using EDAC based heuristics.

SPOT5 Benchmark

SPOT5 benchmark contains a collection of large real scheduling problems for the daily management of Earth observing satellites [7]. These problems can be described as follows:

- Given a set $\mathbf{P}$ of photographs which can be taken the next day from at least one of the three instruments, w.r.t. the satellite trajectory;
- Given, for each photograph, a weight expressing its importance;

- Given a set of imperative constraints: non overlapping and minimal transition time between two successive photographs on the same instrument, limitation on the instantaneous data flow through the satellite telemetry;

- The goal is to find an admissible subset $\mathbf{P}'$ of $\mathbf{P}$ which maximizes the sum of the weights of the photographs in $\mathbf{P}'$ when all imperative constraints are satisfied.

They can be casted as WCSPs by:

- Associating a variable X_i with each photograph $p_i \in \mathbf{P}$;

- Associating with X_i a domain D_i to express the different ways of achieving p_i and adding to D_i a special value, called *rejection* value, to express the possibility of not selecting the photograph p_i;

- Associating with every X_i an unary constraint forbidding the rejection value, with a valuation equal to the weight of p_i;

- Translating as imperative constraints (binary or ternary) the constraints of non overlapping and minimal transition time between two (or three) photographs on the same instrument, and of limitation on the instantaneous data flow. Each imperative constraint is defined over a subset of two or three photographs and for each value combination of its scope variables it associates a high penalty cost (10^6) if the corresponding photographs cannot be taken simultaneously, on the same instrument.

The task is to compute: $min_{\mathbf{X}} \sum_{i=1}^{r} f_i$, where r is the number of unary, binary and ternary cost functions in the problem.

Table 3.16 reports the results obtained for experiments with 7 SPOT5 networks, using min-fill pseudo trees. We see that `AOBB+SMB(`i`)` is the best performing algorithm on this dataset. The overhead of the dynamic mini-bucket heuristics outweighs search pruning here. For instance, on the `404` network, the difference between `AOBB+SMB(12)` and

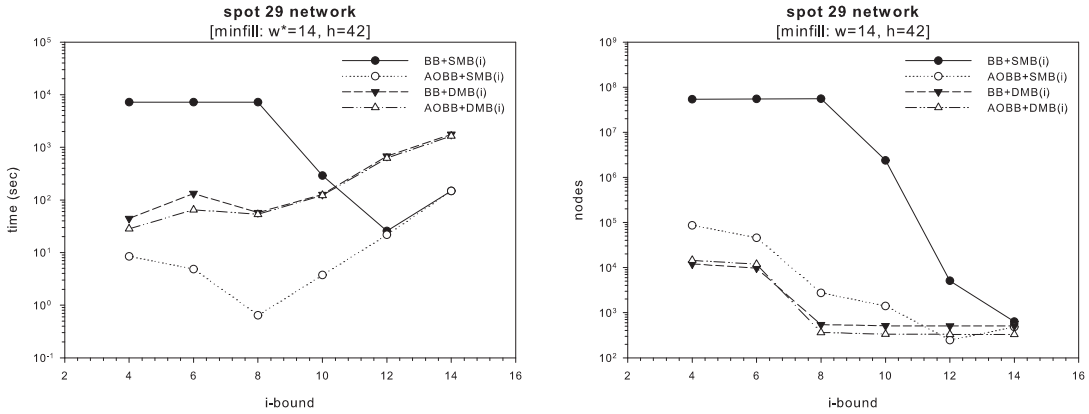


Figure 3.20: Comparison of the impact of static and dynamic mini-bucket heuristics on the **29 SPOT5 instance** from Table 3.16.

BB+SMB(12), in terms of runtime and size of the search space explored, is up to 3 orders of magnitude. The best performances on this domain are obtained by AOBB+SMB(i) at relatively large i-bounds which generate very accurate heuristic estimates. For example, AOBB+SMB(14) is the only algorithm able to solve the 505b network. AOEDAC and toolbar were able to solve relatively efficiently only 3 out of the 7 test instances ($e.g.$, 29, 54 and 404).

In Figure 3.20 we plot the running time and number of nodes visited by AOBB+SMB(i) and AOBB+DMB(i) (resp. BB+SMB(i) and BB+DMB(i)), as a function of the i-bound, on the 29 SPOT5 network ($i.e.$, corresponding to the first horizontal block from Table 3.16). In this case AOBB+DMB(i) (resp. BB+DMB(i)) is inferior to AOBB+SMB(i) (resp. BB+DMB(i)) across all reported i-bounds. We see that AOBB+SMB(i) achieves the best performance at $i = 8$, whereas AOBB+DMB(i) performs best only at the smallest reported i-bound, namely $i = 4$.

Figure 3.21 displays the runtime distribution of AOBB+SMB(i) guided by hypergraph based pseudo trees, over 20 independent runs. Hypergraph based trees have far smaller depths than the min-fill ones, and therefore are again able to improve the runtime over min-fill based ones only at relatively small i-bounds ($e.g.$, 404). On average, however, the min-fill pseudo trees generally yield a more robust performance, especially for larger

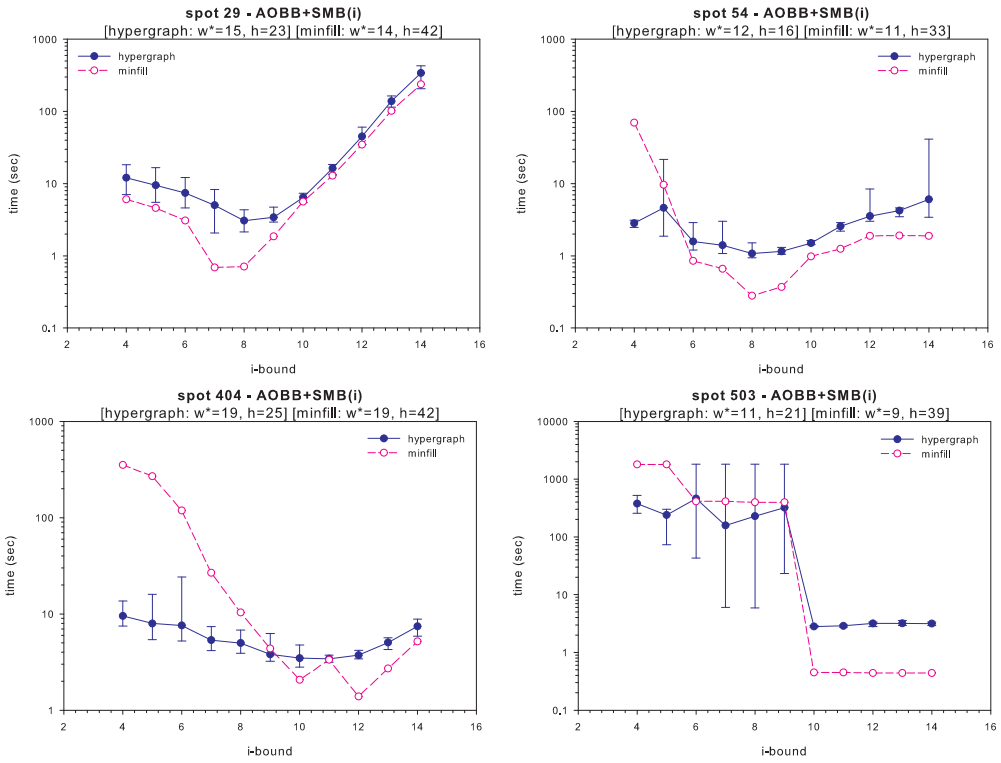


Figure 3.21: Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. CPU time in seconds for solving **SPOT5 networks** with AOBB+SMB(i).

i-bounds of the mini-bucket heuristics ($e.g.$, 503).

ISCAS'89 Circuits (WCSP)

For our purpose, we converted each of the ISCAS'89 circuits into a non-binary WCSP instance by removing flip-flops and buffers in a standard way and creating for each gate a cost function that assigns a high penalty cost (1000) to the forbidden tuples. For each of the input signals we created, in addition, a unary cost function with penalty costs distributed uniformly at random between 1 and 10.

Table 3.17 shows the results for experiments with 10 circuits, using min-fill pseudo trees. The EDAC based algorithms performed very poorly on this dataset and could not solve any of the test instances within the 30 minute time limit. This was due to the relatively large arity of the constraints, with up to 10 variables in their scope.

AOBB+SMB(i) is superior, especially at relatively large i-bounds. For example, on the

minfill pseudo tree										
iscas89 (w*, h) (n, d)	MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=6		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=8		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=12		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=14		MBE(i) BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=16	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	0.06		0.07		0.14		0.27		0.89	
c432	-	-	-	-	13.27	103,088	13.29	102,546	6.79	34,671
(27, 45)	-	-	1373.07	23,355,897	3.85	76,346	3.89	75,420	**0.97**	1,958
(432, 2)	-	-	104.57	35,073	485.61	70,401	125.95	35,502	122.09	35,609
	-	-	3.04	1,578	26.26	16,482	9.17	1,070	6.67	692
	0.03		0.04		0.14		0.36		0.99	
c499	-	-	-	-	11.71	53,171	9.62	63,177	5.80	24,397
(23, 55)	4.72	117,563	61.48	1,265,425	**0.83**	18,851	24.18	486,656	1.80	22,065
(499, 2)	56.49	29,664	141.89	78,830	65.42	40,123	132.20	56,002	203.74	76,832
	3.87	10,147	23.31	13,529	5.71	1,002	37.34	3,353	87.99	1,736
	0.06		0.07		0.16		0.49		1.48	
c880	-	-	-	-	-	-	-	-	816.47	4,953,611
(27, 67)	2284.65	39,448,762	957.25	19,992,512	275.51	5,835,825	607.43	13,568,696	137.31	2,837,010
(880, 2)	2463.80	321,585	-	-	2461.68	270,166	3532.50	410,360	2817.47	238,297
	28.43	40,057	809.53	796,699	101.88	32,748	232.97	36,187	625.50	20,357
	0.01		0.01		0.06		0.19		0.46	
s386	3.26	31,903	0.48	5,118	0.51	5,108	0.61	4,543	0.86	4,543
(19, 44)	0.12	3,705	**0.07**	2,073	0.14	2,699	0.22	1,420	0.49	1,420
(172, 2)	2.92	4,543	3.14	4,543	4.46	4,543	5.92	4,543	8.64	4,543
	0.42	1,420	0.65	1,420	1.98	1,420	3.44	1,420	6.13	1,420
	0.06		0.07		0.31		1.00		3.35	
s953	-	-	-	-	-	-	-	-	-	-
(66, 101)	-	-	1734.71	21,438,706	-	-	28.40	348,699	7.14	51,441
(440, 2)	110.11	100,180	125.49	103,086	466.71	106,825	1412.68	107,063	1094.88	103,383
	6.44	6,885	17.49	7,400	350.17	9,164	1294.39	11,164	984.06	8,377
	0.06		0.08		0.37		1.27		4.51	
s1196	-	-	-	-	-	-	-	-	-	-
(54, 97)	-	-	-	-	3146.04	34,576,509	1281.38	15,775,180	269.73	3,318,953
(560, 21)	828.59	217,500	1126.06	216,777	-	-	-	-	-	-
	39.22	26,501	62.99	21,849	355.39	15,443	1443.72	13,687	-	-
	0.06		0.09		0.41		1.25		4.72	
s1238	-	-	-	-	-	-	-	-	-	-
(59, 94)	2245.60	32,501,292	-	-	1061.12	18,302,873	821.55	14,213,319	**26.13**	360,788
(540, 2)	2744.88	294,977	1661.09	141,562	-	-	-	-	-	-
	142.51	44,980	288.25	39,493	844.40	20,945	1449.22	13,857	-	-
	0.04		0.05		0.12		0.33		0.94	
s1423	-	-	-	-	-	-	-	-	167.07	448,044
(24, 54)	25.97	309,520	51.60	648,520	**5.03**	68,102	5.50	70,043	7.62	87,483
(748, 2)	-	-	-	-	1969.46	539,925	2056.07	565,423	2156.59	579,511
	57.03	52,996	27.67	26,772	38.85	19,719	31.92	3,513	56.80	4,323
	0.06		0.09		0.45		1.50		5.43	
s1488	-	-	-	-	20.49	58,330	21.56	58,859	23.59	50,080
(47, 67)	1076.11	13,244,002	4.79	50,613	**3.08**	29,729	4.28	33,827	6.63	17,904
(667, 2)	192.51	48,822	204.68	49,417	286.90	50,803	495.13	50,803	1205.42	50,803
	11.58	15,025	18.02	15,064	94.05	13,762	304.60	13,762	1022.09	13,762
	0.08		0.10		0.50		1.57		5.66	
s1494	3483.40	11,667,673	94.08	362,002	396.38	1,544,960	22.78	66,745	26.81	68,848
(48, 69	345.91	3,076,992	91.55	833,720	343.58	3,207,718	**9.06**	83,318	17.01	124,765
(661, 2)	233.36	55,236	279.75	59,161	350.23	53,067	391.96	47,139	1431.41	48,119
	41.40	21,156	64.60	21,743	162.70	15,699	232.34	9,706	1260.97	9,913

Table 3.17: CPU time and nodes visited for solving **ISCAS'89 circuits** as WCSPs. Time limit 1 hour. `AOEDAC` and `toolbar` were not able to solve any of the test instances within the time limit.

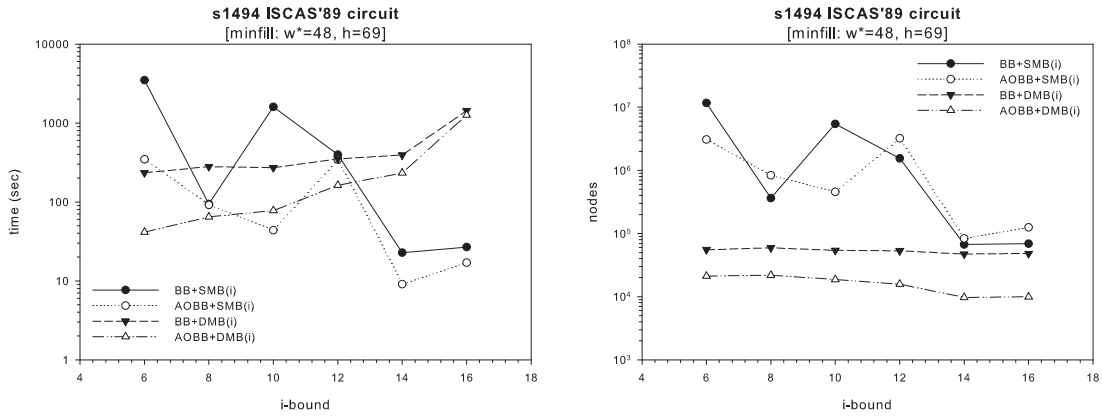


Figure 3.22: Comparison of the impact of static and dynamic mini-bucket heuristics on the **s1494 ISCAS'89 circuit** from Table 3.17.

`s1238` circuit, `AOBB+SMB(16)` finds the optimal solution in about 26 seconds, whereas `BB+SMB(16)` as well as `AOBB+DMB(16)` and `BB+DMB(16)` exceed the time limit. In this case, `AOBB+DMB(`i`)` is competitive at relatively small i-bounds, which cause a relatively small computational overhead. For instance, `AOBB+DMB(`6`)` is the best performing algorithm on the `s953` network. It is 18 times faster and expands 14 times fewer nodes than `BB+DMB(6)`.

In Figure 3.22 we show the running time and size of the search space explored by `AOBB+SMB(`i`)` and `AOBB+DMB(`i`)` (resp. `BB+SMB(`i`)` and `BB+DMB(`i`)`), as a function of the i-bound, on the `s1494` ISCAS'89 circuit ($i.e.$, corresponding to the last horizontal block from Table 3.17). We see that the power of the dynamic mini-bucket heuristics is again more prominent for relatively small i-bounds. At larger i-bounds, the static mini-bucket heuristics are cost effective, namely the difference in running time between `AOBB+SMB(`i`)` and `AOBB+DMB(`i`)` (resp. between `BB+SMB(`i`)` and `BB+DMB(`i`)`) is about two orders of magnitude in favor of the former.

Figure 3.23 depicts the runtime distribution of `AOBB+SMB(`i`)` guided by hypergraph based pseudo trees on the instances: `c499`, `c880`, `s1238` and `s1488`, respectively. In some cases ($e.g.$, `s1238`), using hypergraph pseudo trees improves the runtime up to one order of magnitude, compared with min-fill ones.

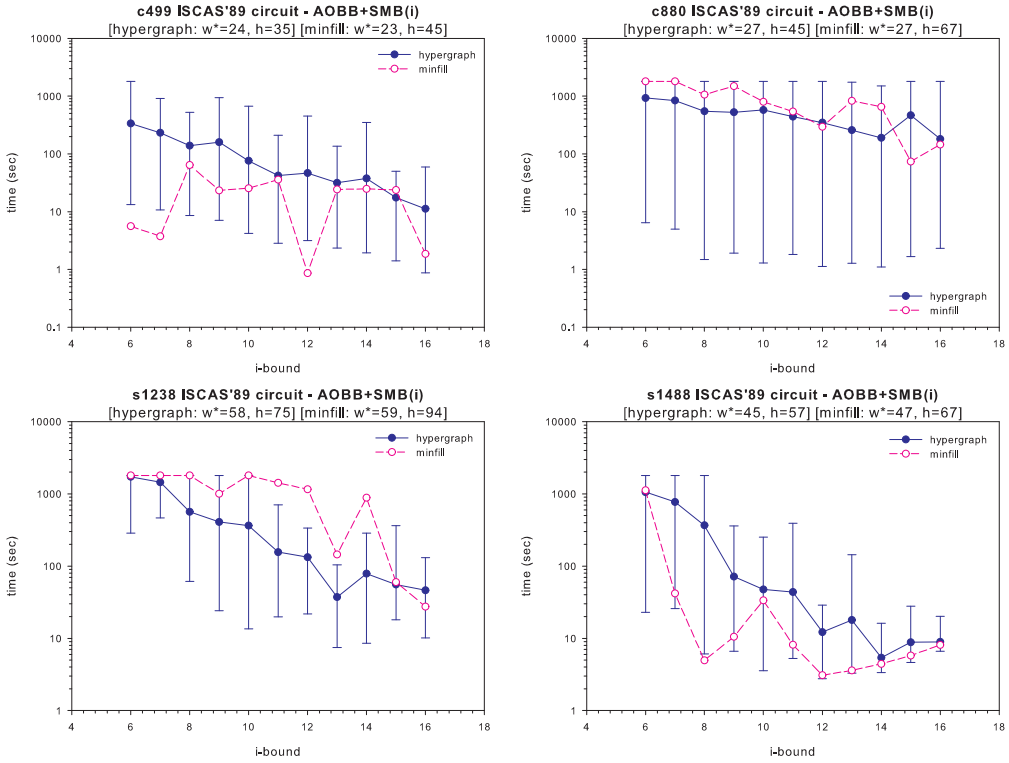


Figure 3.23: Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. CPU time in seconds for solving **ISCAS'89 networks** with AOBB+SMB(i).

Mastermind Games

Each of these networks is a ground instance of a relational Bayesian network that models differing sizes of the popular game of Mastermind. These networks were produced by the PRIMULA System[9] and used in experimental results from [17]. For our purpose, we converted these networks into equivalent WCSP instances by taking the negative log probability of each conditional probability table entry and rounding it to the nearest integer. The resulting WCSP instances are quite large with the number of bi-valued variables n ranging between 1220 and 3692, and containing n unary and ternary cost functions.

Table 3.18 shows the results for experiments with 6 game instances of increasing difficulty, using min-fill based pseudo trees. As before, AOBB+SMB(i) offers the overall best performance. For example, AOBB+SMB(10) solves the mm-04-08-03 instance in about

[9] http://www.cs.auc.dk/jaeger/Primula

minfill pseudo tree												
mastermind	MBE(i) BB+SMB(i) AOBB+SMB(i)		MBE(i) BB+SMB(i) AOBB+SMB(i)		MBE(i) BB+SMB(i) AOBB+SMB(i)		MBE(i) BB+SMB(i) AOBB+SMB(i)		MBE(i) BB+SMB(i) AOBB+SMB(i)		MBE(i) BB+SMB(i) AOBB+SMB(i)	
(w*, h)	i=8		i=10		i=12		i=14		i=16		i=18	
(n, r, k)	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
mm-03-08-03	0.17		0.22		0.35		0.91		2.83		7.99	
(20, 57)	-	-	-	-	-	-	897.87	873,606	946.84	915,095	738.13	720,764
(1220, 3, 2)	1.16	10,369	**0.88**	7,075	0.93	6,349	1.23	3,830	3.11	3,420	8.25	3,153
mm-03-08-04	0.48		0.60		0.89		2.08		6.45		25.15	
(33, 87)	-	-	-	-	-	-	-	-	-	-	-	-
(2288, 3, 2)	72.37	150,642	66.69	193,805	36.22	71,622	**10.15**	31,177	25.16	63,669	29.27	13,870
mm-04-08-03	0.21		0.27		0.48		1.06		3.54		12.52	
(26, 72)	-	-	-	-	1609.86	1,315,415	1603.71	1,175,430	1157.09	901,309	1924.02	1,451,854
(1418, 3, 2)	8.20	68,929	**3.05**	26,111	4.23	34,445	3.10	17,255	5.29	15,443	13.71	10,570
	i=12		i=14		i=16		i=18		i=20		i=22	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
mm-04-08-04	1.19		2.35		6.85		26.47		106.37		395.57	
(39, 103)	-	-	-	-	-	-	-	-	-	-	-	-
(2616, 3, 2)	324.06	744,993	166.67	447,464	310.06	798,507	**64.72**	107,463	192.39	242,865	414.54	62,964
mm-03-08-05	2.14		4.54		11.82		39.01		134.46		497.45	
(41, 111)	-	-	-	-	-	-	-	-	-	-	-	-
(3692, 3, 2)	-	-	-	-	-	-	**835.90**	1,122,008	1162.22	1,185,327	1200.65	1,372,324
mm-10-08-03	1.48		3.78		11.39		34.53		127.55		593.25	
(51, 132)	-	-	-	-	-	-	-	-	-	-	-	-
(2606, 3, 2)	109.50	290,594	128.29	326,662	**64.31**	151,128	74.14	127,130	169.84	133,112	623.83	79,724

Table 3.18: CPU time and nodes visited for solving **Mastermind game instances** using static mini-bucket heuristics. Time limit 1 hour. `AOEDAC` and `toolbar` did not solve any of the test instances within the time limit.

3 seconds, whereas `BB+SMB(10)` exceeds the 1 hour time limit. We did not report results with dynamic mini-bucket heuristics because of the prohibitively large computational overhead associated with relatively large i-bounds. We also note that the EDAC based algorithms were not able to solve any of these instances within the alloted time bound (not shown in the table).

In Figure 3.24 we display the runtime distribution of `AOBB+SMB(`i`)` guided by hypergraph based pseudo trees over 20 independent runs, for 4 test instances. The spectrum of results is similar to what we observed earlier.

3.6.6 The Impact of Dynamic Variable Orderings

In this section we evaluate the impact of dynamic variable orderings on AND/OR Branch-and-Bound search guided by local consistency (EDAC) based heuristics. We did not use dynamic variable orderings with dynamic mini-bucket heuristics because of the prohibitively large computational overhead.

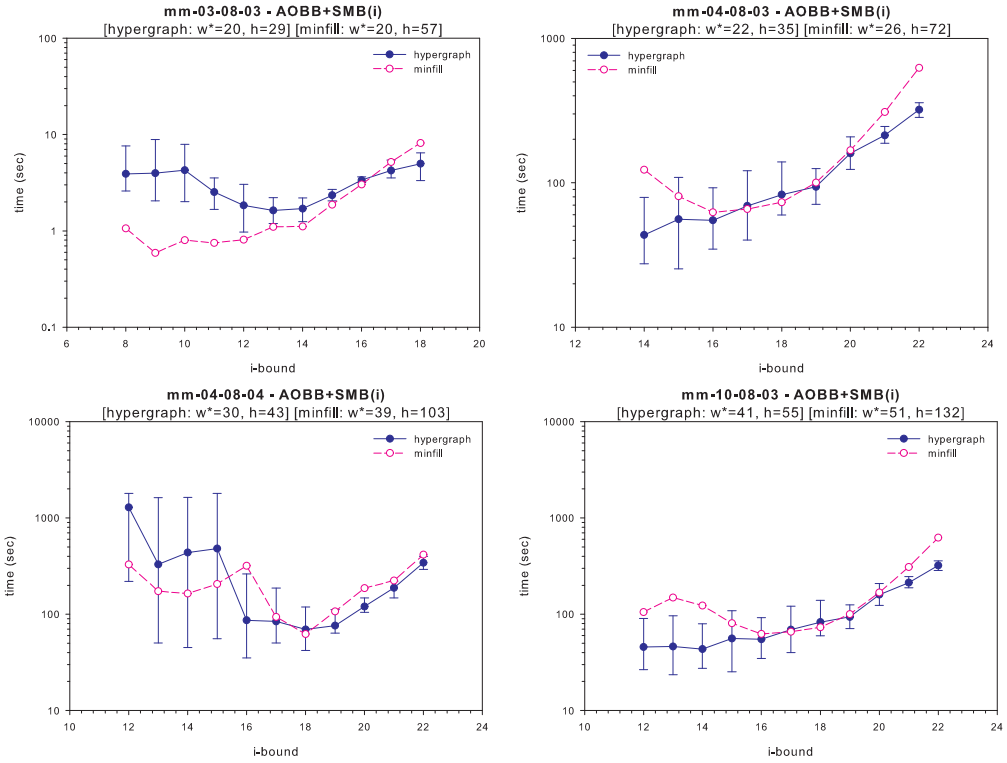


Figure 3.24: Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. CPU time in seconds for solving **Mastermind networks** with AOBB+SMB(i).

SPOT5 Benchmark

Table 3.19 shows the results for experiments with the 7 SPOT5 networks described in Section 3.6.5. We see that variable ordering can have a tremendous impact on performance. AOEDAC+DSO is the best performing among the EDAC based algorithms, and is able to solve 6 out of 7 test instances. The second best algorithm in this category is DVO+AOEDAC which solves relatively efficiently 3 test networks. This demonstrates the benefit of using variable ordering heuristics within AND/OR Branch-and-Bound search. We also observe that the best performance points highlighted in Table 3.19 are inferior to those from Table 3.16 corresponding to AOBB+SMB(i). For example, on the 42b network, the difference in runtime and size of the search space explored between AOBB+SMB(12) and AOEDAC+DSO is up to one order of magnitude in favor of the former. Similarly, the 505b network could not be solved by any of the EDAC based algorithms, whereas

minfill pseudo tree									
spot5	n c	w* h		toolbar	BBEDAC	AOEDAC	AOEDAC+PVO	DVO+AOEDAC	AOEDAC+DSO
29	16 57	7 8	time nodes	4.56 218,846	109.66 710,122	613.79 8,997,894	545.43 7,837,447	**0.83** 8,698	11.36 92,970
42b	14 75	9 9	time nodes	- -	- -	- -	- -	- -	**6825.4** 27,698,614
54	14 75	9 9	time nodes	0.31 21,939	0.97 8,270	31.34 823,326	9.11 90,495	**0.06** 688	0.75 6,614
404	16 89	10 12	time nodes	151.11 6,215,135	2232.89 7,598,995	255.83 3,260,610	152.81 1,984,747	12.09 88,079	**1.74** 14,844
408b	18 106	10 13	time nodes	- -	- -	- -	- -	- -	**747.71** 2,134,472
503	22 131	11 15	time nodes	- -	- -	- -	- -	- -	**53.72** 231,480
505b	16 70	9 10	time nodes	- -	- -	- -	- -	- -	- -

Table 3.19: CPU time and nodes visited for solving **SPOT5 benchmarks** with EDAC heuristics and dynamic variable orderings. Time limit 2 hours.

`AOBB+SMB(14)` finds the optimal solution in about 6 minutes. Notice that `toolbar` is much better than `BBEDAC` in all test cases. This can be explained by a more careful and optimized implementation of EDAC which is available in `toolbar`.

In Figure 3.25 we show the runtime distribution of `AOEDAC+PVO` with hypergraph pseudo trees on 20 independent runs. In this case, the difference between the min-fill and the hypergraph case is dramatic, resulting in up to three orders of magnitude in favor of the latter.

CELAR Benchmark

Radio Link Frequency Assignment Problem (RLFAP) is a communication problem where the goal is to assign frequencies to a set of radio links in such a way that all links may operate together without noticeable interferences [15]. It can be naturally casted as a binary WCSP where each forbidden tuple has an associated penalty cost.

Table 3.20 shows detailed results for experiments with `CELAR6` and `CELAR7` sub-instances. We considered only the OR and AND/OR using EDAC heuristics. The performance of the mini-bucket based algorithms was quite poor on this domain, due to the very low quality of the heuristic estimates resulted from approximating subproblems with very large domains (up to 44 values).

We observe that `toolbar` is the overall best performing algorithm on this dataset.

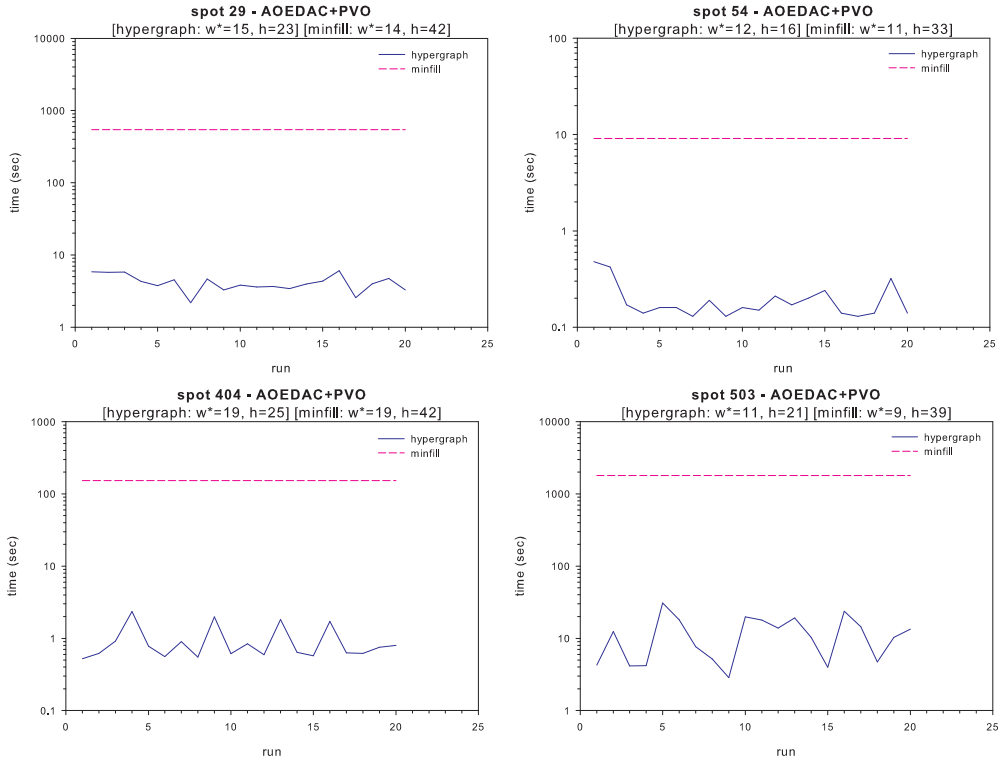


Figure 3.25: Min-Fill versus Hypergraph partitioning heuristics for pseudo tree construction. CPU time in seconds for solving **SPOT5 networks** with AOEDAC+PVO.

One reason is that h is close to n, so the AND/OR search is close to OR search. When looking at the AND/OR algorithms we notice that DVO+AOEDAC offers the best performance. On average, the speedups caused by DVO+AOBB over the other algorithms are as follows: 1.9x over AOEDAC, 1.6x over AOEDAC+PVO and 2.5x over BBEDAC. Furthermore, AOEDAC+DSO performs similarly to AOEDAC+PVO indicating that the quality of the dynamic problem decomposition is comparable to the static one.

Random Binary WCSPs

A random binary WCSP class [117] is defined by $\langle n, d, c, t \rangle$ where n is the number of variables, d is the domain size, c is the number of binary constraints ($i.e.$, graph *connectivity*), and t the number of forbidden tuples in each constraint ($i.e.$, *tightness*). Pairs of constrained variables and their forbidden tuples are randomly selected using a uniform distribution.

Using this model we first experimented with the following 6 classes of random binary

minfill pseudo tree									
celar	n c	w* h		toolbar	BBEDAC	AOEDAC	AOEDAC+PVO	DVO+AOEDAC	AOEDAC+DSO
celar6-sub0	16 57	7 8	time nodes	**0.66** 8,952	0.88 2,985	1.20 2,901	0.79 1,565	0.82 2,652	0.67 1,633
celar6-sub1	14 75	9 9	time nodes	**488.58** 7,521,496	5079.28 6,381,472	6693.33 5,558,900	4972.42 4,376,510	4961.16 4,420,050	4999.17 4,326,480
celar6-sub1-24	14 75	9 9	time nodes	**47.80** 1,028,814	269.88 716,746	319.20 512,419	251.11 446,808	248.55 440,238	252.65 440,857
celar6-sub2	16 89	10 12	time nodes	**1887.40** 30,223,624	6579.99 10,941,839	23896.83 21,750,156	12026.15 8,380,049	6097.33 6,700,589	11323.30 5,584,139
celar6-sub3	18 106	10 13	time nodes	**4376.37** 61,700,735	14686.60 63,304,285	32439.00 39,352,900	28251.70 32,467,100	11131.00 28,803,649	28407.40 32,451,800
celar6-sub4-20	22 131	11 15	time nodes	**27.76** 167,960	1671.55 8,970,211	277.51 522,981	415.02 952,894	268.57 893,609	413.48 1,256,102
celar7-sub0	16 70	9 10	time nodes	**1.11** 6,898	4.56 9,146	6.20 10,248	5.00 10,198	4.64 9,151	4.71 9,761
celar7-sub1	14 75	9 9	time nodes	**23.86** 134,404	188.11 501,145	470.36 589,117	239.20 329,236	189.15 372,790	245.41 318,351
celar7-sub1-20	14 75	9 9	time nodes	**0.67** 10,438	3.49 18,959	14.09 27,805	3.56 15,860	3.30 15,637	3.33 14,351
celar7-sub2	16 89	10 11	time nodes	**627.97** 1,833,808	4822.89 4,026,263	7850.10 7,644,780	5424.98 3,454,750	4727.30 3,326,511	5545.80 2,654,120
celar7-sub3	18 106	10 13	time nodes	**6944.96** 14,754,723	-	-	-	-	-
celar7-sub4-22	22 129	11 15	time nodes	3604.47 6,391,923	23882.20 23,700,235	26210.05 34,941,835	7958.44 11,533,163	23166.40 23,674,049	**2999.55** 3,429,708

Table 3.20: CPU time and nodes visited for solving **CELAR6** and **CELAR7** sub-instances with EDAC heuristics and dynamic variable orderings. Time limit 10 hours.

min-fill pseudo tree												
wcsp (n, d, c, t) (w*, h)	BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=2		BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=4		BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=6		BB+SMB(i) AOBB+SMB(i) BB+DMB(i) AOBB+DMB(i) i=8		AOEDAC AOEDAC+PVO DVO+AOEDAC AOEDAC+DSO		toolbar BBEDAC	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
			161.86	1,043,900	53.00	385,426	12.30	64,612	7.33	28,831		
medium	-	-	159.46	1,085,370	51.98	388,580	12.44	67,733	2.80	11,404	**0.28**	1,910
(20,5,100,0.7)	144.44	69,690	134.17	4,935	160.49	506	-	-	2.38	9,721	2.39	9,736
(12, 17)	144.06	73,587	134.01	4,997	160.40	484	-	-	2.40	9,796		
			157.21	962,935	26.99	187,522	4.87	15,513	5.70	21,017		
low	-	-	131.08	1,243,260	25.04	249,556	5.01	22,523	2.79	11,121	**0.14**	1,487
(30,5,90,0.7)	128.07	62,836	87.10	3,708	122.07	877	160.84	527	1.08	4,383	1.08	4,532
(11, 19)	127.38	103,388	87.17	4,805	122.02	587	160.61	223	1.25	5,117		
	-	-	94.71	660,928	2.16	15,365	1.36	2,754	0.69	3,801		
sparse	102.12	1,240,740	1.93	32,717	0.29	3,767	1.09	470	0.70	3,876	**0.07**	2,256
(50,5,80,0.7)	79.78	106,779	6.49	3,049	8.08	2,722	10.48	2,640	0.31	1,881	0.37	2,233
(8, 16)	19.55	33,801	4.72	810	6.22	299	8.64	237	0.48	2,937		
	-	-	-	-	274.51	772,298	-	-	133.49	353,812		
medium	-	-	-	-	274.62	806,001	-	-	41.47	101,627	**2.80**	11,079
(20,10,100,0.5)	-	-	-	-	-	-	-	-	33.86	83,060	33.91	83,074
(12, 17)	-	-	-	-	-	-	-	-	33.96	85,164		
	-	-	251.91	613,804	107.21	265,294	-	-	0.60	3,111		
low	254.01	536,805	174.37	520,488	96.50	316,600	-	-	0.60	3,061	**0.06**	1,502
(30,10,90,0.5)	181.47	18,857	-	-	-	-	-	-	0.23	1,318	0.23	1,400
(11, 19)	176.83	25,377	-	-	-	-	-	-	0.28	1,549		
	255.20	825,232	165.96	491,965	21.20	50,687	197.53	5,836	0.02	193		
sparse	39.47	146,927	8.35	40,152	5.08	2,977	201.83	2,564	0.02	182	**0.01**	1,525
(50,10,80,0.5)	17.78	5,186	38.83	3,743	196.72	3,554	-	-	0.02	192	0.07	362
(8, 16)	0.64	644	13.79	624	179.45	570	-	-	0.03	274		

Table 3.21: CPU time in seconds and number of nodes visited for solving **random binary WCSPs**. Time limit 5 minutes.

WCSPs:

1 $\langle 20, 5, 100, 0.7 \rangle$	2 $\langle 30, 5, 90, 0.7 \rangle$	3 $\langle 50, 5, 80, 0.7 \rangle$
4 $\langle 20, 10, 100, 0.7 \rangle$	5 $\langle 30, 10, 90, 0.7 \rangle$	6 $\langle 50, 10, 80, 0.7 \rangle$

Classes 1 and 4 have medium connectivity, classes 2 and 5 have low connectivity, and classes 3 and 6 represent sparse problems. For each problem class we chose the tightness to obtain over-constrained instances, and the penalty cost of the forbidden tuples was selected uniformly at random between 1 and 10.

In Table 3.21 we give the detailed results of the experiment with 20 random instances from each problem class. The columns are indexed by the i-bound of the mini-bucket heuristics. We allowed each algorithm a 180 second time limit for problem classes 1, 2 and 3, and a 300 second time limit for classes 4, 5 and 5, respectively. The guiding pseudo-tree of the AND/OR Branch-and-Bound algorithms was constructed using the *min-fill* heuristic.

When comparing the mini-bucket based algorithms, we observe that AND/OR Branch-and-Bound with static mini-buckets, AOBB+SMB(i), offers the best performance, especially for relatively sparse problems and larger i-bounds. For example, on problem class 3, AOBB+SMB(6) is 7 times faster than BB+SMB(i), and explores 5 times fewer nodes. Alternatively, the AND/OR Branch-and-Bound with dynamic mini-buckets AOBB+DMB(i) is superior only for the smallest reported i-bound. For instance, on problem class 6, AOBB+DMB(2) causes a speed up in CPU time of 27 over BB+DMB(2), while exploring a search space 9 times smaller.

The overall best performance on this dataset is obtained by the EDAC based algorithms, in particular by toolbar, which outperforms dramatically (with up to several orders of magnitude) the mini-bucket based algorithms. When comparing the AND/OR Branch-and-Bound with EDAC based heuristics, we observe that AOEDAC+PVO improves over the static AOEDAC, especially on problems with low and medium graph connectivity. DVO+AOBB is only slightly better than BBEDAC, which indicates that the semantic variable selection heuristic is strong enough to shrink the search space significantly, thus

leaving not much room for additional problem decompositions. AOEDAC+DSO is the third better among these algorithms, showing the effectiveness of computing the separators dynamically. The difference between BBEDAC and `toolbar` can be explained by a more efficient implementation of EDAC which is available in `toolbar`.

For our second experiment we consider four classes of random binary WCSPs with domain size and maximum penalty cost for forbidden tuples of 10, as described in [71]. For fixed values of n, d and c, and increasing tightness t, most problems are solved almost instantly until a cross-over point is reached. Then problems become harder and much harder to solve. We denote t^o the lowest tightness where every instance is over-constrained. Based on this different categories of problems can be defined as follows:

- For graph density, we defined two problem types: *sparse* (S) with $c = 2.5n$, and *dense* (D) with $c = \frac{n(n-1)}{8}$;
- For tightness, we define two problem types: *loose* (L) with $t = t^o$, and *tight* (T) with $t = d^2 - 0.25t^o$.

Combining the different types, we obtain 4 different classes, each being denoted by a pair of characters (SL, ST, DL and DT). In each class, the domain size and maximum penalty cost are set to 10, and the number of variables n is used as a varying parameter.

Figures 3.26 and 3.27 display the average CPU time results in seconds. Each data point represents an average over 20 samples. When comparing AOEDAC+PVO with static AOEDAC we notice a considerable improvement in terms of both running time and size of search space explored. AOEDAC+DSO has a similar performance as AOEDAC+PVO indicating that both algorithms use decompositions of similar quality. The best performance of all 4 problem classes is offered by DVO+AOEDAC and BBEDAC with no clear winner between the two. This implies that the semantic ordering heuristic is powerful and it does not leave much room for additional problem decompositions. The overall best performance is offered on this dataset by `toolbar`.

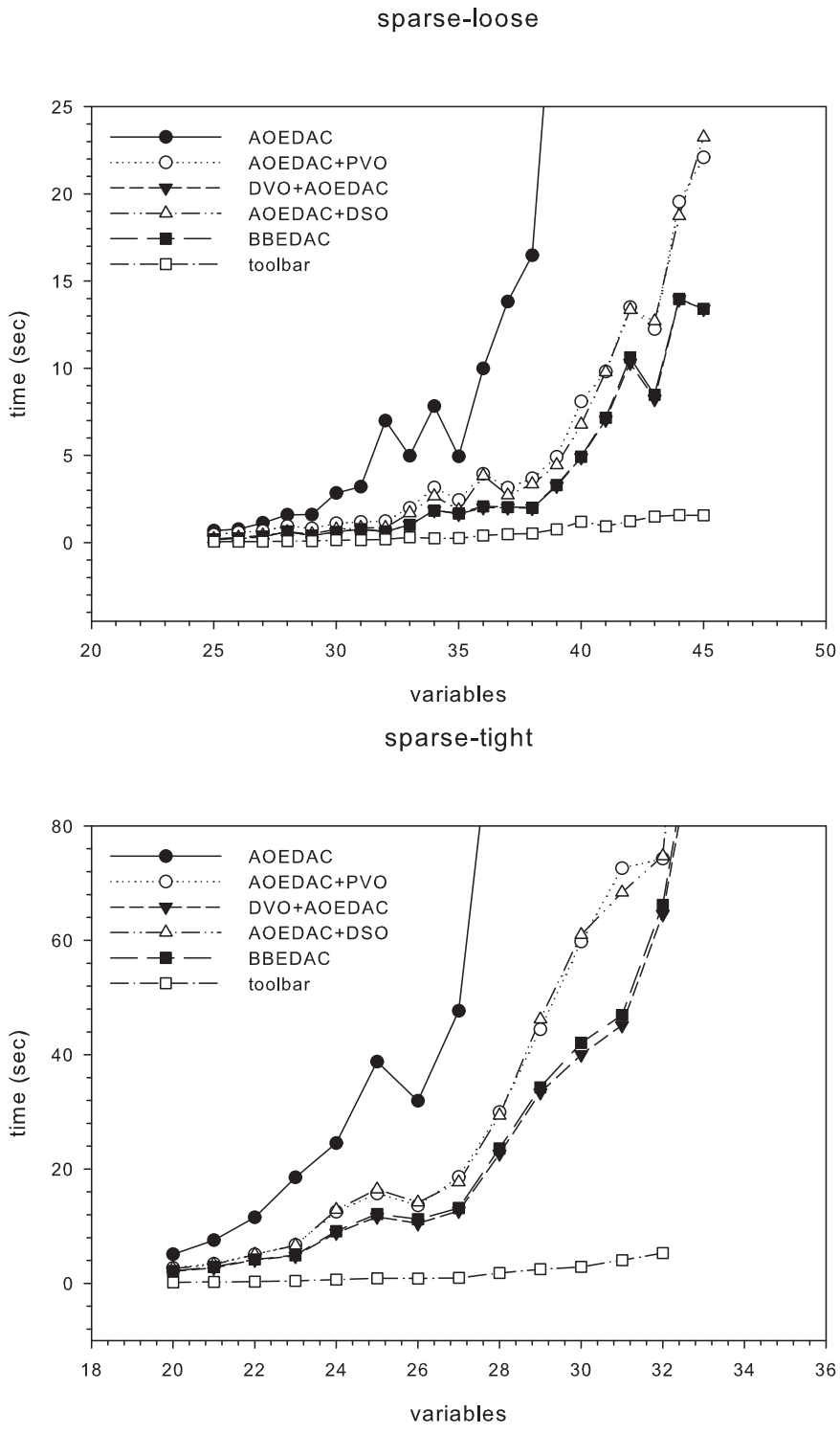


Figure 3.26: CPU time for solving **sparse random binary WCSPs**. Time limit 300 seconds.

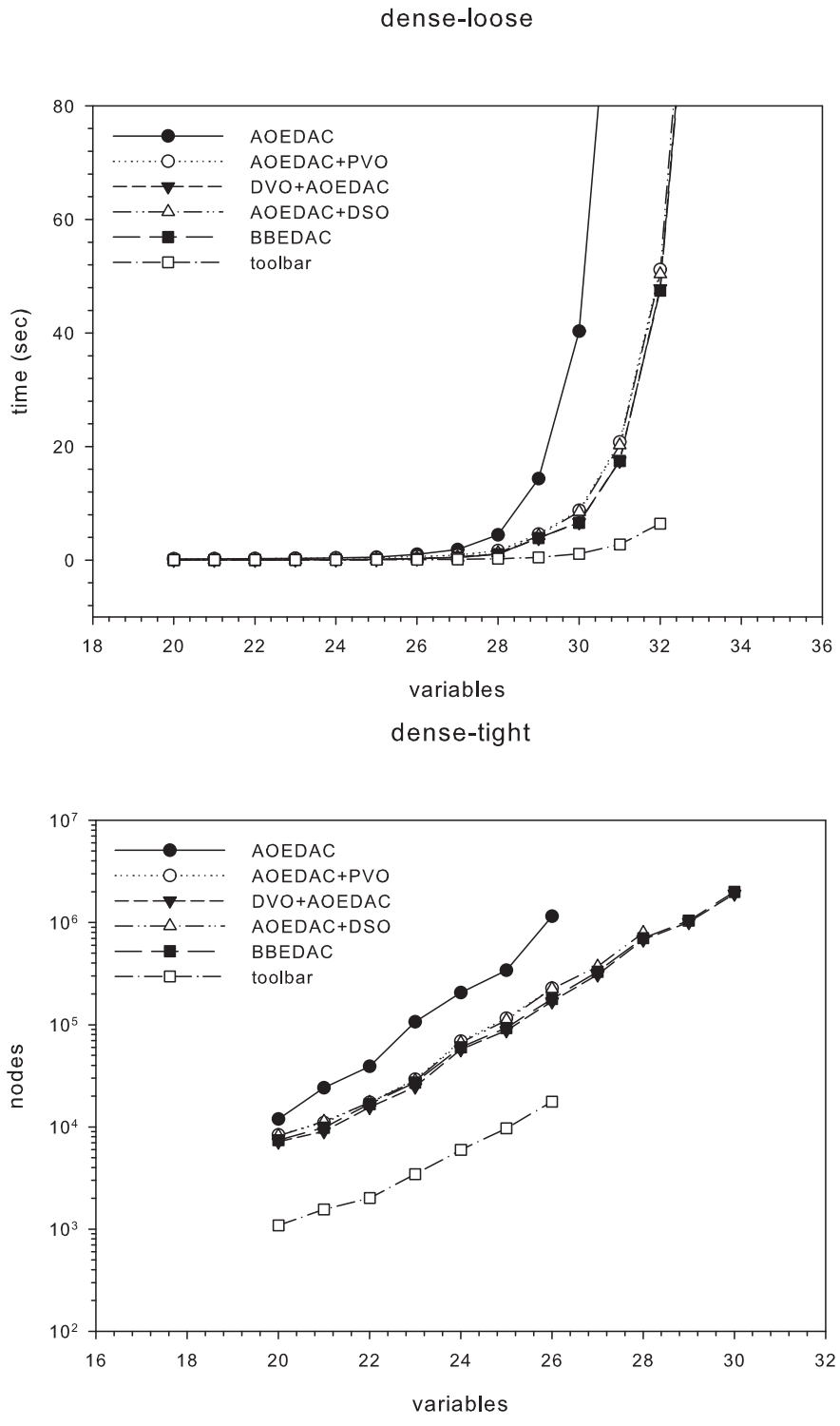


Figure 3.27: CPU time for solving **dense random binary WCSPs**. Time limit 300 seconds.

3.7 Related Work

The idea of exploiting structural properties of the problem in order to enhance the performance of search algorithms in constraint satisfaction is not new. Freuder and Quinn [48] introduced the concept of pseudo tree arrangement of a constraint graph as a way of capturing independencies between subsets of variables. Subsequently, *pseudo tree search* [48] is conducted over a pseudo tree arrangement of the problem which allows the detection of independent subproblems that are solved separately. More recently, [70] extended pseudo tree search [48] to optimization tasks in order to boost the Russian Doll search [51] for solving Weighted CSPs. Our AND/OR Branch-and-Bound algorithm is also related to the Branch-and-Bound method proposed by [62] for acyclic AND/OR graphs and game trees.

Dechter's graph-based back-jumping algorithm [29] uses a depth-first (DFS) spanning tree to extract knowledge about dependencies in the graph. The notion of DFS-based search was also used by [19] for a distributed constraint satisfaction algorithm. Bayardo and Miranker [106] reformulated the pseudo tree search algorithm in terms of back-jumping and showed that the depth of a pseudo-tree arrangement is always within a logarithmic factor off the induced width of the graph.

Recursive Conditioning (RC) [24] is based on the divide and conquer paradigm. Rather than instantiating variables to obtain a tree structured network like the cycle cutset scheme, RC instantiates variables with the purpose of breaking the network into independent subproblems, on which it can recurse using the same technique. The computation is driven by a data-structure called *dtree*, which is a full binary tree, the leaves of which correspond to the network CPTs. It can be shown that RC explores an AND/OR space [38]. A pseudo tree can be generated from the static ordering of RC dictated by the dtree. This ensures that whenever RC splits the problem into independent subproblems, the same happens in the AND/OR space.

Value Elimination [4] is a recently developed algorithm for Bayesian inference. It was already explained in [4] that, under static variable ordering, there is a strong relation be-

tween Value Elimination and Variable Elimination. Given a static ordering d for Value Elimination, it can be shown that it actually traverses an AND/OR space [38]. The pseudo tree underlying the AND/OR search space traversal by Value Elimination can be constructed as the bucket tree in reversed d. However, the traversal of the AND/OR space will be controlled by d, advancing the frontier in a hybrid depth or breadth first manner. Value Elimination is not a linear space algorithm.

Backtracking with Tree-Decomposition (BTD) [59] is a memory intensive method for solving constraint satisfaction (or optimization) problems which combines search techniques with the notion of tree decomposition. This mixed approach can in fact be viewed as searching an AND/OR search space whose backbone pseudo tree is defined by and structured along the tree decomposition. What is defined in [59] as structural goods, that is parts of the search space that would not be visited again as soon as their consistency (or optimal value) is known, corresponds precisely to the decomposition of the AND/OR space at the level of AND nodes, which root independent subproblems. The BTD algorithm is not linear space, it uses substantial caching and may be exponential in the induced width.

3.8 Conclusion to Chapter 3

The chapter investigates the impact of AND/OR search spaces perspective on solving general constraint optimization problems in graphical models. In contrast to the traditional OR search, the new AND/OR search is sensitive the problem's structure. The linear space AND/OR tree search algorithms can be exponentially better (and never worse) than the linear space OR tree search algorithms. Specifically, the size of the AND/OR search tree is exponential in the depth of the guiding pseudo tree rather than the number of variables, as in the OR case.

We introduced a general Branch-and-Bound algorithm that explores the AND/OR search tree in a depth-first manner. It can be guided by any heuristic function. We investigated

extensively the mini-bucket heuristic and showed that it can prune the search space very effectively. The mini-bucket heuristics can be either pre-compiled (static mini-buckets) or generated dynamically at each node in the search tree (dynamic mini-buckets). They are parameterized by the Mini-Bucket i-bound which allows for a controllable trade-off between heuristic strength and computational overhead. In conjunction with the mini-bucket heuristics we also explored the effectiveness of another class of heuristic lower bounds that is based on exploiting local consistency algorithms for cost functions, in the context of WCSPs.

Since variable ordering can influence dramatically the search performance, we also introduced several ordering schemes that combine the AND/OR decomposition principle with dynamic variable ordering heuristics. There are three approaches to incorporating dynamic orderings into AND/OR Branch-and-Bound search. The first one applies an independent semantic variable ordering heuristic whenever the partial order dictated by the static decomposition principle allows. The second, orthogonal approach gives priority to the semantic variable ordering heuristic and applies problem decomposition as a secondary principle. Since the structure of the problem may change dramatically during search we presented a third approach that uses a dynamic decomposition method coupled with semantic variable ordering heuristics.

We focused our empirical evaluation on two common optimization problems in graphical models: finding the MPE in Bayesian networks and solving WCSPs. Our results demonstrated conclusively that in many cases the depth-first AND/OR Branch-and-Bound algorithms guided by either mini-bucket or local consistency based heuristics improve dramatically over traditional OR Branch-and-Bound search, especially for relatively weak guiding heuristic estimates. We summarize next the most important aspects reflecting the better performance of the AND/OR algorithms, including the mini-bucket i-bound, dynamic variable orderings, constraint propagation and the quality of the guiding pseudo tree.

- **Impact of the mini-bucket i-bound.** Our results show conclusively that when enough memory is available the static mini-bucket heuristics with relatively large i-bounds are cost effective ($e.g.$, genetic linkage analysis networks from Table 3.8, Mastermind game instances from Table 3.18). However, if space is restricted, the dynamic mini-bucket heuristics, which exploit the partial assignment along the search path, appear to be the preferred choice, especially for relatively small i-bounds ($e.g.$, ISCAS'89 networks from Tables 3.17 and 3.9). This is because these heuristics are far more accurate for the same i-bound than the pre-compiled version and the savings in number of nodes explored translate into important time savings.

- **Impact of dynamic variable ordering.** Our dynamic AND/OR search approach was shown to be powerful especially when used in conjunction with local consistency based heuristics. The AND/OR Branch-and-Bound algorithms with EDAC based heuristics and dynamic variable orderings were sometimes able to outperform the Branch-and-Bound counterpart with static variable orderings by two orders of magnitude in terms of running time ($e.g.$, see for example the `503` SPOT5 network from Table 3.19).

- **Impact of determinism.** When the graphical model contains both deterministic information (hard constraints) as well as general cost functions, we demonstrated that is beneficial to exploit the computational power of the constraints explicitly, via constraint propagation. Our experiments on selected classes of deterministic Bayesian networks showed that enforcing a form of constraint propagation, called unit resolution, over the CNF encoding of the determinism present in the network was able in some cases to render the search space almost backtrack-free ($e.g.$, ISCAS'89 networks from Table 3.14). This caused a tremendous reduction in running time for the corresponding AND/OR algorithms ($e.g.$, see for example the `s953` network from Table 3.14).

- **Impact of the static variable ordering via the pseudo tree.** The performance of the AND/OR search algorithms is highly influenced by the quality of the guiding pseudo tree. We investigated two heuristics for generating small induced width/depth pseudo trees. The min-fill based pseudo trees usually have small induced width but significantly larger depth, whereas the hypergraph partitioning heuristic produces much smaller depth trees but with larger induced widths. Our experiments demonstrated that the AND/OR algorithms using mini-bucket heuristics benefit, on average, from the min-fill based pseudo trees because the guiding mini-bucket heuristic is sensitive to the induced width size which is obtained for these types of pseudo trees. In some exceptional cases however, the hypergraph partitioning based pseudo trees were able to improve significantly the search performance, especially at relatively small i-bounds ($e.g.$, see for example the `s1238` network from Figure 3.23), because in those cases the smaller depth guarantees a smaller AND/OR search tree. The picture is reversed for the AND/OR algorithms that enforce local consistency, which is not sensitive to the problem's induced width. Here, the hypergraph based trees were able to improve performance up to 3 orders of magnitude over the min-fill based trees ($e.g.$, SPOT5 networks from Figure 3.25).

Chapter 4

Memory Intensive AND/OR Search for Graphical Models

4.1 Introduction

In Chapter 3 we presented the AND/OR search space perspective for optimization tasks over graphical models. In contrast with traditional OR search, the main virtue of AND/OR search consists in exploiting independencies between variables during search. This can provide exponential speedups over traditional search methods oblivious to problem structure. The AND/OR search tree is guided by a pseudo tree that spans the primal graph. Assigning a value to a variable (also known as conditioning) is equivalent in graph terms to removing that variable (and its incident edges) from the primal graph. A partial assignment can therefore lead to the decomposition of the residual primal graph into independent components, each of which can be searched (or solved) separately. The pseudo tree captures some of these decompositions given an order of variable instantiation.

It is often the case that a search space that is a tree can become a graph if identical nodes are merged, because identical nodes root identical subspaces and correspond to identical subproblems. Some of these unifiable nodes in the AND/OR search tree can be identified based on the graph notion of *contexts*. The context of a node is a subset of the currently assigned variables that completely determines the remaining subproblem using graph information only. Consequently, algorithms that explore the search graph involve controlled

memory management that allows improving their time-performance by increasing their use of memory.

Contribution

In this chapter we improve significantly the AND/OR Branch-and-Bound tree search algorithm introduced in Chapter 3 by using *caching* schemes. Namely, we extend the algorithm to explore the context minimal AND/OR search *graph* rather than the AND/OR search tree, using a flexible caching mechanism that can adapt to memory limitations. The caching scheme, which is based on contexts, is similar to good and no-good recording and recent schemes appearing in Recursive Conditioning [24], Valued Backtracking [4] and Backtracking with Tree Decompositions [59].

Since best-first search is known to be superior among memory intensive search algorithms [40], the comparison with the best-first approach that exploits similar amounts of memory is warranted. We therefore present a new AND/OR search algorithm that explores a context minimal AND/OR search graph in a *best-first* rather than depth-first manner. Under conditions of admissibility and monotonicity of the heuristic function, best-first search is known to expand the minimal number of nodes, at the expense of using additional memory [40]. These savings in number of nodes may often translate into time savings as well.

The efficiency of the proposed memory intensive depth-first and best-first AND/OR search methods also depends on the accuracy of the guiding heuristic function, which is based on the Mini-Bucket approximation [42]. Like in Chapter 3, we continue to explore empirically the efficiency of the mini-bucket heuristics in both static and dynamic settings, as well as the interaction between the heuristic strength and the level of caching.

We apply the memory intensive depth-first and best-first AND/OR search algorithms to two common optimization problems in graphical models: finding the Most Probable Explanation (MPE) in belief networks [104] and solving Weighted CSPs [9]. We experiment with both random models and real-world benchmarks. Our results show conclusively that

the new memory intensive AND/OR search algorithms improve dramatically over competitive approaches, especially when the heuristic estimates are inaccurate and the algorithms rely primarily on search rather than on pruning based on the heuristic evaluation function.

The research presented in this chapter is based in part on [82, 84, 85].

Chapter Outline

The chapter is organized as follows. Section 4.2 provides background on the AND/OR search graph for graphical models. Sections 4.3 and 4.4 present the new depth-first and best-first AND/OR search algorithms exploring the context minimal AND/OR graph. Section 4.5 is dedicated to an extensive empirical evaluation of the proposed memory intensive search methods, while Section 4.6 provides a summary and concluding remarks.

4.2 AND/OR Search Graphs for Graphical Models

The AND/OR search tree for graphical models presented in Chapter 3 exploits problem structure during search by utilizing value assignment as a problem simplification mechanism. An *AND/OR search tree* is defined using a backbone *pseudo tree* that spans the primal graph and captures problem decomposition during search. The search tree contains alternating levels of OR and AND nodes. The OR nodes correspond to the variables while the AND nodes represent value assignments. A depth-first search algorithm traversing the AND/OR search tree is time-exponential in the depth of the guiding pseudo tree and may operate in linear space.

It is often the case that a search space that is a tree can become a graph if identical nodes are *merged*, because identical nodes root identical search subspaces and correspond to identical subproblems. Some of these nodes can be identified based on *contexts*. The transition from a search tree to a search graph in AND/OR representations also yields significant savings compared to the same transition in the original OR space. The notion of

AND/OR search *graph* was presented for general graphical models in [38]. We next give an overview of the main concepts.

First, we present the notion of *induced width of a pseudo tree of* G [38] which is necessary for bounding the size of the AND/OR search graphs. We denote by $d_{DFS}(\mathcal{T})$ a linear DFS ordering of a tree $\mathcal{T}$.

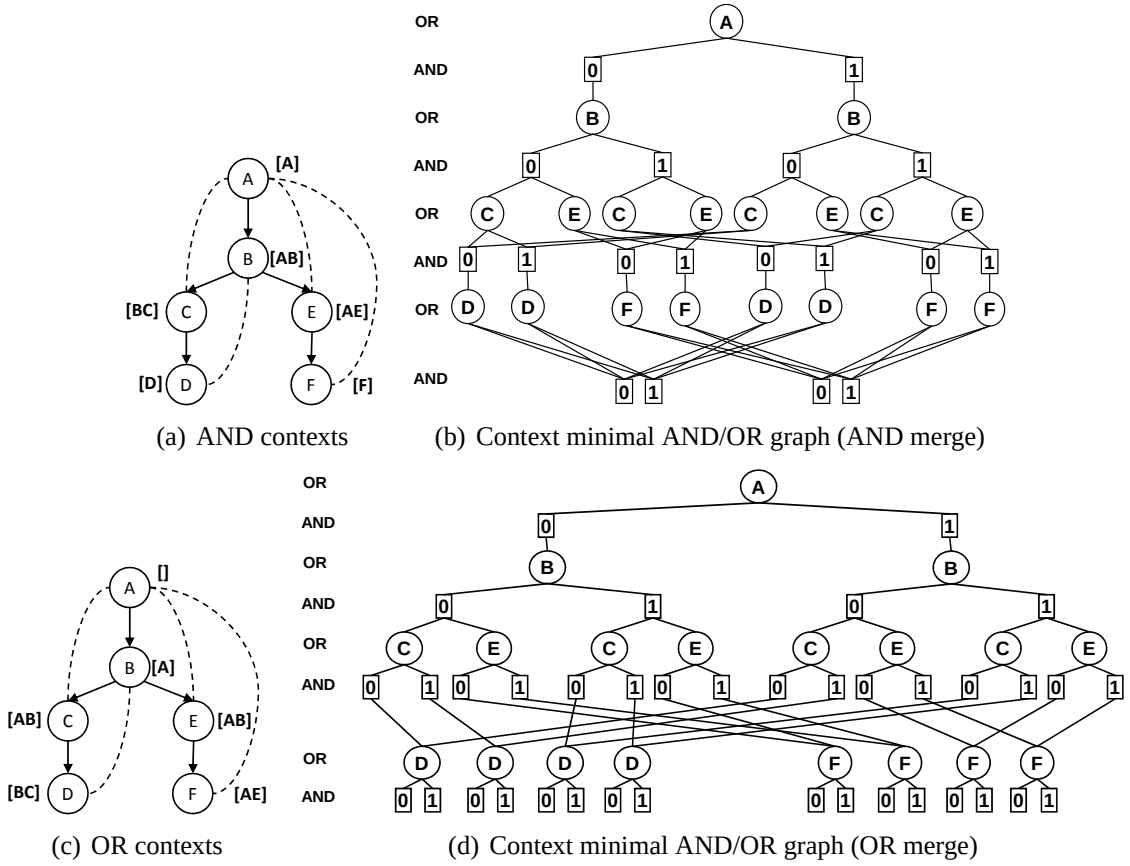
DEFINITION **38 (induced width of a pseudo tree)** *The induced width of G relative to a pseudo tree $\mathcal{T}$, $w_{\mathcal{T}}(G)$, is the induced width along $d_{DFS}(\mathcal{T})$ ordering of the extended graph of G relative to $\mathcal{T}$, denoted $G^{\mathcal{T}}$.*

We now provide definitions which allow identifying nodes that can be merged. The idea is to find a minimal set of variable assignments from the current path that will always root the same conditioned subproblem, regardless of the assignments that are not included in this minimal set. Since the path for an OR node X_i and an AND node $\langle X_i, x_i \rangle$ differ by the assignment of X_i to x_i, the minimal set of assignments that we want to identify will be different for X_i and for $\langle X_i, x_i \rangle$. The following definitions distinguish between two types of context-based caching which may yield into two different schemes. The difference may seem a bit subtle. In these definitions, ancestors and descendants are with respect to the pseudo tree $\mathcal{T}$, while connection is with respect to the primal graph G.

DEFINITION **39 (parents)** *Given a primal graph G and a pseudo tree $\mathcal{T}$ of a graphical model $\mathcal{R}$, the* parents *of an OR node X_i, denoted by pa_i or pa_{X_i}, are the ancestors of X_i that have connections in G to X_i or to descendants of X_i.*

DEFINITION **40 (parent-separators)** *Given a primal graph G and a pseudo tree $\mathcal{T}$ of a graphical model $\mathcal{R}$, the* parent-separators *of X_i (or of $\langle X_i, x_i \rangle$), denoted by pas_i or pas_{X_i}, are formed by X_i and its ancestors that have connections in G to descendants of X_i.*

It follows from these definitions that the parents of X_i, pa_i, separate in the primal graph G (and also in the extended graph $G^{\mathcal{T}}$ and in the induced extended graph $G^{\mathcal{T}^*}$) the ancestors



(a) AND contexts (b) Context minimal AND/OR graph (AND merge)

(c) OR contexts (d) Context minimal AND/OR graph (OR merge)

Figure 4.1: AND/OR search graph for graphical models.

of X_i from its descendants. Similarly, the parent-separators set of X_i, pas_i, separate the ancestors of X_i from its descendants. It is also easy to see that each variable X_i and its parents pa_i form a clique in the induced graph $G^{\mathcal{T}^*}$. The following proposition establishes the relation between pa_i and pas_i.

PROPOSITION **2 ([38])** *(1) If Y is the single child of X in $\mathcal{T}$, then $pas_X = pa_Y$. (2) If X has children $Y_1, ..., Y_k$ in $\mathcal{T}$, then $pas_X = \cup_{i=1}^{k} pa_{Y_i}$.*

THEOREM **8 (context based merge [38])** *Given $G^{\mathcal{T}^*}$, let π_{n_1} and π_{n_2} be any two partial paths in an AND/OR search graph, ending with two nodes, n_1 and n_2.*

1. *If n_1 and n_2 are AND nodes annotated by $\langle X_i, x_i \rangle$ and*

$$asgn(\pi_{n_1})[pas_{X_i}] = asgn(\pi_{n_2})[pas_{X_i}]$$

then the AND/OR search subtrees rooted by n_1 and n_2 are identical and n_1 and n_2 can be merged. $asgn(\pi_{n_i})[pas_{X_i}]$ is called the **AND context** *of n_i.*

2. *If n_1 and n_2 are OR nodes annotated by X_i and*

$$asgn(\pi_{n_1})[pa_{X_i}] = asgn(\pi_{n_2})[pa_{X_i}]$$

then the AND/OR search subtrees rooted by n_1 and n_2 are identical and n_1 and n_2 can be merged. $asgn(\pi_{n_i})[pa_{X_i}]$ is called the **OR context** *of n_i.*

DEFINITION **41 (context minimal AND/OR search graph)** *The AND/OR search graph of $\mathcal{R}$ based on the backbone pseudo tree $\mathcal{T}$ that is closed under the context-based merge operator is called* context-minimal AND/OR search graph *and is denoted by $\mathcal{G}_{\mathcal{T}}(\mathcal{R})$.*

We should note that we can in general merge nodes based both on AND and OR contexts. However, Proposition 2 shows that doing just one of them renders the other unnecessary (up to some some small constant factor). In this chapter we will be using AND context based merging.

THEOREM **9 (complexity [38])** *Given a graphical model $\mathcal{R}$, its primal graph G, and a pseudo tree $\mathcal{T}$ having induced width $w = w_{\mathcal{T}}(G)$, the size of the context minimal AND/OR search graph based on $\mathcal{T}$, $\mathcal{G}_{\mathcal{T}}(\mathcal{R})$, is $O(n \cdot k^w)$, where k bounds the domain size.*

Example 16 *Consider the example given in Figure 4.1(a). The AND contexts of each node in the pseudo tree is given in square brackets. The context minimal AND/OR search graph (based on AND merging) is given in Figure 4.1(b). Its size is far smaller than that of the AND/OR search tree from Figure 3.1(c) (16 vs. 54 AND nodes). Similarly, Figure 4.1(d) shows the context minimal AND/OR graph based on the OR contexts given in Figure 4.1(c). Its size is larger than that of graph based on AND contexts (38 vs. 16 nodes) in this case.*

Finding Good Pseudo Trees

The performance of any AND/OR search algorithm is influenced by the quality of the underlying pseudo tree. In Chapter 3 we described two heuristics for generating small induced width/depth pseudo trees. The *min-fill* heuristic extracts the pseudo tree by a depth-first traversal of the induced graph obtained by a min-fill elimination ordering [67]. The *hypergraph partitioning* heuristic constructs the pseudo tree by recursively decomposing the dual hypergraph associated with the graphical model [24]. We observed that the min-fill heuristic usually generates lower width trees, whereas the hypergraph heuristic produces much smaller depth trees. Therefore, the hypergraph based pseudo trees appear to be favorable for tree search algorithms, while the min-fill pseudo trees, which minimize the context size, are more appropriate for graph search algorithms. In the experimental section we provide an extensive evaluation detailing the impact of the pseudo tree quality on the AND/OR graph search algorithms.

4.3 AND/OR Branch-and-Bound with Caching

The depth-first *AND/OR Branch-and-Bound* algorithm, `AOBB-C`, for searching context minimal AND/OR graphs for graphical models, is described by Algorithm 8. It interleaves a forward expansion step of the current partial solution tree (`EXPAND`) with a backward propagation step (`PROPAGATE`) that updates the node values. This performance is identical to the tree-based variant from Chapter 3 and we describe it here for completeness sake.

The context based caching uses table representation. For each variable X_i, a table is reserved in memory for each possible assignment to its parent-separator set pas_i ($i.e.$, AND context). During search, each table entry records the optimal solution (both the cost and an optimal solution tree) to the subproblem below the corresponding AND node. Initially, each entry has a predefined value, in our case `NULL`. The fringe of the search is maintained by a stack called `OPEN`. The current node is denoted by n, its parent by p, and the current

Algorithm 8: AOBB-C: AND/OR Branch-and-Bound Graph Search

Input: An optimization problem $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum, min \rangle$, pseudo-tree $\mathcal{T}$ rooted at X_1, AND contexts pas_i for every variable X_i, heuristic function $h(n)$.

Output: Minimal cost solution and an optimal solution assignment.

```
1  v(s) ← ∞; ST(s) ← ∅; OPEN ← {s}                                   // Initialize search stack
2  Initialize cache tables with entries "NULL"                        // Initialize cache tables
3  while OPEN ≠ ∅ do
4      n ← top(OPEN); remove n from OPEN                              // EXPAND
5      if n is an OR node, labeled X_i then
6          foreach x_i ∈ D_i do
7              create an AND node n', labeled ⟨X_i, x_i⟩
8              v(n') ← 0; ST(n') ← ∅
9              w(n, n') ← Σ_{f∈B_T(X_i)} f(asgn(π_n))              // Compute the OR-to-AND arc weight
10             succ(n) ← succ(n) ∪ {n'}
11     else if n is an AND node, labeled ⟨X_i, x_i⟩ then
12         cached ← false; deadend ← false
13         if Cache(asgn(π_n)[pas_i]) ≠ NULL then
14             v(n) ← Cache(asgn(π_n)[pas_i]).value                   // Retrieve value
15             ST(n) ← Cache(asg(π_n)[pas_i]).assignment;             // Retrieve optimal assignment
16             cached ← true                                          // No need to expand below
17         foreach OR ancestor m of n do
18             lb ← evalPartialSolutionTree(T'_m)
19             if lb ≥ v(m) then
20                 deadend ← true
21                 break
22         if deadend == false and cached == false then
23             foreach X_j ∈ children_T(X_i) do
24                 create an OR node n' labeled X_j
25                 v(n') ← ∞; ST(n') ← ∅
26                 succ(n) ← succ(n) ∪ {n'}
27         else if deadend == true then
28             succ(p) ← succ(p) − {n}
29     Add succ(n) on top of OPEN                                     // PROPAGATE
30     while succ(n) == ∅ do
31         if n is an OR node, labeled X_i then
32             if X_i == X_1 then
33                 return (v(n), ST(n))                               // Search is complete
34             v(p) ← v(p) + v(n)                                     // Update AND node value (summation)
35             ST(p) ← ST(p) ∪ ST(n)                                  // Update solution tree below AND node
36         else if n is an AND node, labeled ⟨X_i, x_i⟩ then
37             Cache(asgn(π_n)[pas_i]).value ← v(n)                   // Save AND node value in cache
38             Cache(asgn(π_n)[pas_i]).assignment ← ST(n);             // Save optimal assignment
39             if v(p) > (w(p, n) + v(n)) then
40                 v(p) ← w(p, n) + v(n)                              // Update OR node value (minimization)
41                 ST(p) ← ST(n) ∪ {(X_i, x_i)}                       // Update solution tree below OR node
42         remove n from succ(p)
43         n ← p
```

path by π_n. The children of the current node are denoted by $succ(n)$.

Each node n in the search graph maintains its current value $v(n)$, which is updated based on the values of its children. For OR nodes, the current $v(n)$ is an upper bound on the optimal solution cost below n. Initially, $v(n)$ is set to ∞ if n is OR, and 0 if n is AND, respectively. A data structure $ST(n)$ maintains the actual best solution tree found in the subgraph rooted at n. The node based heuristic function $h(n)$ of $v(n)$ is assumed to be available to the algorithm, either retrieved from a cache or computed during search.

Since we use AND caching, before expanding the current AND node n, its cache table is checked (line 13). If the same context was encountered before, it is retrieved from the cache, and $succ(n)$ is set to the empty set, which will trigger the PROPAGATE step. The algorithm also computes the heuristic evaluation function for every partial solution subtree rooted at the OR ancestors of n along the path from the root (lines 17–21). The search below n is terminated if, for some OR ancestor m, $f(T'_m) \geq v(m)$, where $v(m)$ is the current upper bound on the optimal cost below m. The recursive computation of $f(T'_m)$ is described by Algorithm 7 from Chapter 3.

If a node is not found in cache, it is expanded in the usual way, depending on whether it is an AND or OR node (lines 5–28). If n is an OR node, labeled X_i, then its successors are AND nodes represented by the values x_i in variable X_i's domain (lines 5–10). Each OR-to-AND arc is associated with the appropriate weight (see Definition 25 in Chapter 3). Similarly, if n is an AND node, labeled $\langle X_i, x_i \rangle$, then its successors are OR nodes labeled by the child variables of X_i in $\mathcal{T}$ (lines 22–26).

The node values are updated by the PROPAGATE step (lines 30–43). It is triggered when a node value has an empty set of descendants (note that as each successor is evaluated, it is removed from the set of successors in line 42). This means that all its children have been evaluated, and their final values are already determined. If the current node is the root, then the search terminates with its value and an optimal solution tree (line 33). If n is an OR node, then its parent p is an AND node, and p updates its current value $v(p)$ by summation

with the value of n (line 34). An AND node n propagates its value to its parent p in a similar way, by minimization (lines 36–41). It also saves in cache the value and optimal solution subtree below it (lines 37–38). Finally, the current node n is set to its parent p (line 43), because n was completely evaluated. Each node in the search graph also records the current best assignment to the variables of the subproblem below it. Specifically, if n is an AND node, then $ST(n)$ is the union of the optimal trees propagated from n's OR children (line 35). Alternatively, if n is an OR node and n' is its AND child such that $n' = argmin_{m \in succ(n)}(w(n,m) + v(m))$, then $ST(n)$ is obtained from the label of n' combined with the optimal solution tree below n' (line 41). Search continues either with a *propagation* step (if conditions are met) or with an *expansion* step. Clearly,

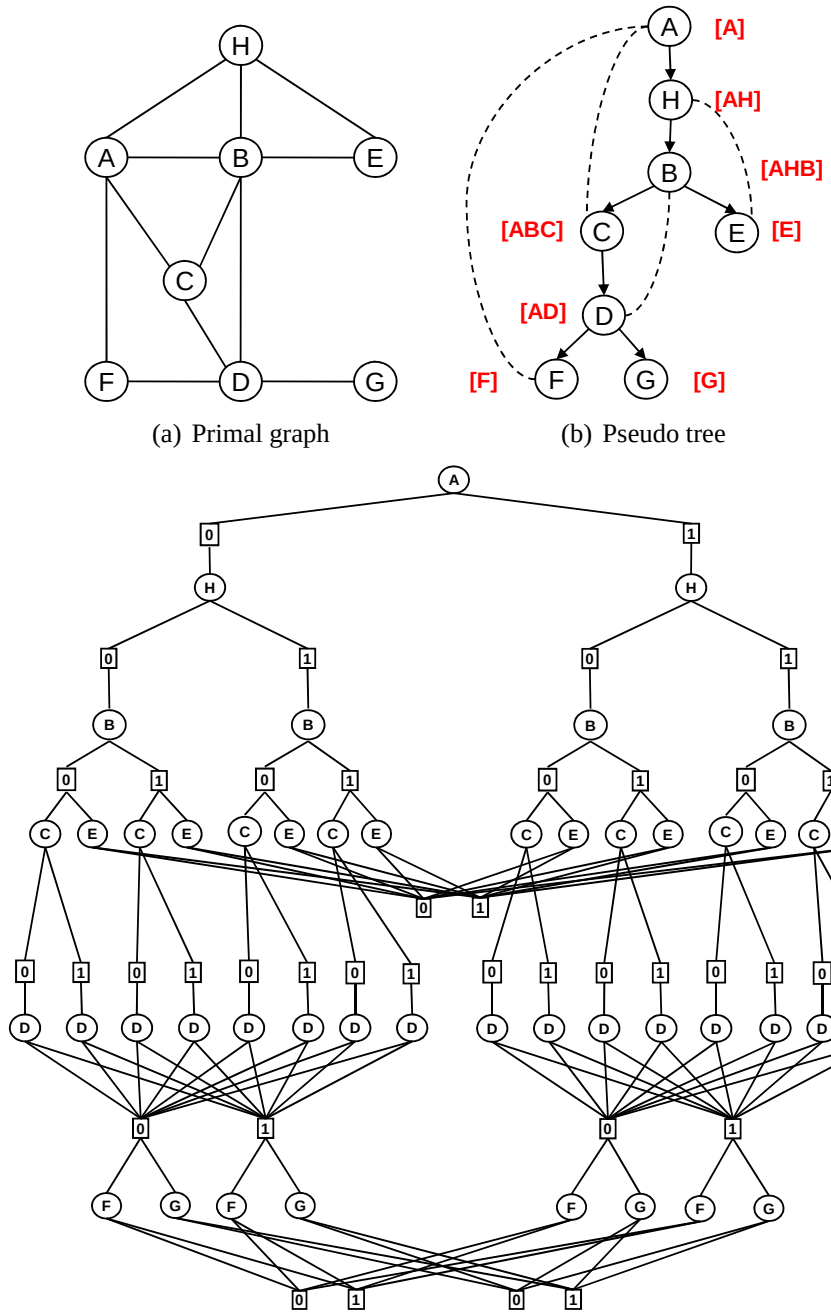
THEOREM **10 (complexity)** *AOBB-C traversing the context minimal AND/OR graph relative to a pseudo tree $\mathcal{T}$ is sound and complete. Its time and space complexity is $O(n \cdot k^{w^*})$, where w^* is the induced width of the pseudo tree and k bounds the domain size.*

Since the space required by AOBB-C can sometimes be prohibitive in practice, we next present two caching schemes that can adapt to the current memory limitations. They use a parameter called *cache bound* (or simply j-bound) to control the amount of memory used for storing unifiable nodes.

4.3.1 Naive Caching

The first scheme, called *naive caching* and denoted hereafter by AOBB-C(j), stores nodes at the variables whose context size is smaller than or equal to the cache bound j. It is easy to see that when j equals the induced width of the pseudo tree the algorithm explores the context minimal AND/OR graph via full caching.

As we mentioned earlier, a straightforward way of implementing the caching scheme is to have a *cache table* for each variable X_k recording the context. Specifically, lets assume that the context of X_k is $context(X_k) = \{X_1, ..., X_k\}$ and $|context(X_k)| \leq j$. A



(a) Primal graph (b) Pseudo tree

(c) AND/OR search graph

Figure 4.2: Illustration of naive caching used by `AOBB-C(2)`.

cache table entry corresponds to a particular instantiation $\{x_1, ..., x_k\}$ of the variables in $context(X_k)$ and records the minimal cost solution to the subproblem rooted at the AND node labeled $\langle X_k, x_k \rangle$.

However, some tables might never get cache hits. These *dead-caches* [24, 38] appear at nodes that have only one incoming arc. AOBB-C(j) needs to record only nodes that are likely to have additional incoming arcs, and these nodes can be determined by inspecting the pseudo tree. For example, if the context of a node includes that of its parent, then there is no need to store anything for that node, because it would be definitely a dead-cache.

Example 17 *Figure 4.2(c) displays the AND/OR search graph obtained with the naive caching scheme AOBB-C(2), relative to the pseudo tree given in Figure 4.2(b). Notice that there is no need to create cache tables for variables H and B, because their AND contexts include those of their respective parents in the pseudo tree, namely $context(A) \subseteq context(H)$ and $context(H) \subseteq context(B)$, respectively. Moreover, AOBB-C(2) does not cache any of the AND nodes corresponding to variable C because its corresponding cache table, which is defined on 3 variables ($e.g.$, A, B and C), cannot be stored in memory.*

4.3.2 Adaptive Caching

The second scheme, called *adaptive caching* and denoted by AOBB-AC(j), is inspired by the AND/OR cutset conditioning scheme and was first explored in [88]. It extends the naive scheme by allowing caching even at nodes with contexts larger than the given cache bound, based on *adjusted contexts*.

Consider the node X_k in the pseudo tree $\mathcal{T}$ with $context(X_k) = \{X_1, ..., X_k\}$, where $k > j$. During search, when variables $\{X_1, ..., X_{k-j}\}$ are instantiated, they can be viewed as part of a cutset. The problem rooted by X_{k-j+1} can be solved in isolation, like a subproblem in the cutset scheme, after variables $X_1, ..., X_{k-j}$ are assigned their current values in all the functions. In this subproblem, conditioned on the values $\{x_1, ..., x_{k-j}\}$, $context(X_k) = \{X_{k-j+1}, ..., X_k\}$ (also called the *adjusted context* of X_k [88]), so it can

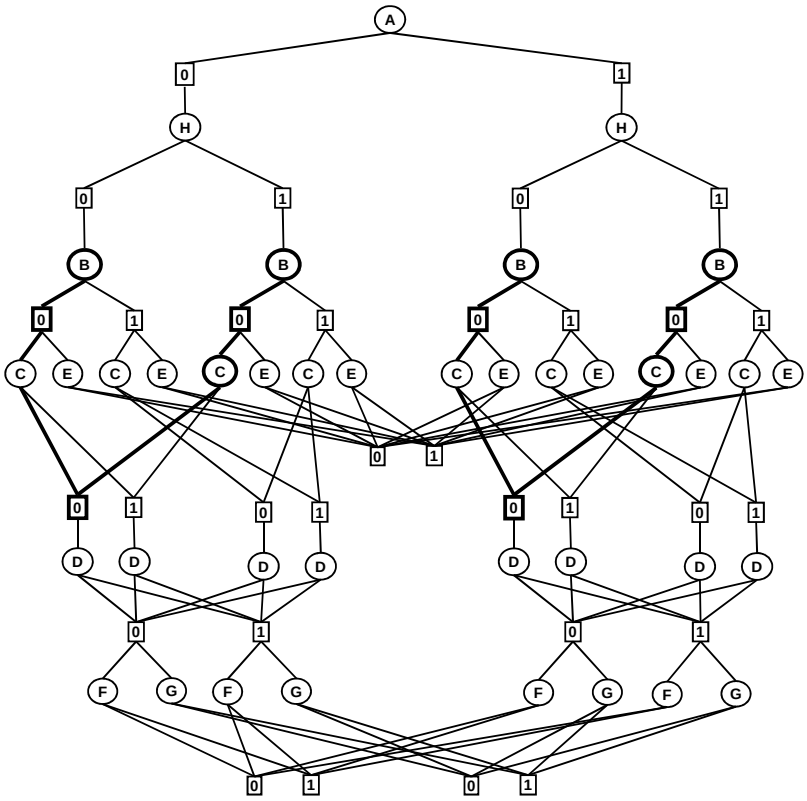


Figure 4.3: Illustration of adaptive caching used by AOBB-AC(2).

be cached within j-bounded space. However, when AOBB-AC(j) retracts to variable X_{k-j} or above, the cache table for variable X_k needs to be purged, and will be used again when a new subproblem rooted at X_{k-j+1} is solved. This caching scheme requires only a linear increase in additional memory, compared to the naive AOBB-C(j), but it has the potential of exponential time savings, as shown in [88].

Example 18 *Figure 4.3 shows the AND/OR graph traversed using the adaptive caching scheme AOBB-AC(2). In contrast to the naive scheme displayed in Figure 4.2, AOBB-AC(2) caches the AND level corresponding to variable C based on its adjusted context. The adjusted AND context of C is $\{C, B\}$ and a flag is installed at variable A, indicating that the cache table must be purged whenever A is instantiated to a different value.*

4.4 Best-First AND/OR Search

We now direct our attention to a *best-first* control strategy for traversing the context minimal AND/OR graph. The best-first search algorithm uses similar amounts of memory as the depth-first AND/OR Branch-and-Bound with full caching and therefore the comparison is warranted.

Best-first search is a search algorithm which optimizes breath-first search by expanding the node whose heuristic evaluation function is the best among all nodes encountered so far. Its main virtue is that it never expands nodes whose cost is beyond the optimal one, unlike depth-first search algorithms, and therefore is superior among memory intensive algorithms employing the same heuristic evaluation function [40].

The best-first AND/OR graph search algorithm, denoted by `AOBF-C`, that traverses the context minimal AND/OR search graph is described in Algorithm 9. It specializes Nilsson's `AO`* algorithm [97] to AND/OR search spaces for graphical models and interleaves forward expansion of the best partial solution tree (`EXPAND`) with a cost revision step (`REVISE`) that updates node values, as detailed in [97]. The explicated AND/OR search graph is maintained by a data structure called $C'_{\mathcal{T}}$, the current node is n, s is the root of the search graph and the current best partial solution subtree is denoted by T'. The children of the current node are denoted by $succ(n)$.

First, a top-down, graph-growing operation finds the best partial solution tree by tracing down through the marked arcs of the explicit AND/OR search graph $C'_{\mathcal{T}}$ (lines 3–9). These previously computed marks indicate the current best partial solution tree from each node in $C'_{\mathcal{T}}$. Before the algorithm terminates, the best partial solution tree, denoted by T', does not yet have all of its leaf nodes terminal. One of its non-terminal leaf nodes n is then expanded by generating its successors, depending on whether it is an OR or an AND node. If n is an OR node, labeled X_i, then its successors are AND nodes represented by the values x_i in variable X_i's domain (lines 11–20). Notice that when expanding an OR node, the algorithm does not generate AND children that are already present in the explicit search graph $C'_{\mathcal{T}}$, but

Algorithm 9: AOBF-C: Best-First AND/OR Graph Search

Input: An optimization problem $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum, min \rangle$, pseudo tree $\mathcal{T}$ rooted at X_1, AND contexts pas_i for every variable X_i, heuristic function $h(n)$.

Output: Minimal cost solution and an optimal solution assignment.

```
1  v(s) ← h(s); G'_T ← {s};                                              // Initialize
2  while s is not labeled SOLVED do
3      S ← {s}; T' ← {};                                                 // Create the marked PST
4      while S ≠ ∅ do
5          n ← top(S); remove n from S
6          T' ← T' ∪ {n}
7          let L be the set of marked successors of n
8          if L ≠ ∅ then
9              add L on top of S
10     let n be any nonterminal tip node of the marked T' (rooted at s)   // EXPAND
11     if n is an OR node, labeled X_i then
12         foreach x_i ∈ D_i do
13             let n' be the AND node in G'_T having context equal to pas_i
14             if n' == NULL then
15                 create an AND node n' labeled ⟨X_i, x_i⟩
16                 v(n') ← h(n')
17                 w(n, n') ← Σ_{f ∈ B_T(X_i)} f(asgn(π_n))
18                 if n' is TERMINAL then
19                     label n' as SOLVED
20             succ(n) ← succ(n) ∪ {n'}
21     else if n is an AND node, labeled ⟨X_i, x_i⟩ then
22         foreach X_j ∈ children_T(X_i) do
23             create an OR node n' labeled X_j
24             v(n') ← h(n')
25             succ(n) ← succ(n) ∪ {n'}
26     G'_T ← G'_T ∪ {succ(n)}
27     S ← {n}                                                           // REVISE
28     while S ≠ ∅ do
29         let m be a node in S such that m has no descendants in G'_T still in S; remove m from S
30         if m is an AND node, labeled ⟨X_i, x_i⟩ then
31             v(m) ← Σ_{m_j ∈ succ(m)} v(m_j)
32             mark all arcs to the successors
33             label m as SOLVED if all its children are labeled SOLVED
34         else if m is an OR node, labeled X_i then
35             v(m) = min_{m_j ∈ succ(m)} (w_(m,m_j) + v(m_j))
36             mark the arc through which this minimum is achieved
37             label m as SOLVED if the marked successor is labeled SOLVED
38         if m changes its value or m is labeled SOLVED then
39             add to S all those parents of m such that m is one of their successors through a marked arc.
40     return v(s)                                                       // Search terminates
```

rather links to them. All these identical AND nodes in $C'_{\mathcal{T}}$ are easily recognized based on their contexts. Each OR-to-AND arc is associated with the appropriate weight. Similarly, if n is an AND node, labeled $\langle X_i, x_i \rangle$, then its successors are OR nodes labeled by the child variables of X_i in $\mathcal{T}$ (lines 21–25). Moreover, a heuristic underestimate $h(n')$ of $v(n')$ is assigned to each of n's successors $n' \in succ(n)$.

The second operation in AOBF-C is a bottom-up, cost revision, arc marking, SOLVE-labeling procedure (lines 27–39). It aims at updating the evaluation function of any subtree that might be affected, and marks the best one. Starting with the node just expanded n, the procedure revises its value $v(n)$, using the newly computed values of its successors, and marks the outgoing arcs on the estimated best path to terminal nodes. This revised value is then propagated upwards in the graph. The revised value $v(n)$ is an updated lower bound on the cost of an optimal solution to the subproblem rooted at n. Only the ancestors of nodes having their values revised can possibly have their values updated, so only these need be considered (lines 38–39). If we assume the monotone restriction on h, cost revisions can only be cost increases [87, 97]. During the bottom-up step, AOBF-C labels an AND node as SOLVED if all of its OR child nodes are solved, and labels an OR node as SOLVED if its marked AND child is also solved. The algorithm terminates with the optimal solution when the root node s is labeled SOLVED.

If $h(n) \leq v(n)$, the exact cost at n, for all nodes, and if h satisfies the monotone restriction, then the algorithm AOBF-C will terminates in an optimal solution tree [87, 97]. The optimal solution tree can be obtained by tracing down from s through the marked connectors at termination and its optimal cost is equal to the value $v(s)$ of s at termination. It is possible to show that since the algorithm explores every node in the context minimal graph just once, we get:

THEOREM **11 (complexity)** *The Best-First AND/OR search algorithm traversing the context minimal AND/OR graph has time and space complexity of $O(n \cdot k^{w^*})$, where w^* is the induced width of the pseudo tree and k bounds the domain size.*

AOBB versus AOBF

We highlight next the main differences between depth-first AND/OR Branch-and-Bound (AOBB-C) and Best-First AND/OR search (AOBF-C) traversing the context minimal graph.

First, AOBF-C with the same heuristic function as AOBB-C is likely to expand the smallest number of nodes [40], but empirically this depends on how quickly AOBB-C will find an optimal solution. Second, AOBB-C can use far less memory by avoiding dead-caches for example ($e.g.$, when the search graph is a tree), while AOBF-C has to keep the explicated search graph in memory. Third, AOBB-C can be used as an anytime scheme, namely whenever interrupted, the algorithm outputs the best solution found so far, unlike AOBF-C which outputs a complete solution upon completion only. All the above points show that the relative merit of best-first versus depth-first over context minimal AND/OR search spaces cannot be determined by theory [40] and empirical evaluation is essential.

4.5 Experimental Results

In Chapter 3 we evaluated empirically AND/OR search algorithms for AND/OR trees only. We now extend this evaluation to algorithms exploring the context minimal AND/OR search graphs just described. We have conducted a number of experiments on two common optimization problems classes in graphical models: finding the Most Probable Explanation in Bayesian networks and solving Weighted CSPs. We implemented our algorithms in C++ and ran all experiments on a 2.4GHz single-core Pentium IV with 2GB of RAM, running Windows XP.

4.5.1 Overview and Methodology

Algorithms. We evaluated the following classes of memory intensive AND/OR search algorithms guided by mini-bucket heuristics:

- Depth-first AND/OR Branch-and-Bound search algorithms with full caching, using static and dynamic mini-bucket heuristics, denoted by AOBB-C+SMB(i) and AOBB-C+DMB(i), respectively.

- Best-first AND/OR search algorithms using static and dynamic mini-bucket heuristics, denoted by AOBF-C+SMB(i) and AOBF-C+DMB(i), respectively.

We compare these algorithms against the AND/OR Branch-and-Bound algorithms searching the AND/OR tree (without caching) guided by the mini-bucket heuristics, denoted by AOBB+SMB(i) and AOBB+DMB(i), which were introduced in Chapter 3. In addition, we also ran the traditional OR Branch-and-Bound search algorithms with full caching and mini-bucket heuristics, denoted by BB-C+SMB(i) and BB-C+DMB(i), respectively. The parameter i represents the mini-bucket i-bound and controls the accuracy of the heuristic.

Throughout our extensive empirical evaluation we will answer the following questions that govern the performance of the proposed algorithms:

1 The impact of graph versus tree AND/OR Branch-and-Bound search.

2 The impact of best-first versus depth-first AND/OR search.

3 The impact of the mini-bucket i-bound.

4 The impact of the cache bound j on naive and adaptive caching.

5 The impact of the pseudo tree quality on AND/OR search.

6 The impact of determinism present in the network.

7 The impact of non-trivial initial upper bounds.

Since the pre-compiled mini-bucket heuristics require a static variable ordering, the corresponding OR and AND/OR search algorithms used the variable ordering as well derived from a depth-first traversal of the guiding pseudo tree. We note however that

Benchmarks	static MBE(i) BB-C+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i)	dynamic MBE(i) BB-C+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)	min-fill vs. hypergraph pseudo trees	nave vs. adaptive caching	constraint propagation	SamIam	Superlink
Coding	√	√	-	-	-	√	-
Grids	√	√	√	√	√	√	-
Linkage	√	-	√	√	-	√	√
ISCAS'89	√	√	√	√	√	√	-
UAI'06 Dataset	√	-	√	-	-	√	-

Table 4.1: Detailed outline of the experimental evaluation for Bayesian networks.

`AOBB-C+SMB(`i`)` and `AOBB-C+DMB(`i`)` support a restricted form of dynamic variable and value ordering. Namely, there is a dynamic internal ordering of the successors of the node just expanded, before placing them onto the search stack. Specifically, in line 29 of Algorithm 8, if the current node n is AND, then the independent subproblems rooted by its OR children can be solved in decreasing order of their corresponding heuristic estimates (variable ordering). Alternatively, if n is OR, then its AND children corresponding to domain values can also be sorted in decreasing order of their heuristic estimates (value ordering).

Bayesian Networks. For the MPE task, we tested the performance of the depth-first AND/OR Branch-and-Bound and best-first AND/OR search algorithms on the following types of problems: random coding networks, grid networks, Bayesian networks derived from the ISCAS'89 digital circuits benchmark, genetic linkage analysis networks, and a subset of networks from the UAI'06 Inference Evaluation Dataset.

The detailed outline of the experimental evaluation for Bayesian networks is given in Table 4.1. We also consider an extension of the AND/OR Branch-and-Bound with caching that exploits the determinism present in the Bayesian network by constraint propagation.

For reference, we also compared with the SAMIAM version 2.3.2 software package[1]. SAMIAM is a public implementation of Recursive Conditioning [24] which can also be viewed as an AND/OR search algorithm. The algorithm uses a context-based caching

[1]Available at http://reasoning.cs.ucla.edu/samiam. We used the `batchtool 1.5` provided with the package.

Benchmarks	static MBE(i) BB-C+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i)	dynamic MBE(i) BB-C+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)	min-fill vs. hypergraph pseudo trees	nave vs. adaptive caching	AOEDAC AOEDAC+PVO DVO+AOEDAC AOEDAC+DSO	toolbar toolbar-BTD
SPOT5	√	√	√	√	√	√
ISCAS'89	√	√	√	√	√	√
Mastermind	√	-	√	√	√	√

Table 4.2: Detailed outline of the experimental evaluation for Weighted CSPs.

mechanism similar to our scheme. This version of recursive conditioning also explores a context minimal AND/OR search graph [38] and therefore its space complexity is exponential in the treewidth. Note that when we use mini-bucket heuristics with high values of i, we use space exponential in i for the heuristic calculation and storing, in addition to the space required for caching.

Weighted CSPs. For WCSPs we evaluated the performance of the AND/OR search algorithms on the following types of problems: scheduling problems from the SPOT5 benchmark, networks derived from the ISCAS'89 digital circuits and instances of the popular game of Mastermind. The outline of the experimental evaluation for WCSPs is detailed in Table 4.2.

For reference, we also report results obtained with the state-of-the-art solvers called `toolbar` [25] and `toolbar-BTD` [28][2]. `toolbar` is an OR Branch-and-Bound algorithm that maintains during search a form of soft local consistency called Existential Directional Arc Consistency (EDAC). `toolbar-BTD` extends the *Backtracking with Tree Decomposition* (BTD) algorithm [59] and computes the guiding heuristic information as well by enforcing EDAC during search. It can be shown that BTD explores a context minimal AND/OR search graph, relative to a pseudo tree corresponding to the given tree decomposition [38]. In addition, we also ran the depth-first AND/OR Branch-and-Bound tree search algorithms with EDAC heuristics and dynamic variable orderings described in Chapter 3: `AOEDAC+PVO` using partial variable orderings, `DVO+AOEDAC` using full dynamic variable

[2] Available at: *http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP*

ordering, and AOEDAC+DSO using dynamic separator orderings, respectively.

The dynamic variable ordering heuristic used by the OR and AND/OR Branch-and-Bound algorithms with EDAC heuristics was the *min-dom/ddeg* heuristic, which selects the variable with the smallest ratio of the domain size divided by the future degree. Ties were broken lexicographically.

Measures of Performance. We report the CPU time in seconds and the number of nodes visited, required for proving optimality. We also specify the number of variables (n), number of evidence variables (e), maximum domain size (k), the depth of the pseudo trees (h) and the induced width of the graphs (w^*) obtained for the test instances. When evidence is asserted in the network, w^* and h are computed after the evidence nodes were removed from the graph. We also report the time required by the Mini-Bucket algorithm MBE(i) to pre-compile the heuristic information. The pseudo trees that guide the AND/OR search algorithms were generated using the min-fill and hypergraph partitioning heuristics. In our experiments we ran the min-fill heuristic just once and broke the ties lexicographically. Since the hypergraph partitioning heuristic uses a non-deterministic algorithm, the runtime of the AND/OR search algorithms guided by the resulting pseudo trees may vary significantly from one run to the next. Therefore, we picked the pseudo tree with the smallest depth out of 10 independent runs (unless otherwise specified). The best performance points are highlighted. In each table, "-" denotes that the respective algorithm exceeded the time limit. Similarly, "out" indicates that the 2GB memory limit was exceeded.

4.5.2 Results for Empirical Evaluation of Bayesian Networks

Our results reported in Chapter 3 demonstrateded conclusively that the AND/OR Branch-and-Bound tree search algorithms with pre-compiled mini-bucket heuristics were the best performing algorithms on this domain. The diference between AOBB+SMB(i) and the OR tree search counterpart BB+SMB(i) was more pronounced at relatively small i-bounds

(corresponding to relatively weak heuristic estimates) and added up to 2 orders of magnitude in terms of both running time and size of the search space explored. For larger i-bounds, when the heuristic estimates are strong enough to prune the search space substantially, the diference between AND/OR and OR Branch-and-Bound decreased. We also showed that AOBB+SMB(i) was in many cases able to outperform dramatically the current state-of-the-art solvers for Bayesian networks such as SAMIAM as well SUPERLINK (for genetic linkage analysis). The AND/OR Branch-and-Bound with dynamic mini-bucket heuristics AOBB+DMB(i) proved competitive only for relatively small i-bounds due to the relatively reduced computational overhead. In this section we continue the empirical evaluation, focusing on memory intensive depth-first and best-first AND/OR search algorithms guided by mini-bucket heuristics.

Coding Networks

We experimented with random coding networks from the class of *linear block codes* described in Chapter 3. They can be represented as 4-layer belief networks with K nodes in each layer ($i.e.$, the number of input bits). The second and third layers correspond to input information bits and parity check bits respectively. Each parity check bit represents an XOR function of the input bits. The first and last layers correspond to transmitted information and parity check bits respectively. Input information and parity check nodes are binary, while the output nodes are real-valued. Given a number of input bits K, number of parents P for each XOR bit, and channel noise variance σ^2, a coding network structure is generated by randomly picking parents for each XOR node. Then we simulate an input signal by assuming a uniform random distribution of information bits, compute the corresponding values of the parity check bits, and generate an assignment to the output nodes by adding Gaussian noise to each information and parity check bit. The decoding algorithm takes as input the coding network and the observed real-valued output assignment and recovers the original input bit-vector by computing an MPE assignment.

minfill pseudo tree												
(K, N)	(w*, h)	SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=4		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=8		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=16		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20	
			time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
(64, 128) $\sigma^2 = 0.22$	(27, 40)	out	0.02		0.02		0.07		0.68		8.33	
			-	-	16.55	174,205	0.09	148	0.72	130	8.36	130
			287.10	5,052,010	6.58	119,289	0.08	152	0.68	129	8.34	129
			250.81	3,600,530	4.25	63,171	0.08	147	0.71	129	8.41	129
			0.04	157	**0.04**	129	0.09	128	0.72	128	8.45	128
(64, 128) $\sigma^2 = 0.36$	(27, 40)	out	0.02		0.02		0.07		0.68		8.32	
			-	-	76.38	807,319	0.99	10,688	0.81	1,189	8.41	158
			277.41	5,250,380	47.80	834,680	1.23	22,406	0.84	3,096	8.33	160
			250.32	3,907,000	35.52	518,125	0.79	12,236	0.81	1,850	8.39	148
			3.94	17,801	0.15	829	**0.12**	363	0.72	162	8.41	133
(128, 256) $\sigma^2 = 0.22$	(53, 71)	out	0.05		0.06		0.18		1.80		25.65	
			-	-	256.23	1,766,930	30.57	213,184	3.30	11,073	25.88	1,656
			-	-	229.02	3,227,110	16.67	206,004	3.51	22,644	25.87	3,081
			-	-	218.58	2,206,490	11.75	116,977	3.03	12,880	25.72	2,109
			0.14	375	**0.11**	266	0.23	262	1.90	257	25.01	258
(128, 256) $\sigma^2 = 0.36$	(53, 71)	out	0.05		0.06		0.18		1.80		25.39	
			-	-	-	-	264.57	1,732,960	202.84	1,426,730	97.98	603,342
			-	-	291.61	4,309,160	240.74	3,409,580	188.44	2,617,880	110.89	1,137,120
			-	-	290.12	2,951,230	235.08	2,312,080	178.90	1,816,940	100.32	781,438
			out		66.98	260,350	19.18	88,692	**7.23**	26,499	28.01	18,357
(K, N)	(w*, h)	SamIam	BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=4		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=8		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=12		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=16		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=20	
			time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
(64, 128) $\sigma^2 = 0.22$	(27, 40)	out	22.46	9,331	0.41	183	1.41	130	12.80	130	122.67	130
			23.62	20,008	0.35	185	1.37	129	12.77	129	121.12	129
			21.26	13,971	0.34	176	1.36	129	12.62	129	120.81	129
			0.19	129	0.37	128	2.15	128	19.98	128	192.66	128
(64, 128) $\sigma^2 = 0.36$	(27, 40)	out	46.66	18,781	5.12	1,204	5.58	432	15.47	162	123.57	144
			48.71	44,734	5.17	1,864	5.53	512	15.53	164	122.90	144
			44.20	29,191	4.91	1,323	5.41	399	15.33	155	122.27	138
			1.96	446	**0.82**	160	2.71	132	20.50	128	191.08	128
(128, 256) $\sigma^2 = 0.22$	(53, 71)	out	195.84	39,109	48.49	3,684	17.48	482	130.41	379	-	-
			195.82	121,822	48.17	9,391	17.15	500	129.38	388	-	-
			193.30	68,571	48.06	5,241	16.88	420	128.23	355	-	-
			0.75	260	1.58	256	11.18	256	131.50	256	-	-
(128, 256) $\sigma^2 = 0.36$	(53, 71)	out	288.97	62,749	229.55	19,776	234.08	4,402	276.95	804	-	-
			289.09	223,938	229.91	46,768	233.96	7,947	276.31	953	-	-
			288.79	121,278	229.09	27,362	233.72	4,662	276.87	649	-	-
			202.41	16,041	**70.68**	2,260	163.78	709	282.36	136	-	-

Table 4.3: CPU time and nodes visited for solving **random coding networks** using **static and dynamic mini-bucket heuristics** as well as min-fill based pseudo trees. Time limit 5 minutes. The top four horizontal blocks show the results for static mini-bucket heuristics, while the bottom four blocks show the dynamic mini-bucket heuristics.

Table 4.3 shows the results for solving two classes of random coding networks with $K = 64$ and $K = 128$ input bits, using static and dynamic mini-bucket heuristics. The number of parents for each XOR bit was $P = 4$ and we chose the channel noise variance $\sigma^2 \in \{0.22, 0.36\}$. For each value combination of the parameters we generated 20 random instances. The guiding pseudo trees were generated using the min-fill heuristic. The top four horizontal blocks show the results for static mini-bucket heuristics, while the bottom four ones correspond to dynamic mini-buckets heuristics. The columns are indexed by the

mini-bucket i-bound, which we varied between 4 and 20.

Tree vs. graph AOBB. When comparing the tree versus the graph AND/OR Branch-and-Bound algorithms we see that AOBB-C+SMB(i) is slightly better than AOBB+SMB(i). We observe a similar picture when using dynamic mini-buckets as well. This indicates that, on this domain, most of the cache entries were actually dead, namely the context minimal AND/OR graph explored was very close to a tree. Notice also that SAMIAM was not able to solve any of these problem instances due to the memory limit.

AOBF vs. AOBB. When comparing the best-first versus the depth-first algorithms using static mini-bucket heuristics, we see that AOBF-C+SMB(i) is better than AOBB-C+SMB(i) for relatively small i-bounds ($i.e.$, $i \in \{4, 8\}$) which generate relatively weak heuristic estimates. For instance, on class $\langle K = 64, P = 4, \sigma^2 = 0.22 \rangle$, best-first search AOBF-C+SMB(4) is 4 orders of magnitude faster than AOBB-C+SMB(4). As the i-bound increases and the heuristics become more accurate, the difference between Branch-and-Bound and best-first search decreases, because Branch-and-Bound finds close to optimal solutions fast, and therefore will not explore solutions whose cost is below the optimum, like best-first search. When looking at the algorithms using dynamic mini-bucket heuristics, we notice that AOBF-C+DMB(i) is again far better than AOBB-C+DMB(i) for smaller i-bounds.

Static vs. dynamic mini-bucket heuristics. When comparing the static versus dynamic mini-bucket heuristic we see that the latter is competitive only for relatively small i-bounds ($i.e.$, $i \in \{4, 8\}$). At higher levels of the i-bound, the accuracy of the dynamic heuristic does not outweigh its computational overhead.

Figure 4.4 plots the average running time and number of nodes visited by the depth-first and best-first AND/OR search algorithms with mini-bucket heuristics, as a function of the mini-bucket i-bound, on the random coding networks with parameters $(K = 64, P =$

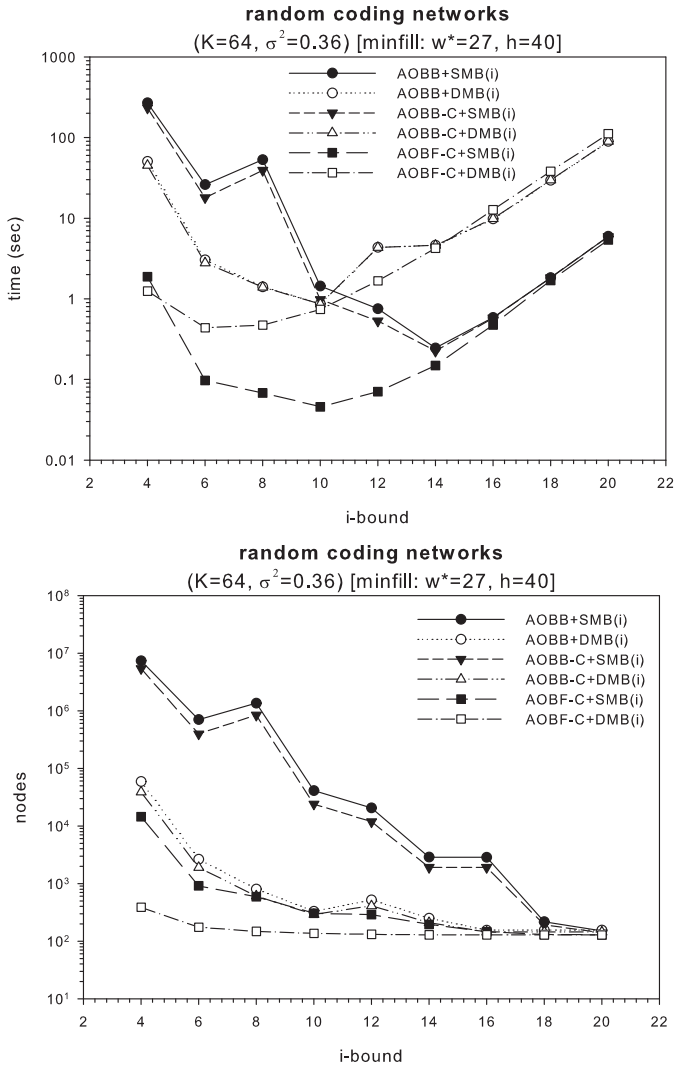


Figure 4.4: The impact of static and dynamic mini-bucket heuristics for solving the **random coding networks** with parameters $(K = 64, \sigma^2 = 0.36)$ from Table 4.3.

$4, \sigma^2 = 0.36)$ ($i.e.$, corresponding to the second and fifth horizontal blocks in Table 4.3). It shows explicitly how the performance of Branch-and-Bound and best-first search changes with the mini-bucket strength for both heuristics. Focusing for example on best-first search, we see that i-bound of 6 is most cost effective for dynamic mini-buckets, while i-bound of 10 yields best performance for static mini-buckets. We also see clearly that the dynamic mini-bucket heuristic is more accurate yielding smaller search spaces. It also demonstrates that the dynamic mini-bucket heuristics are cost effective at relatively small i-bounds, whereas the pre-compiled version is more powerful for larger i-bounds.

minfill pseudo tree											
grid (w*, h) (n, e)	SamIam v. 2.3.2	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=8		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=10		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=16	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.02		0.03		0.03		0.06		0.06	
90-10-1		0.23	3,297	0.06	373	**0.05**	102	0.06	102	0.06	102
(13, 39)	0.13	0.33	8,080	0.11	2,052	**0.05**	101	0.06	101	0.06	101
(100, 0)		0.14	2,638	0.06	819	**0.05**	101	0.06	101	0.06	101
		0.27	2,012	0.11	661	**0.05**	100	0.06	100	0.06	100
		0.03		0.03		0.08		0.14		0.44	
90-14-1		126.69	1,233,891	121.00	1,317,992	1.52	16,547	0.42	2,770	0.61	1,450
(22, 66)	11.97	8.00	130,619	6.59	100,696	1.06	17,479	0.33	3,321	0.61	2,938
(196, 0)		4.22	55,120	3.66	48,513	0.45	5,585	**0.23**	1,361	0.53	1,210
		3.20	18,796	2.70	15,764	0.55	2,899	0.30	898	0.63	857
		0.05		0.05		0.11		0.31		0.63	
90-16-1		-	-	-	-	40.05	345,255	2.38	16,942	1.23	5,327
(24, 82)	147.19	666.68	10,104,350	173.49	2,600,690	14.36	193,440	2.97	39,825	2.08	23,421
(256, 0)		209.60	2,695,249	35.45	441,364	4.23	50,481	1.19	11,029	**0.95**	4,810
		25.70	126,861	10.59	54,796	4.47	22,993	1.42	6,015	1.22	3,067

		i=12		i=14		i=16		i=18		i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.28		0.64		1.69		4.60		19.14	
90-24-1		-	-	-	-	-	-	-	-	-	-
(33, 111)	out	-	-	2338.67	24,117,151	1548.09	18,238,983	138.67	1,413,764	146.85	1,308,009
(576, 20)		-	-	1273.09	9,047,518	596.27	4,923,760	70.42	473,675	74.99	412,291
		out		21.94	75,637	10.59	33,770	**6.06**	5,144	23.80	17,291
		0.33		0.72		2.14		7.09		22.02	
90-26-1		-	-	-	-	395.67	1,635,447	-	-	67.09	277,685
(36, 113)	out	311.89	2,903,489	369.49	3,205,257	8.42	59,055	22.99	165,182	22.56	5,777
(676, 40)		146.97	878,874	152.80	962,484	4.36	15,632	12.92	46,489	22.13	2,242
		19.06	65,271	24.39	79,619	**4.27**	7,190	8.05	3,777	22.44	1,435
		0.47		0.98		2.77		7.98		30.44	
90-30-1		-	-	-	-	-	-	-	-	-	-
(43, 150)	out	1131.07	9,445,224	386.27	3,324,942	350.28	3,039,966	149.69	1,358,569	97.09	485,300
(900, 60)		652.15	3,882,300	165.74	1,070,823	155.20	956,837	40.14	212,963	59.28	174,715
		158.97	534,385	46.73	157,187	47.27	154,496	**21.06**	45,201	57.97	100,800
		0.63		1.25		3.72		11.66		40.00	
90-34-1		-	-	-	-	-	-	-	-	-	-
(45, 153)	out	-	-	-	-	-	-	-	-	478.10	1,549,829
(1154, 80)		-	-	-	-	-	-	-	-	369.36	823,604
		out		out		243.63	596,978	270.88	667,013	**71.19**	67,611
		0.78		1.67		4.20		12.36		43.69	
90-38-1		-	-	-	-	-	-	-	-	-	-
(47, 163)	out	2032.33	6,835,745	-	-	807.38	2,850,393	568.69	2,079,146	369.31	1,038,065
(1444, 120)		969.02	2,623,971	1753.10	3,794,053	203.67	614,868	165.45	488,873	113.06	214,919
		101.69	174,786	103.80	146,237	54.00	95,511	**53.44**	78,431	73.10	59,856

Table 4.4: CPU time and nodes visited for solving **grid networks** using **static mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour. Top part of the table shows results for i-bounds between 8 and 16, while the bottom part shows i-bounds between 12 and 20.

We addressed so far the impact of tree versus graph AND/OR search, the impact of the mini-bucket i-bound and best-first versus depth-first search regimes. In the remainder we will also investigate the impact of the level of caching, the impact of pseudo tree quality, the impact of determinism present in the network, as well as the anytime behavior of AND/OR Branch-and-Bound and the impact of good initial bounds.

minfill pseudo tree										
grid	BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i)	
(w*, h)										
(n, e)	i=8		i=10		i=12		i=14		i=16	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
90-10-1	0.66	303	0.47	197	0.33	102	0.41	102	0.38	102
(13, 39)	0.31	344	0.28	241	0.25	101	0.30	101	0.28	101
(100, 0)	0.28	235	0.25	170	**0.23**	101	0.28	101	0.30	101
	0.39	135	0.36	115	0.36	100	0.41	100	0.41	100
90-14-1	128.92	16,176	37.34	2,590	7.44	340	8.61	211	11.72	199
(22, 66)	56.66	31,476	23.61	4,137	4.69	397	7.25	211	10.19	199
(196, 0)	46.94	7,641	22.72	1,996	**4.67**	281	7.20	211	10.19	199
	54.09	4,007	12.84	462	6.83	221	11.94	211	16.05	199
90-16-1	639.91	42,786	388.47	12,563	112.44	1,913	103.14	1,017	39.16	262
(24, 82)	975.58	462,180	296.76	47,121	70.81	3,227	50.36	719	25.03	260
(256, 0)	382.78	44,949	245.50	11,855	65.41	1,430	48.61	525	**24.52**	260
	194.08	11,453	252.99	6,622	94.88	1,061	75.41	413	38.46	258
	i=12		i=14		i=16		i=18		i=20	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
90-24-1	-	-	-	-	2586.38	3,243	1724.68	700	2368.83	601
(33, 111)	-	-	-	-	1367.38	2,739	1979.42	1,228	1696.56	598
(576, 20)	-	-	-	-	**781.21**	1,058	1211.99	788	1693.00	598
	3456.77	11,818	1834.71	2,728	1153.48	855	1871.03	759	2573.08	591
90-26-1	-	-	-	-	-	-	-	-	-	-
(36, 113)	-	-	-	-	1514.18	2,545	2889.49	1,191	-	-
(676, 40)	2801.39	35,640	2593.74	10,216	**892.88**	1,178	1698.70	861	2647.60	687
	1262.76	5,392	1737.01	2,585	1347.54	1,049	2587.10	828	-	-
90-30-1	-	-	-	-	-	-	-	-	-	-
(43, 150)	-	-	-	-	-	-	-	-	-	-
(900, 60)	-	-	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-	-	-
90-34-1	-	-	-	-	-	-	-	-	-	-
(45, 153)	-	-	-	-	-	-	-	-	-	-
(1154, 80)	-	-	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-	-	-
90-38-1	-	-	-	-	-	-	-	-	-	-
(47, 163)	-	-	-	-	-	-	-	-	-	-
(1444, 120)	-	-	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-	-	-

Table 4.5: CPU time and nodes visited for solving **grid networks** using **dynamic mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour. Top part of the table shows results for i-bounds between 8 and 16, while the bottom part shows i-bounds between 12 and 20.

Random Grid Networks

Tables 4.4 and 4.5 show detailed results for experiments with 8 grids of increasing difficulty from Chapter 3, using static and dynamic mini-bucket heuristics. The columns are indexed by the mini-bucket i-bound. We varied the mini-bucket i-bound between 8 and 16 for the first 3 grids, and between 12 and 20 for the remaining ones. For each instance we ran a single MPE query with e nodes picked randomly and instantiated as evidence. The guiding pseudo trees were generated using the min-fill heuristic.

Tree vs. graph AOBB. First, we observe that `AOBB-C+SMB(`i`)` using full caching improves significantly over the tree version of the algorithm, especially for relatively small i-bounds which generate relatively weak heuristic estimates. For example, on the `90-16-1`

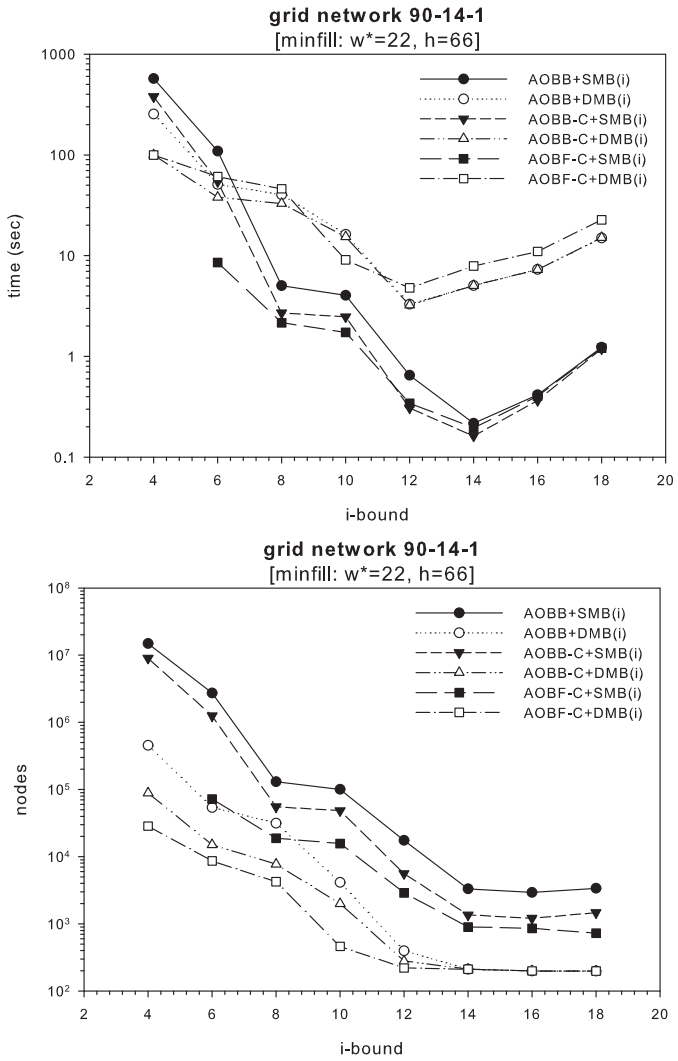


Figure 4.5: The impact of static and dynamic mini-bucket heuristics for solving the **`90-14-1` grid network** from Tables 4.4 and 4.5, respectively. We show the CPU time in seconds (top) and the number of nodes visited (bottom).

grid, AOBB-C+SMB(8) is 3 times faster than AOBB+SMB(8) and explores a search space 5 times smaller. Notice also the significant additional reduction produced by the best-first search algorithm AOBF-C+SMB(8). While overall AOBF-C+SMB(i) is superior to AOBB-C+SMB(i) with the same i-bound, the best performance on this network is obtained by AOBB-C+SMB(16). The algorithm is 2 times faster than the cache-less AOBB+SMB(16), and 155 times faster than SAMIAM, respectively. When looking at the algorithms using dynamic mini-bucket heuristics (Table 4.5) we observe a similar pattern, namely the graph search AND/OR Branch-and-Bound algorithm improves sometimes significantly over the tree search one. For instance, on the 90-24-1 grid, AOBB-C+DMB(16) is about 2 times faster than AOBB+DMB(16). Notice also that the AND/OR algorithms with dynamic mini-buckets could not solve the last 3 test instances due to exceeding the time limit. The OR Branch-and-Bound search algorithms with caching BB-C+SMB(i) (resp. BB-C+DMB(i)) are inferior to the AND/OR Branch-and-Bound graph search, especially on the harder instances ($e.g.$, 90-30-1).

AOBF vs. AOBB. When comparing further the best-first and depth-first graph search algorithms, we notice again the superiority of AOBF-C+SMB(i) over AOBB-C+SMB(i), especially for relatively weak heuristic estimates which are generated at relatively small i-bounds. For example, on the 90-38-1 grid, one of the hardest instances, best-first search with the smallest reported i-bound ($i = 12$) is 9 times faster than AOBB-C+SMB(12) and visits 15 times less nodes. The difference between best-first and depth-first search is not too prominent when using dynamic mini-bucket heuristics. This is because these heuristics are far more accurate than the pre-compiled ones and the savings in number of nodes explored by best-first search do not translate into additional time savings as well.

Static vs. dynamic mini-bucket heuristics. When comparing the static versus dynamic mini-bucket heuristics, we see that the former are more powerful for relatively large i-bounds, whereas the latter are cost effective only for relatively small i-bounds. Figure 4.5

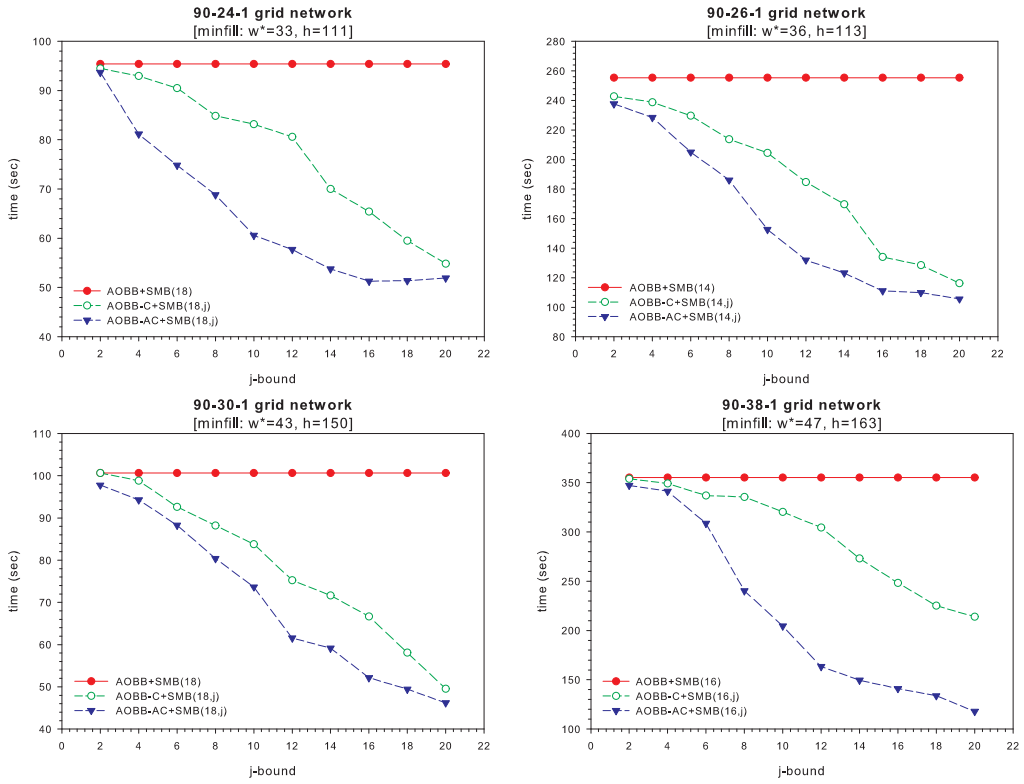


Figure 4.6: Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **grid networks**. Shown is the CPU time in seconds.

shows the CPU time and size of the search space explored by the AND/OR algorithms with mini-bucket heuristics, as a function of the i-bound, on the `90-14-1` grid from Tables 4.4 and 4.5, respectively. Focusing on `AOBB-C+SMB(`i`)`, for example, we see that its running time, as a function of i, forms a U-shaped curve. At first ($i = 4$) it is high, then as the i-bound increases the total time decreases (when $i = 14$ the time is 0.23), but then as i increases further the time starts to increase again because the pre-processing time of the mini-bucket heuristic outweighs the search time. The same behavior can be observed in the case of dynamic mini-buckets as well.

Impact of the level of caching. Figure 4.6 compares the naive (`AOBB-C+SMB(`i,j`)`) and adaptive (`AOBB-AC+SMB(`i,j`)`) caching schemes, in terms of CPU time, on 4 grid networks from Table 4.4 using AND/OR Branch-and-Bound search with static mini-bucket heuristics. In each test case we chose a relatively small mini-bucket i-bound and varied

the cache bound j (the X axis) from 2 to 20. We see that adaptive caching improves significantly over the naive scheme especially for relatively small j-bounds. This may be important because small j-bounds mean restricted space. At large j-bounds the two schemes are identical and approach the full-caching scheme.

Impact of the pseudo tree. Figure 4.7 plots the runtime distribution of AOBB-C+SMB(i) and AOBF-C+SMB(i) using hypergraph based pseudo trees. For each reported i-bound, the corresponding data point and error bar represent the average as well as the minimum and maximum run times obtained over 20 independent runs with a 30 minute time limit. The hypergraph based pseudo trees, which have far smaller depths, are sometimes able to improve the performance of AOBB-C+SMB(i), especially for relatively small i-bounds ($e.g.$, 90-24-1). For larger i-bounds, the pre-compiled mini-bucket heuristic benefits from the small induced widths obtained with the min-fill ordering. Therefore, AOBB-C+SMB(i) using min-fill based pseudo trees is generally faster. We also see that on average AOBF-C+SMB(i) is faster when it is guided by min-fill rather than hypergraph based pseudo trees. This verifies our hypothesis that memory intensive algorithms exploring the AND/OR graph are more sensitive to the context size (which is smaller for min-fill orderings), rather than the depth of the pseudo tree.

Memory usage of AND/OR graph search. Figure 4.8 displays the memory usage of AOBB-C+SMB(i) and AOBF-C+SMB(i) on grids 90-30-1 and 90-38-1, respectively. We see that for relatively small i-bounds the memory requirements of AOBF-C+SMB(i) are significantly larger than those of AOBB-C+SMB(i). This is because AOBF-C+SMB(i) has to keep in memory the entire search space explored, unlike AOBB-C+SMB(i) which can save space by avoiding dead-caches for example.

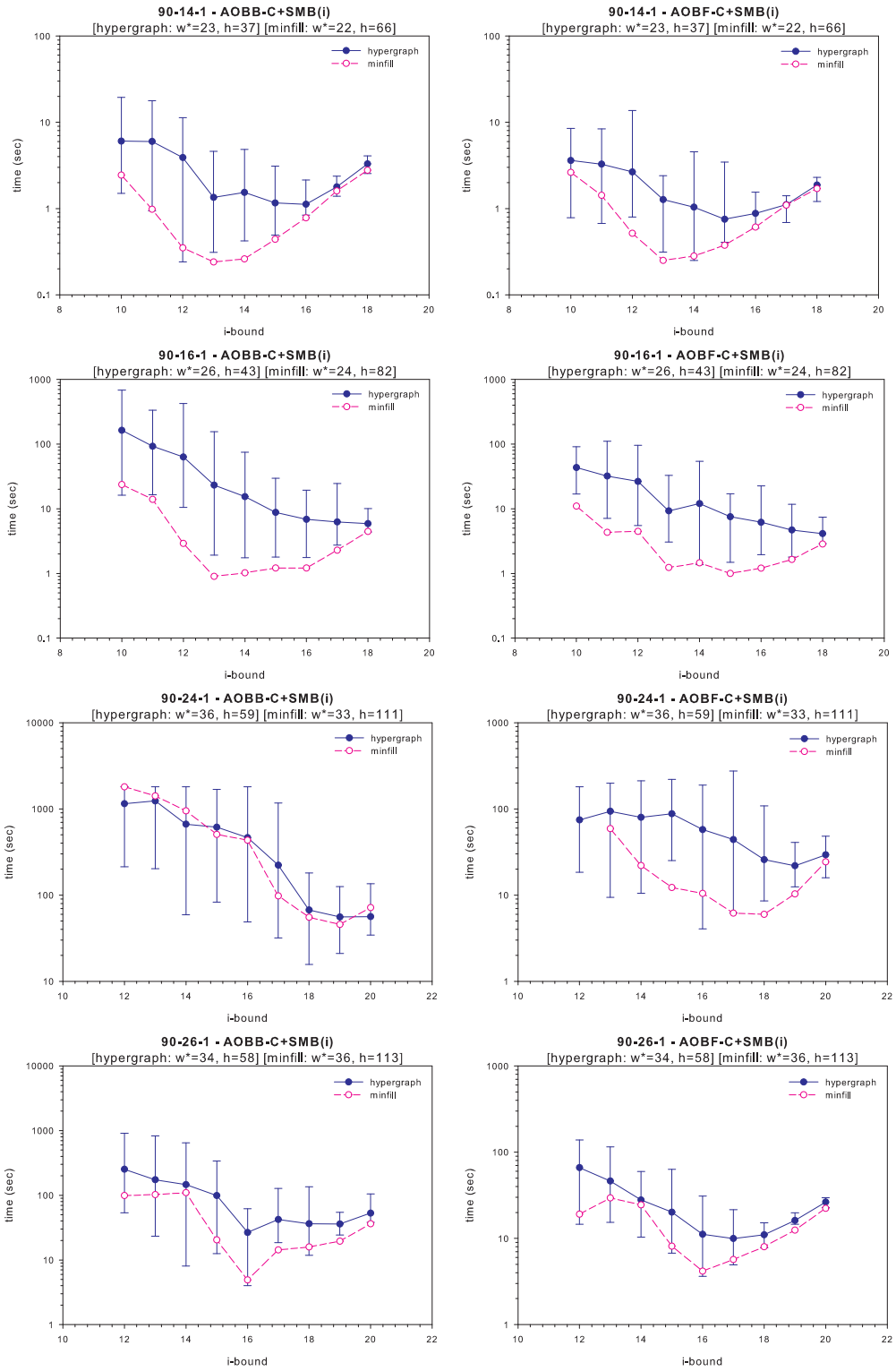


Figure 4.7: Min-fill versus ypergraph partitioning heuristics. CPU time in seconds for solving **grid networks** with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

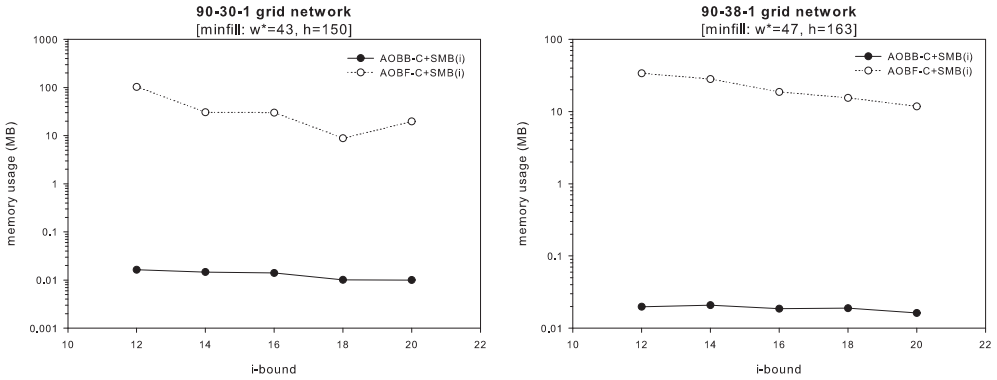


Figure 4.8: Memory usage by AOBB-C+SMB(i) and AOBF-C+SMB(i) on **grid networks**.

minfill pseudo tree											
pedigree (w*, h) (n, d)	SamIam Superlink	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=6		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=8		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=10		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.05		0.05		0.11		0.31		0.97	
ped1	5.44	-	-	-	-	1.14	7,997	0.73	3,911	1.31	2,704
(15, 61)	54.73	24.30	416,326	13.17	206,439	1.58	24,361	1.84	25,674	1.89	15,156
(299, 5)		4.19	69,751	2.17	33,908	0.39	4,576	0.65	6,306	1.36	4,494
		1.30	7,314	2.17	13,784	**0.26**	1,177	0.87	4,016	1.54	3,119
		0.12		0.45		5.38		60.97		out	
ped38	out	-	-	-	-	-	-	-	-		
(17, 59)	**28.36**	-	-	8120.58	85,367,022	-	-	3040.60	35,394,461		
(582, 5)		5946.44	34,828,046	1554.65	8,986,648	2046.95	11,868,672	272.69	1,412,976		
		out		134.41	348,723	216.94	583,401	103.17	242,429		
		0.11		0.74		5.38		37.19		out	
ped50	out	-	-	-	-	-	-	-	-		
(18, 58)	-	-	-	-	-	476.77	5,566,578	104.00	748,792		
(479, 5)		4140.29	28,201,843	2493.75	15,729,294	66.66	403,234	52.11	110,302		
		78.53	204,886	36.03	104,289	**12.75**	25,507	38.52	5,766		
		i=10		i=12		i=14		i=16		i=18	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.42		2.33		11.33		274.75		out	
ped23	out	-	-	-	-	76.11	339,125	270.22	74,261		
(27, 71)	9146.19	498.05	6,623,197	15.45	154,676	16.28	67,456	286.11	117,308		
(310, 5)		193.78	1,726,897	**10.06**	74,672	13.33	23,557	274.00	62,613		
		out		15.33	58,180	14.36	12,987	out			
		0.67		5.16		21.53		58.59		out	
ped37	out	-	-	-	-	-	-	-	-		
(21, 61)	64.17	273.39	3,191,218	1682.09	25,729,009	1096.79	15,598,863	128.16	953,061		
(1032, 5)		39.16	222,747	488.34	4,925,737	301.78	2,798,044	67.83	82,239		
		29.16	72,868	38.41	102,011	95.27	223,398	62.97	12,296		

Table 4.6: CPU time and nodes visited for solving **genetic linkage networks** using **static mini-bucket heuristics**. Time limit 3 hours. Top part of the table shows results for i-bounds between 6 and 14, while the bottom part shows i-bounds between 10 and 18.

minfill pseudo tree											
pedigree (w*, h) (n, d)	SamIam Superlink	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=16		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=18		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.51		1.42		4.59		12.87		19.30	
ped18	157.05	-	-	-	-	-	-	-	-	1515.43	1,388,791
(21, 119)	139.06	-	-	2177.81	28,651,103	270.96	2,555,078	100.61	682,175	20.27	7,689
(1184, 5)		-	-	406.88	3,567,729	52.91	397,934	23.83	118,869	20.60	2,972
		out		127.41	542,156	42.19	171,039	**19.85**	53,961	19.91	2,027
		1.42		5.11		37.53		410.96		out	
ped20	out	-	-	-	-	-	-	-	-		
(24, 66)	**14.72**	3793.31	54,941,659	1293.76	18,449,393	1259.05	17,810,674	1080.05	9,151,195		
(388, 5)		1983.00	18,615,009	635.74	6,424,477	512.16	4,814,751	681.97	2,654,646		
		out		out		out		out			
		0.34		0.89		3.20		10.46		33.42	
ped25	out	-	-	-	-	-	-	-	-	-	-
(34, 89)	-	-	-	-	-	9399.28	111,301,168	3607.82	34,306,937	2965.60	28,326,541
(994, 5)		-	-	1644.67	12,631,406	865.83	6,676,835	249.47	1,789,094	**236.88**	1,529,180
		out		out		out		out		out	
		0.42		0.83		1.78		5.75		21.30	
ped30	out	-	-	-	-	-	-	-	-	-	-
(23, 118)	13095.83	-	-	-	-	-	-	214.10	1,379,131	91.92	685,661
(1016, 5)		10212.70	93,233,570	8858.22	82,552,957	-	-	34.19	193,436	30.48	66,144
		out		out		out		30.39	72,798	**27.94**	18,795
		0.58		2.31		7.84		33.44		112.83	
ped33	out	-	-	-	-	-	-	-	-	-	-
(37, 165)	-	2804.61	34,229,495	737.96	9,114,411	3896.98	50,072,988	159.50	1,647,488	2956.47	35,903,215
(581, 5)		1426.99	11,349,475	307.39	2,504,020	1823.43	14,925,943	86.17	453,987	1373.90	10,570,695
		out		140.61	407,387	out		**74.86**	134,068	out	
		0.52		2.32		8.41		33.15		81.27	
ped39	out	-	-	-	-	-	-	-	-	-	-
(23, 94)	322.14	-	-	-	-	4041.56	52,804,044	386.13	2,171,470	141.23	407,280
(1272, 5)		-	-	-	-	968.03	7,880,928	61.20	313,496	93.19	83,714
		out		out		68.52	218,925	**41.69**	79,356	87.63	14,479
		4.20		31.33		96.28		out		out	
ped42	out	-	-	-	-	-	-				
(25, 76)	561.31	-	-	-	-	-	-				
(448, 5)		-	-	-	-	2364.67	22,595,247				
		out		out		**133.19**	93,831				

Table 4.7: CPU time and nodes visited for solving **genetic linkage networks**. Time limit 3 hours. Shown here are 7 linkage networks in addition to the 5 networks from Table 4.6.

Genetic Linkage Analysis

In human genetic linkage analysis [98], the *haplotype* is the sequence of alleles at different loci inherited by an individual from one parent, and the two haplotypes (maternal and paternal) of an individual constitute this individual's *genotype*. When genotypes are measured by standard procedures, the result is a list of unordered pairs of alleles, one pair for each locus. The *maximum likelihood haplotype* problem consists of finding a joint haplotype configuration for all members of the pedigree which maximizes the probability of data. The pedigree data can be represented as a belief network as described in Chapter 3. The haplotyping problem is equivalent to computing the Most Probable Explanation (MPE) of the corresponding belief network [47, 46].

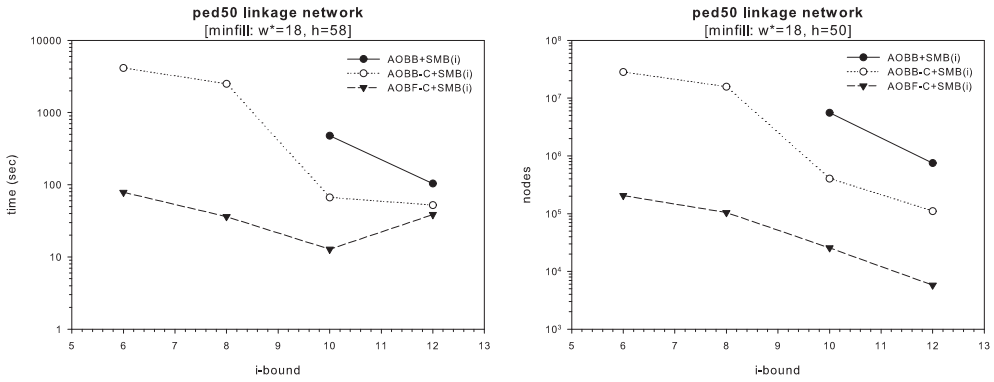


Figure 4.9: CPU time and nodes visited for solving the **ped50 linkage network**.

Tables 4.6 and 4.7 display the results obtained for 12 hard linkage analysis networks[3]. We report only on the AND/OR search algorithms guided by static mini-bucket heuristics. The dynamic mini-bucket heuristics performed very poorly on this domain because of their prohibitively high computational overhead at large i-bounds. For comparison, we include results obtained with SUPERLINK 1.6 (see also Chapter 3 for an overview).

Tree versus graph AOBB. We observe that AOBB-C+SMB(i) improves significantly over AOBB+SMB(i), especially for relatively small i-bounds for which the heuristic estimates are less accurate. On ped37, for example, AOBB-C+SMB(10) is 7 times faster than AOBB+SMB(10) and expands about 14 times fewer nodes. As the i-bound increases, the accuracy of the heuristics increases as well prunning the search space more efficiently and the diference between AOBB-C+SMB(i) and AOBB+SMB(i) decreases. Notice also that the OR Branch-and-Bound with caching BB-C+SMB(i) was able to solve only 3 out of the 12 test instances ($e.g.$, ped1, ped23, ped18). Similarly, the performance of SAMIAM was very poor and it was able to solve only 2 instances, namely ped1 and ped18.

AOBB vs. AOBF. The best performing algorithm on this dataset is AOBF-C+SMB(i), outperforming its competitors on 8 out of the 12 test cases. On ped42, for instance,

[3]http://bioinfo.cs.technion.ac.il/superlink/

		hypergraph pseudo tree					min-fill pseuso tree				
pedigree (n, d)	SamIam Superlink	(w*, h)	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20 time	nodes	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=22 time	nodes	(w*, h)	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20 time	nodes	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=22 time	nodes
ped7			25.26		164.49			117.03		out	
(868, 4)	out		-	-	-	-		-	-		
	-	(36, 60)	88571.68	1,807,878,340	9395.17	195,845,851	(32, 133)	-	-		
			30504.84	285,084,124	**3005.66**	27,761,219		-	-		
			out		out			out			
ped9			67.93		300.06			76.31		out	
(936, 7)	out		-	-	-	-		-	-		
	-	(35, 58)	11483.89	231,301,374	3982.69	72,844,362	(27, 130)	1515.50	15,825,340		
			8922.81	117,328,162	**3292.30**	40,251,723		**1163.09**	12,444,961		
			out		out			out			
ped19			59.31		150.38			out		out	
(693, 5)	out		-	-	-	-					
	-	(35, 53)	98941.75	1,519,213,924	12530.00	174,000,317	(24, 122)				
			45075.31	466,748,365	**8321.42**	90,665,870					
			out		out						
ped34			42.21		209.51			out		out	
(923, 4)	out		-	-	-	-					
	-	(34, 60)	70504.72	1,453,705,377	13598.50	294,637,173	(32, 127)				
			67647.42	1,293,350,829	**11719.28**	220,199,927					
			out		out						
ped41			35.41		111.24			out		out	
(886, 5)	out		-	-	-	-					
	-	(36, 61)	6669.50	84,506,068	531.40	4,990,995	(33, 128)				
			3891.86	31,731,270	**380.01**	2,318,544					
			out		out						
ped44			32.92		140.81			57.88		344.68	
(644, 4)	out		-	-	-	-		-	-	-	-
	-	(31, 52)	8388.18	196,823,840	401.84	7,648,962	(26, 73)	127.42	1,114,641	385.47	668,737
			3597.12	62,385,573	**204.96**	1,355,595		**95.09**	752,970	366.18	447,514
			out		out			out		out	

Table 4.8: Impact of the pseudo tree quality on **genetic linkage networks**. Time limit 24 hours. We show results for the hypergraph partitioning heuristic (left side) and the min-fill heuristic (right side).

AOBF-C+SMB(16) is 18 times faster than AOBB-C+SMB(16) and explores a search space 240 times smaller. In some cases, AOBF-C+SMB(i) was up to 3 orders of magnitude faster than SUPERLINK as well ($e.g.$, ped1, ped23, ped30). Figure 4.9 displays the CPU time and number of nodes explored, as a function of the mini-bucket i-bound, for solving the ped50 instance. It shows how the performance of best-first and depth-first AND/OR search changes with the i-bound. In this case, AOBB+SMB(i) could not solve the problem instance for $i \in \{6, 8\}$, due to exceeding the time limit.

Impact of the pseudo tree. Figure 4.10 plots the runtime distribution of the depth-first and best-first search algorithms AOBB-C+SMB(i) and AOBF-C+SMB(i), with hypergraph based pseudo trees, over 20 independent runs. We see that both AOBB-C+SMB(i) and AOBF-C+SMB(i) perform much better when guided by hypergraph based pseudo

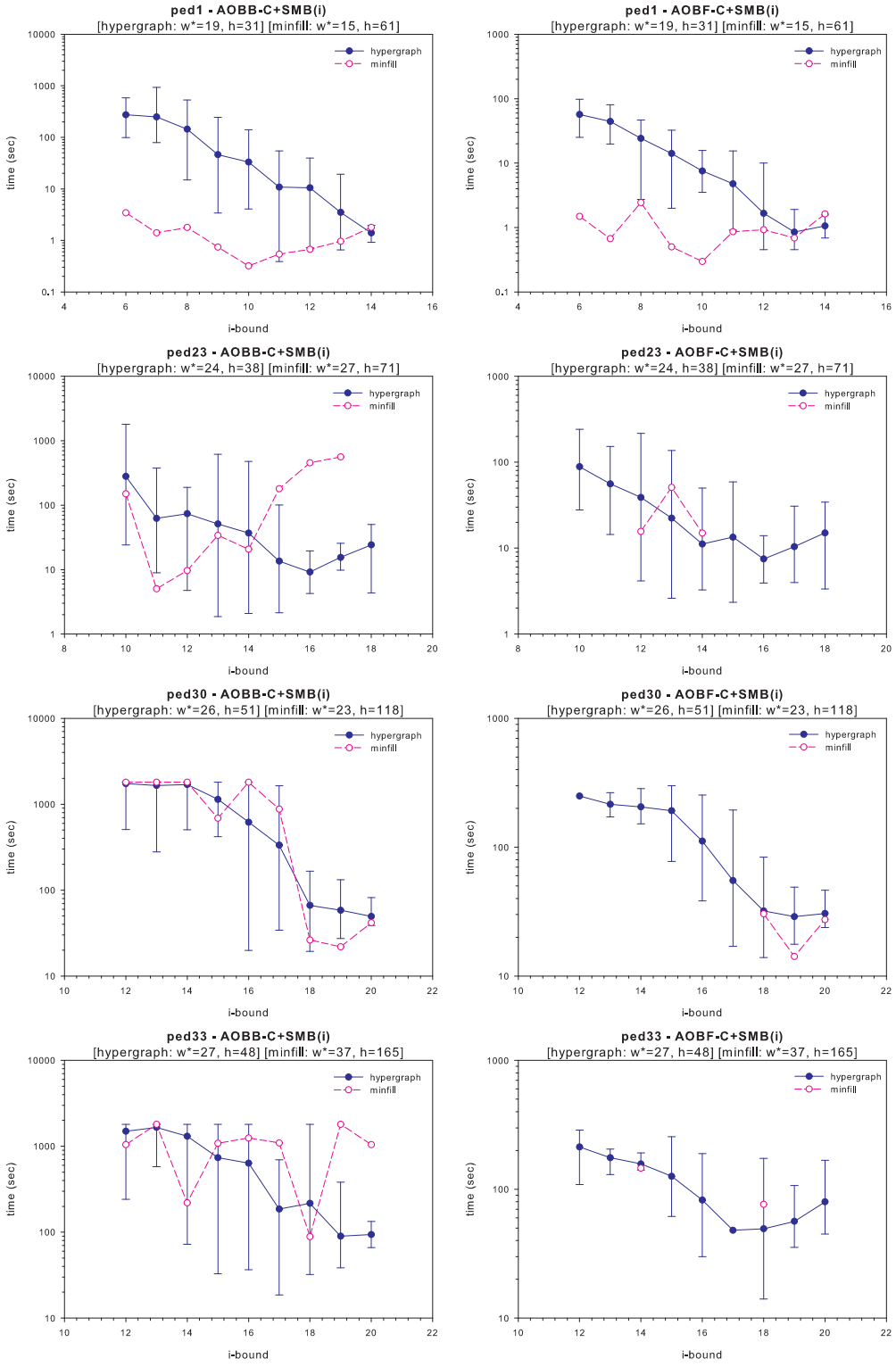


Figure 4.10: Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **genetic linkage networks** with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

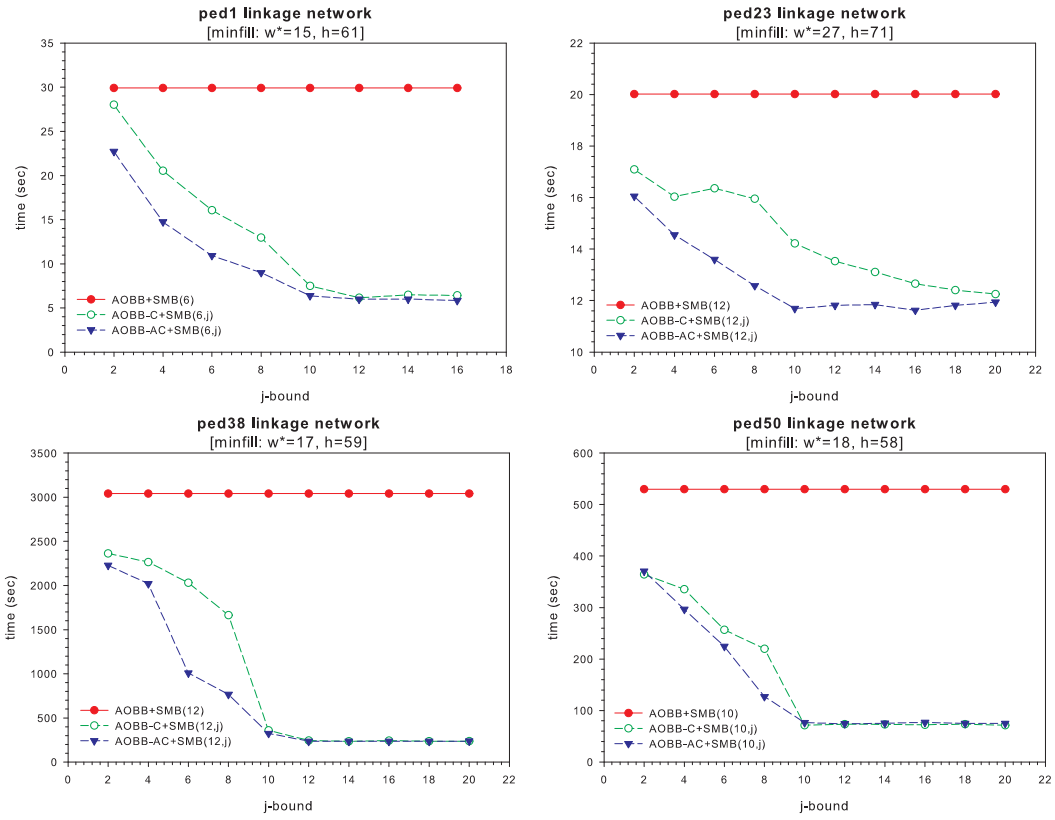


Figure 4.11: Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **genetic linkage networks**. Shown is CPU time in seconds.

trees, especially on harder instances. For instance, on `ped33`, `AOBB-C+SMB(16)` using a hypergraph tree was able to outperform `AOBB-C+SMB(16)` guided by a min-fill tree by almost 2 orders of magnitude. Similarly, `AOBF-C+SMB(`i`)` with hypergraph trees was able to solve the problem instance across all i-bounds, unlike `AOBF-C+SMB(`i`)` with a min-fill tree which succeded only for $i \in \{14, 18\}$. Notice that the induced width of this problem along the min-fill ordering is very large ($w^* = 37$) which causes the mini-bucket heuristics to be relatively weak. More importantly, it causes the AND/OR search algorithms to traverse and AND/OR search graph that is very close to a tree because most of the cache entries are dead.

Table 4.8 displays the results obtained for 6 additional linkage analysis networks using randomized hypergraph partitioning based pseudo trees. We selected the hypergraph tree having the smallest depth over 100 independent runs (ties were broken on the smallest

induced width). To the best of our knowledge, these networks were never before solved for the maximum likelihood haplotype task. We see that the hypergraph pseudo trees offer the overall best performance as well. This can be explained by the large induced width which in this case renders most of the cache entries dead (see for instance that the difference between AOBB+SMB(i) and AOBB-C+SMB(i) is not too prominent). Therefore, the AND/OR graph explored effectively is very close to a tree and the dominant factor that impacts the search performance is then the depth of the guiding pseudo tree, which is far smaller for hypergraph trees compared with min-fill based ones. Notice also that best-first search could not solve any of these networks due to running out of memory. The AND/OR Branch-and-Bound algorithms with min-fill based pseudo trees could only solve 2 of the test instances ($e.g.$, ped9 and ped44). This is because the induced width of these problem instances was small enough and the mini-bucket heuristics were relatively accurate to prune the search space substantially, thus overcomming the increase in pseudo tree depth. One thing that these experiments demonstrate is that the selection of the pseudo tree can have an enormous impact if the i-bound is not large enough.

Impact of the level of caching. Figure 4.11 displays the CPU time for solving 4 linkage analysis networks from Tables 4.6 and 4.7 using AOBB-C+SMB(i,j) (naive caching) and AOBB-AC+SMB(i,j) (adaptive caching), respectively. In each test case we chose a relatively small mini-bucket i-bound and varied the cache bound j (the X axis) from 2 to 20. We see again that adaptive caching is more powerful than the naive scheme especially, for relatively small j-bounds, which require restricted space. As the j-bound increases, the two schemes approach gradually full caching.

UAI'06 Evaluation Dataset

Tables 4.9 and 4.10 show the results for experiments with 15 networks from the UAI'06 repository described in Chapter 3. Instances BN_31 through BN_41 are random grid net-

minfill pseudo tree											
bn (w*, h) (n, k)	SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=17		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=18		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=19		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=21	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		5.53		10.31		17.45		38.36		62.11	
BN_31		-	-	-	-	-	-	-	-	-	-
(46, 160)	out	1026.73	4,741,037	1394.90	7,895,304	664.27	3,988,933	680.61	4,293,760	131.17	380,470
(1156, 2)		411.33	1,445,200	486.47	2,131,977	209.80	831,431	210.81	889,782	81.61	94,507
		140.41	293,445	126.23	292,293	85.69	142,650	86.00	114,046	**73.14**	25,392
		7.39		13.34		24.38		46.08		81.72	
BN_33		-	-	-	-	-	-	-	-	-	-
(43, 163)	-	1404.15	3,540,778	293.85	685,246	618.55	1,441,245	410.08	1,018,353	197.08	360,880
(1444, 2)		429.02	982,130	125.78	210,552	236.42	408,855	160.61	256,191	120.33	89,308
		75.92	142,932	**41.14**	41,865	58.14	61,064	73.20	49,760	95.16	22,256
		7.61		12.86		24.50		40.33		64.63	
BN_35		-	-	-	-	-	-	-	-	-	-
(41, 168)	-	464.44	1,755,561	548.11	1,954,720	316.78	1,108,708	199.67	663,784	226.10	622,551
(1444, 2)		42.95	126,215	107.17	243,533	81.59	151,632	56.11	65,657	78.27	58,973
		29.77	29,837	36.58	34,987	43.28	28,088	51.28	15,953	76.28	18,048
		7.25		13.58		22.61		44.14		87.30	
BN_37		-	-	-	-	-	-	-	-	-	-
(45, 159)	-	126.85	428,643	97.03	298,477	79.75	183,016	65.74	89,948	121.39	168,957
(1444, 2)		26.42	55,571	20.19	33,475	25.45	14,703	45.61	8,815	94.55	16,400
		15.83	15,399	19.47	11,046	26.55	6,621	46.84	4,315	90.66	5,610
		6.86		13.13		25.58		44.06		75.49	
BN_39		-	-	-	-	-	-	-	-	-	-
(48, 162)	-	-	-	-	-	-	-	-	-	1202.01	3,366,427
(1444, 2)		1161.65	2,615,679	1370.21	3,448,072	507.18	1,499,020	403.07	1,043,378	220.74	518,011
		117.03	340,362	247.08	725,738	131.44	316,862	112.27	213,676	**111.20**	127,872
		6.97		11.98		21.09		36.44		65.75	
BN_41		-	-	-	-	-	-	-	-	-	-
(49, 164)	-	188.60	486,844	151.80	364,363	83.39	168,340	109.92	195,506	123.58	162,274
(1444, 2)		56.72	119,737	47.30	77,653	33.81	32,774	50.81	38,467	76.42	31,763
		23.50	42,795	**22.05**	20,485	27.22	12,030	43.38	16,549	71.61	11,648

Table 4.9: CPU time and nodes visited for solving **UAI'06 networks**. Time limit 30 minutes.

works with deterministic CPTs, while instances `BN_126` through `BN_134` represent random coding networks with 128 input bits, 4 parents per XOR bit and channel variance $\sigma^2 = 0.40$. We report only on the Branch-and-Bound and Best-First search algorithms using static mini-bucket heuristics. The dynamic mini-bucket heuristics were not competitive due to their much higher computational overhead at relatively large i-bounds. The guiding pseudo trees were generated in this case using the min-fill heuristic.

We notice again the superiority of `AOBB-C+SMB(`i`)` compared with the tree version of the algorithm, `AOBB+SMB(`i`)`, at relatively small i-bounds where both algorithms rely primarily on search rather than on pruning, and especially on the first set of grid networks ($e.g.$, `BN_31`, ..., `BN_41`). For instance, on the `BN_35` network, `AOBB-C+SMB(17)` finds the most probable explanation 11 times faster than `AOBB+SMB(17)` exploring a search space 14 times smaller. This is in contrast to what we observe on the second set of coding networks ($e.g.$, `BN_126`, ..., `BN_133`), where `AOBB-C+SMB(`i`)` is only slightly better than

minfill pseudo tree											
bn (w*, h) (n, k)	SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=17		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=18		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=19		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=20		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=21	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		3.27		6.69		11.63		23.42		47.84	
BN_126		301.56	2,085,673	823.32	6,662,948	512.27	3,189,855	55.16	257,866	54.39	70,027
(54, 70)	-	363.05	4,459,174	953.71	10,991,861	118.58	1,333,266	52.24	386,490	57.74	150,391
(512, 2)		255.27	2,324,776	816.71	8,423,064	92.85	829,994	47.67	244,943	55.27	99,056
		16.22	64,202	25.64	85,148	26.88	76,645	30.95	37,666	54.59	30,197
		3.42		6.66		14.59		26.66		47.66	
BN_127		-	-	-	-	-	-	-	-	130.27	631,093
(57, 74)	out	-	-	-	-	-	-	-	-	155.09	1,384,957
(512, 2)		-	-	-	-	-	-	-	-	128.94	860,026
		51.80	223,327	58.63	251,134	62.75	215,796	64.28	166,741	66.35	84,007
		3.81		7.58		13.64		28.30		49.02	
BN_128		4.14	2,558	7.59	1,266	14.84	10,244	28.31	471	49.13	3,147
(48, 73)	out	4.13	5,587	7.47	1,712	14.89	18,734	29.05	625	49.39	5,823
(512, 2)		4.11	3,636	7.48	1,411	15.00	12,034	28.38	552	49.33	4,203
		3.97	883	7.75	925	13.78	808	28.39	478	49.13	575
		3.56		5.58		12.67		27.81		50.60	
BN_129		-	-	-	-	176.24	1,603,304	1337.90	11,794,805	257.42	1,855,134
(52, 68)	out	865.99	11,469,012	-	-	194.91	1,999,591	-	-	259.83	2,542,057
(512, 2)		573.74	5,730,592	-	-	167.14	1,688,675	1388.01	13,437,762	219.09	1,747,613
		out		194.56	922,831	out		**132.45**	537,371	246.39	910,769
		3.03		6.50		10.95		26.31		46.44	
BN_130		**21.56**	182,120	-	-	869.44	7,310,190	-	-	57.06	109,669
(54, 67)	out	28.67	348,660	-	-	1015.05	10,905,151	-	-	60.91	205,010
(512, 2)		22.49	239,771	-	-	863.15	8,414,475	-	-	58.94	147,085
		27.72	115,091	68.53	273,987	76.53	299,172	60.55	158,650	69.63	107,771
		3.44		6.59		11.20		21.88		39.70	
BN_131		**17.06**	137,631	39.02	323,431	1149.74	10,230,128	47.25	228,703	-	-
(48, 72)	out	24.36	296,576	55.20	677,149	-	-	66.63	673,358	-	-
(512, 2)		18.69	176,456	41.63	396,234	1254.88	12,395,143	50.42	303,818	-	-
		29.03	116,166	50.13	209,748	28.47	79,689	36.89	73,163	65.74	120,153
		2.95		5.59		10.50		25.56		45.77	
BN_132		-	-	-	-	-	-	756.69	6,584,446	**578.99**	4,819,402
(49, 71)	out	-	-	-	-	-	-	912.40	10,251,600	823.40	10,207,347
(512, 2)		-	-	-	-	-	-	778.22	7,456,812	643.96	6,037,908
		out		out		out		out		out	
		3.61		7.03		13.20		27.50		52.69	
BN_133		-	-	**16.84**	104,521	31.28	171,645	127.32	929,016	55.33	30,699
(54, 71)	out	-	-	19.38	169,574	35.58	272,258	168.17	1,859,117	56.22	71,195
(512, 2)		-	-	17.19	133,794	31.64	202,954	135.60	1,184,600	55.22	40,483
		59.61	258,891	27.44	98,148	32.91	93,613	45.09	90,337	55.31	13,491
		3.38		6.34		12.09		27.08		54.35	
BN_134		-	-	-	-	-	-	-	-	-	-
(52, 70)	out	-	-	-	-	-	-	-	-	-	-
(512, 2)		-	-	-	-	-	-	-	-	-	-
		out		**85.77**	373,081	out		96.19	377,064	97.59	214,591

Table 4.10: CPU time and nodes visited for solving **UAI'06 networks**. Time limit 30 minutes.

AOBB+SMB(i) across the reported i-bounds. This is because the AND/OR graph explored effectively was very close to a tree due to the substantial pruning caused by the mini-bucket heuristics (as also observed in Section 4.5.2).

Overall, best-first AND/OR search offers the best performance on this domain and the difference in running time as well as size of the search space explored is up to several orders of magnitude, compared to the Branch-and-Bound algorithms. For example, on the BN_133 network, AOBF-C+SMB(17) found the optimal solution in about 1 minute, whereas both AOBB+SMB(17) and AOBB-C+SMB(17) exceed the 30 minute

time bound.

Figure 4.12 plots the runtime distribution of AOBB-C+SMB(i) and AOBF-C+SMB(i) using hypergraph partitioning based pseudo trees, over 20 independent runs, on 4 UAI'06 networks. We see that the hypergraph trees are sometimes able to improve the performance of AOBB-C+SMB(i), especially at small i-bounds ($e.g.$, BN_133). For best-first search, the min-fill trees usually offer the best performance (except on BN_131, where the hypergraph trees are superior across i-bounds).

ISCAS'89 Circuits

Tables 4.11 and 4.12 show the results for experiments with 10 belief networks derived from ISCAS'89 circuits (as described in Chapter 3), using min-fill based pseudo trees as well as static and dynamic mini-bucket heuristics. For each test instance we generated a single MPE query without any evidence. We see that AOBB-C+SMB(i) improves over AOBB+SMB(i), especially at relatively small i-bounds. For instance, on the s1196 circuit, AOBB-C+SMB(8) is about 3 times faster than AOBB+SMB(i). This is in contrast to what we see when using dynamic mini-bucket heuristics. Here, there is no noticable difference between the tree and graph AND/OR Branch-and-Bound, because the pruning power of the heuristics rendered the search space almost backtrack free, across i-bounds. Overall, the dynamic mini-bucket heuristics were inferior to the corresponding static ones for large i-bounds, however, smaller i-bound dynamic mini-buckets were often cost-effective. Notice that SAMIAM is able to solve only 2 out of 10 test instances. Moreover, AOBF-C+SMB(i) (resp. AOBF-C+DMB(i)) was overall inferior to AOBB-C+SMB(i) (resp. AOBB-C+DMB(i)) because of its computational overhead.

4.5.3 The Anytime Behavior of AND/OR Branch-and-Bound Search

As mentioned earlier, the virtue of AND/OR Branch-and-Bound search is that, unlike Best-First AND/OR search, it is an anytime algorithm. Namely, whenever interrupted,

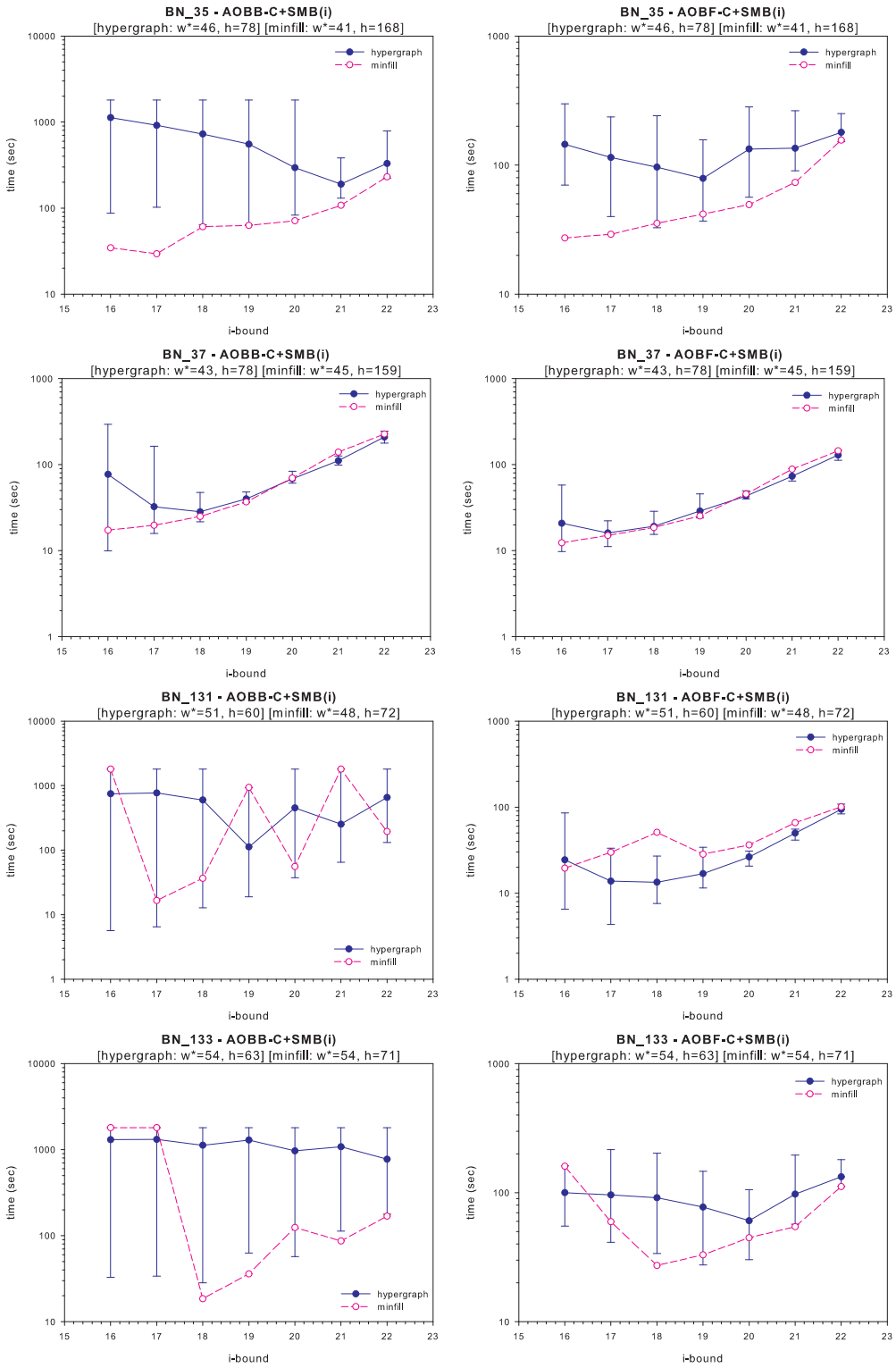


Figure 4.12: Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **UAI'06 networks** with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

minfill pseudo tree											
iscas89 (w*, h) (n, d)	SamIam	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=6		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=8		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=10		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.06		0.08		0.09		0.14		0.20	
c432		-	-	-	-	-	-	0.35	432	0.45	432
(27, 45)	out	-	-	-	-	1154.96	20,751,699	**0.16**	432	0.24	432
(432, 2)		-	-	-	-	182.53	2,316,024	**0.16**	432	0.24	432
		out		out		106.27	488,462	0.20	432	0.28	432
		0.08		0.09		0.09		0.16		0.30	
c499		0.38	499	0.42	499	0.42	499	0.48	499	0.59	499
(23, 55)	139.89	**0.11**	499	0.13	499	0.13	499	0.19	499	0.33	499
(499, 2)		0.13	499	0.11	499	0.14	499	0.19	499	0.31	499
		0.17	499	0.17	499	0.17	499	0.25	499	0.39	499
		0.17		0.16		0.19		0.22		0.44	
c880		-	-	1.56	881	1.80	881	1.70	881	1.84	881
(27, 67)	out	0.23	884	**0.22**	881	0.25	881	0.28	881	0.50	881
(880, 2)		**0.22**	884	0.24	881	0.25	881	0.28	881	0.48	881
		0.33	884	0.34	881	0.36	881	0.39	881	0.61	881
		0.03		0.03		0.05		0.08		0.16	
s386		0.17	1,358	0.11	677	0.06	172	0.09	172	0.17	172
(19, 44)	3.66	**0.05**	257	**0.05**	257	**0.05**	172	0.08	172	0.16	172
(172, 2)		**0.05**	207	**0.05**	207	**0.05**	172	0.08	172	0.16	172
		0.05	194	**0.05**	194	0.06	172	0.08	172	0.16	172
		0.13		0.14		0.17		0.28		0.70	
s953		-	-	-	-	-	-	-	-	1170.80	4,031,967
(66, 101)	out	1054.79	9,919,295	23.67	238,780	58.00	549,181	36.06	434,481	2.72	21,499
(440, 2)		899.63	7,715,133	17.99	155,865	48.13	417,924	17.00	132,139	**2.19**	13,039
		out		41.03	150,598	110.45	408,828	36.50	113,322	4.06	12,256
		0.14		0.16		0.19		0.34		0.91	
s1196		-	-	-	-	-	-	-	-	-	-
(54, 97)	out	31.55	316,875	332.14	3,682,077	7.44	77,205	31.39	320,205	26.24	289,873
(560, 2)		18.05	104,316	124.53	686,069	**3.69**	26,847	14.23	94,985	9.47	62,883
		26.16	77,019	158.19	372,129	7.22	23,348	26.97	80,264	17.64	48,114
		0.14		0.16		0.20		0.36		0.86	
s1238		-	-	-	-	398.13	2,078,885	208.45	1,094,713	931.71	4,305,175
(59, 94)	out	4.45	57,355	14.77	187,499	3.70	47,340	2.28	25,538	2.45	20,689
(540, 2)		1.77	12,623	4.95	34,056	1.30	8,476	**1.00**	5,418	1.42	4,780
		2.30	5,921	6.61	17,757	1.70	4,298	1.31	2,730	1.69	2,415
		0.13		0.12		0.14		0.16		0.31	
s1423		-	-	-	-	-	-	0.98	762	1.19	749
(24, 54)	107.48	0.27	1,986	0.50	5,171	0.53	5,078	**0.22**	866	0.36	749
(748, 2)		**0.22**	1,246	**0.22**	1,256	**0.22**	1,235	**0.22**	818	0.36	749
		0.31	959	0.31	921	0.31	913	0.31	774	0.44	749
		0.14		0.17		0.22		0.39		1.00	
s1488		15.38	92,764	1.69	6,460	3.20	17,410	1.77	6,511	1.94	4,083
(47, 67)	out	16.58	135,563	2.20	17,150	3.39	28,420	1.63	12,285	1.64	6,670
(667, 2)		13.22	82,294	**1.02**	5,920	2.50	15,621	1.19	6,024	1.47	3,516
		21.75	74,658	1.67	5,499	4.22	14,445	1.84	5,372	1.80	3,124
		0.14		0.17		0.22		0.42		1.06	
s1494		10.86	64,629	978.87	3,412,403	222.28	815,708	5.94	36,804	73.35	268,814
(48, 69)	out	14.75	158,070	47.41	479,498	11.69	118,754	18.74	202,343	3.06	21,530
(661, 2)		7.30	41,798	19.69	108,768	4.81	27,711	7.00	41,977	**2.06**	8,104
		9.67	24,849	27.28	65,859	7.86	19,678	11.48	28,793	3.03	6,484

Table 4.11: CPU time and nodes visited for solving belief networks derived from **ISCAS'89 circuits** with static mini-bucket heuristics and min-fill pseudo trees. Time limit 30 minutes.

minfill pseudo tree										
iscas89 (w*, h) (n, d)	BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=6		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=8		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=10		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=12		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=14	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
c432	-	-	159.56	21,215	2.50	432	3.20	432	4.61	432
(27, 45)	-	-	32.00	39,711	1.02	432	1.69	432	3.06	432
(432, 2)	1161.25	323,359	23.02	4,951	**1.00**	432	1.73	432	3.09	432
	1019.19	86,460	26.05	2,342	1.58	432	2.70	432	4.70	432
c499	1.95	499	2.11	499	2.52	499	3.77	499	6.67	499
(23, 55)	**0.58**	499	0.73	499	1.14	499	2.41	499	5.25	499
(499, 2)	**0.58**	499	0.74	499	1.14	499	2.41	499	5.27	499
	0.83	499	1.11	499	1.88	499	3.75	499	8.03	499
c880	8.30	881	10.64	881	10.19	881	13.33	881	18.56	881
(27, 67)	1.25	881	1.47	881	2.16	881	3.92	881	9.11	881
(880, 2)	**1.20**	881	1.42	881	2.11	881	3.94	881	9.03	881
	1.74	881	2.20	881	3.41	881	6.14	881	13.81	881
s386	0.22	172	0.28	172	0.39	172	0.59	172	1.05	172
(19, 44)	0.13	172	0.17	172	0.28	172	0.52	172	0.97	172
(172, 2)	**0.11**	172	0.17	172	0.30	172	0.52	172	0.97	172
	0.18	172	0.30	172	0.50	172	0.83	172	1.51	172
s953	33.02	2,737	16.75	912	46.28	1,009	17.20	467	137.08	577
(66, 101)	32.08	2,738	**15.95**	913	45.80	1,010	16.17	468	135.61	578
(440, 2)	32.23	2,738	15.98	913	45.92	1,010	16.14	468	136.09	578
	54.72	2,738	25.22	913	73.86	1,010	26.45	468	213.59	578
s1196	3.75	580	4.81	568	37.45	924	88.91	863	386.75	1,008
(54, 97)	1.56	660	2.45	568	33.30	924	77.02	863	362.32	1,008
(560, 2)	**1.55**	620	2.44	568	33.52	924	79.05	863	355.10	1,008
	2.53	604	4.03	568	63.70	924	154.17	857	676.68	1,008
s1238	43.56	5,841	6.77	601	302.53	17,278	36.39	651	76.70	558
(59, 94)	2.61	1,089	3.70	795	13.16	1,824	26.39	849	59.20	744
(540, 2)	**2.52**	704	3.63	619	12.97	996	26.22	667	59.09	571
	4.00	635	6.17	610	21.30	769	44.23	657	97.00	564
s1423	5.05	751	5.27	749	5.67	749	6.66	749	9.09	749
(24, 54)	0.88	751	0.97	749	1.36	749	2.27	749	4.75	749
(748, 2)	**0.83**	751	0.95	749	1.34	749	2.22	749	4.73	749
	1.24	751	1.56	749	2.28	749	3.69	749	7.45	749
s1488	4.34	670	4.39	670	5.81	668	10.64	667	27.50	667
(47, 67)	**1.13**	670	1.67	670	3.11	668	7.70	667	24.19	667
(667, 2)	**1.13**	670	1.64	670	3.06	668	7.67	667	24.25	667
	1.89	670	2.95	670	5.62	668	13.58	667	41.12	667
s1494	7.80	814	5.61	679	15.16	719	25.03	686	70.19	686
(48, 69)	7.53	898	**2.95**	679	12.59	719	22.44	686	68.11	686
(661, 2)	5.06	814	2.97	679	12.66	719	22.98	686	69.81	686
	8.00	814	4.50	679	17.39	719	30.20	686	88.50	686

Table 4.12: CPU time and nodes visited for solving belief networks derived from **ISCAS'89 circuits** with dynamic mini-bucket heuristics and min-fill pseudo trees. Time limit 30 minutes.

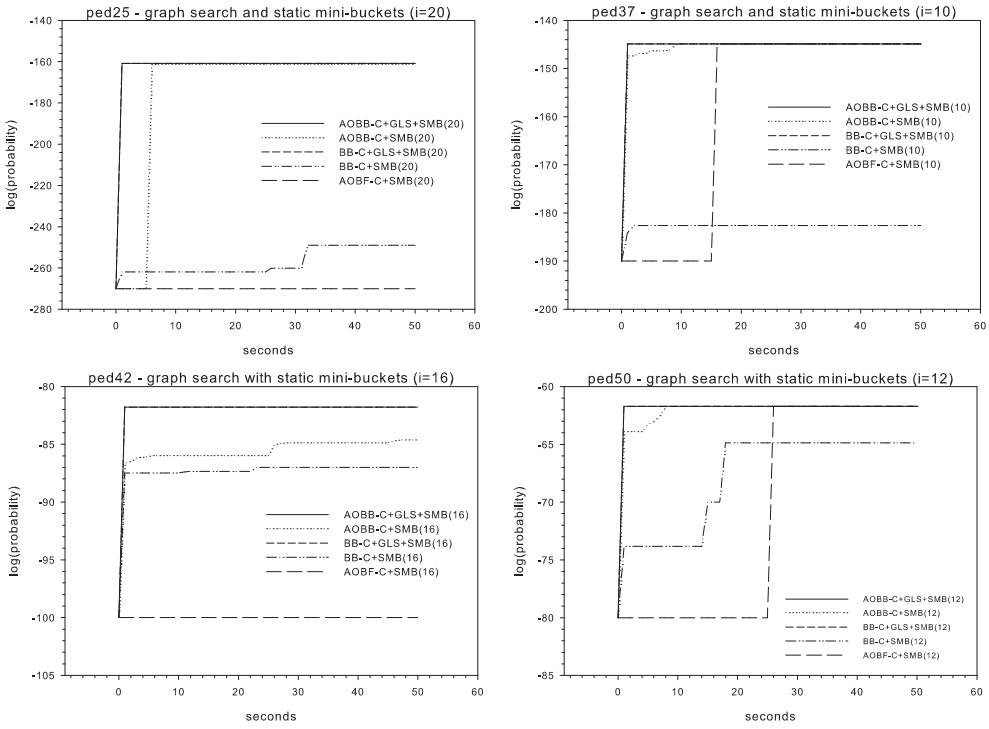


Figure 4.13: Anytime behavior of AOBB-C+SMB(i) on **genetic linkage networks**. Number of flips for GLS is 50,000. GLS running time is less than 1 second.

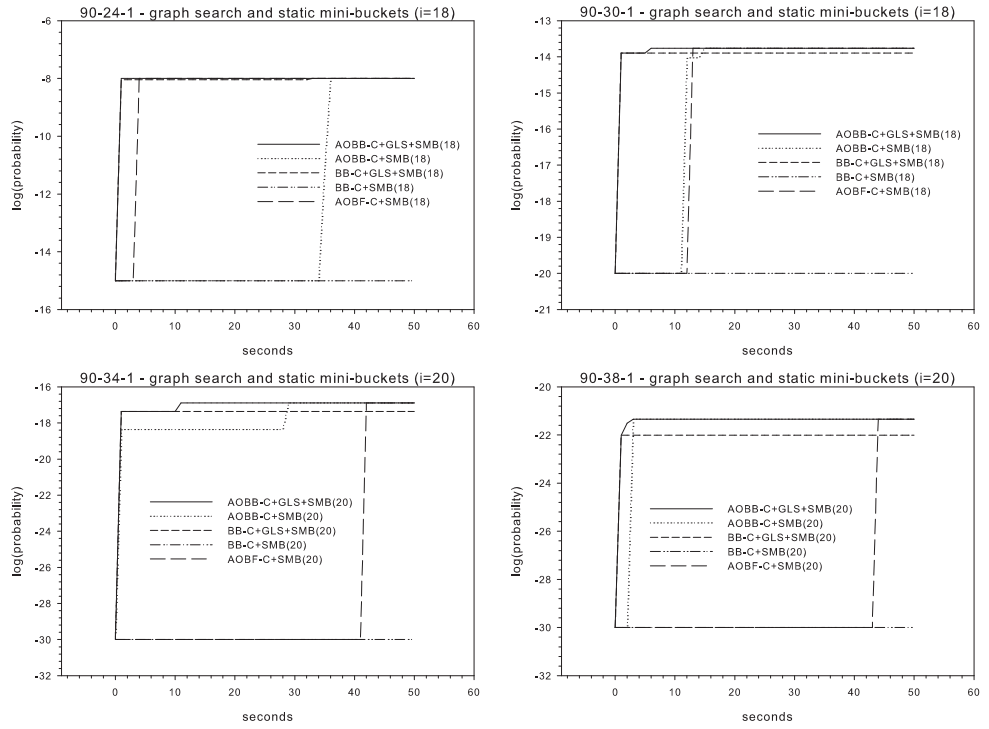


Figure 4.14: Anytime behavior of AOBB-C+SMB(i) on **grid networks**. Number of flips for GLS is 50,000. GLS running time is less than 1 second.

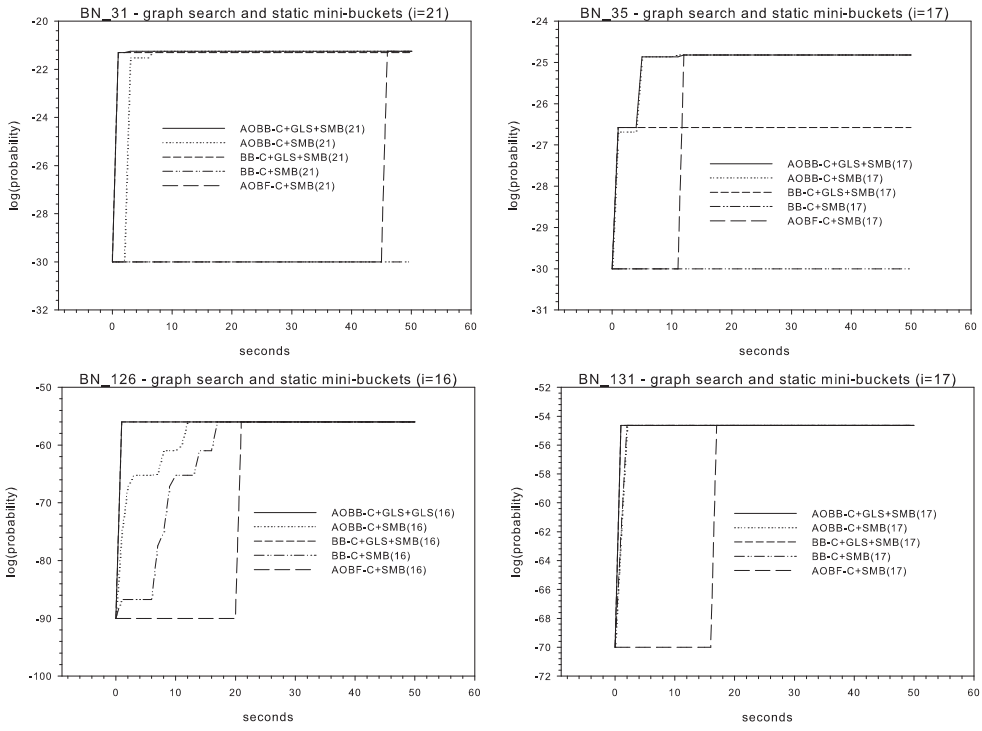


Figure 4.15: Anytime behavior of `AOBB-C+SMB(i)` on **UAI'06 networks**. Number of flips for GLS is 50,000. GLS running time is less than 1 second.

`AOBB-C` outputs a suboptimal solution ($i.e.$, the best solution found far), which yields a lower bound on the most probable explanation. On the other hand, `AOBF-C` outputs a complete solution only upon completion. In this section we evaluate the anytime behavior of `AOBB-C+SMB(`i`)`. We compare it against the state-of-the-art local search algorithm for Bayesian MPE, called *Guided Local Search* (GLS) first introduced in [102], and improved more recently by [57].

GLS [121] is a penalty-based meta-heuristic, which works by augmenting the objective function of a local search algorithm ($e.g.$ hill climbing) with penalties, to help guide them out of local minima. GLS has been shown to be successful in solving a number of practical real life problems, such as the traveling salesman problem, radio link frequency assignment problem and vehicle routing. It was also applied to solving the MPE in belief networks [102, 57] as well as weighted MAX-SAT problems [93].

The AND/OR Branch-and-Bound algorithms assumed a trivial initial lower bound ($i.e.$,

minfill pseudo tree											
pedigree (w*, h) (n, d)	SamIam Superlink GLS	BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=6		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=8		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=10		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=12		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=14	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	1.14	7,997	0.73	3,911	1.31	2,704
ped1	5.44	8943.68	59,627,660	1367.98	9,013,771	3.84	1,798	4.05	2,524	4.75	2,077
(15, 61)	54.73	4.19	69,751	2.17	33,908	0.39	4,576	0.65	6,306	1.36	4,494
(299, 5)	0.31	3.01	46,663	2.10	29,877	**0.13**	3,138	0.33	6,092	0.92	4,350
		1.30	7,314	2.17	13,784	0.26	1,177	0.87	4,016	1.54	3,119
		-	-	-	-	-	-	-	-	out	
ped38	out	-	-	-	-	-	-	-	-		
(17, 59)	**28.36**	5946.44	34,828,046	1554.65	8,986,648	2046.95	11,868,672	272.69	1,412,976		
(582, 5)	7.05	4410.70	32,599,034	780.46	4,487,470	1650.05	9,844,485	226.44	1,366,242		
		out		134.41	348,723	216.94	583,401	103.17	242,429		
		-	-	-	-	-	-	-	-	out	
ped50	out	-	-	-	-	-	-	52.95	83,025		
(18, 58)	-	4140.29	28,201,843	2493.75	15,729,294	66.66	403,234	52.11	110,302		
(479, 5)	5.30*	3177.43	24,209,840	1610.33	13,299,343	67.85	400,698	32.67	15,865		
		78.53	204,886	36.03	104,289	**12.75**	25,507	38.52	5,766		
		i=10		i=12		i=14		i=16		i=18	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	76.11	339,125	270.22	74,261	out	
ped23	out	8556.84	39,184,112	6640.68	28,790,468	15.27	23,947	270.25	55,412		
(27, 71)	9146.19	193.78	1,726,897	10.06	74,672	13.33	23,557	274.00	62,613		
(310, 5)	3.94	196.68	1,720,633	**7.56**	73,082	10.58	20,329	274.26	60,424		
		out		15.33	58,180	14.36	12,987	out			
		-	-	-	-	-	-	-	-	out	
ped37	out	-	-	2073.12	10,612,906	-	-	3386.01	16,382,262		
(21, 61)	64.17	39.16	222,747	488.34	4,925,737	301.78	2,798,044	67.83	82,239		
(1032, 5)	8.97*	**16.36**	141,867	26.97	254,219	82.08	604,239	52.32	23,572		
		29.16	72,868	38.41	102,011	95.27	223,398	62.97	12,296		

Table 4.13: CPU time and nodes visited for solving **genetic linkage analysis networks** with static mini-bucket heuristics. Number of flips for GLS was set to 250,000. Time limit 3 hours.

0), which effectively guarantees that the MPE will be computed, however it provides limited pruning. We therefore extended AOBB-C+SMB(i) to exploit a non-trivial initial lower bound computed by GLS. The algorithm is denoted by AOBB-C+GLS+SMB(i). For reference, we also ran the OR version of the algorithm, denoted by BB-C+GLS+SMB(i)

Figure 4.13 displays the search trace of the OR and AND/OR algorithms on 4 genetic linkage networks presented earlier. We chose the mini-bucket i-bound that offered the best performance in Tables 4.6 and 4.7, respectively, and show the first 50 seconds of the search. We ran GLS for a fixed number of flips. We see that including the GLS lower bound improves performance throughout. In all these test case, the initial lower bound was in fact the optimal solution (we did not plot the GLS running time because it was less than 1 second). Therefore, AOBB-C+GLS+SMB(i) and BB-C+GLS+SMB(i) were able to output the optimal solution quite early in the search, unlike AOBB-C+SMB(i) and BB-C+SMB(i). For instance, on the ped50 network, AOBB-C+GLS+SMB(12) and

minfill pseudo tree											
pedigree (w*, h) (n, d)	SamIam Superlink GLS	BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=12		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=14		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=16		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=18		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	-	-	-	-	1515.43	1,388,791
ped18	157.05	-	-	-	-	-	-	-	-	1672.15	1,389,831
(21, 119)	139.06	-	-	406.88	3,567,729	52.91	397,934	23.83	118,869	20.60	2,972
(1184, 5)	10.16	10780.40	107,804,665	170.14	1,824,835	37.64	396,961	11.66	118,170	**10.58**	2,720
		out		127.41	542,156	42.19	171,039	19.85	53,961	19.91	2,027
		-	-	-	-	-	-	-	-	out	
ped20	out	-	-	-	-	-	-	-	-		
(24, 66)	**14.72**	1983.00	18,615,009	635.74	6,424,477	512.16	4,814,751	681.97	2,654,646		
(388, 5)	4.22	2079.43	18,611,778	667.66	6,419,317	567.20	4,812,068	682.03	2,653,400		
		out		out		out		out			
		-	-	-	-	-	-	-	-	-	-
ped25	out	-	-	-	-	-	-	-	-	-	-
(34, 89)	-	-	-	1644.67	12,631,406	865.83	6,676,835	249.47	1,789,094	**236.88**	1,529,180
(994, 5)	11.03*	-	-	1644.87	12,631,282	864.09	6,676,061	245.79	1,788,621	239.08	1,529,588
		out		out		out		out		out	
		-	-	-	-	-	-	-	-	-	-
ped30	out	-	-	-	-	-	-	-	-	-	-
(23, 118)	13095.83	10212.70	93,233,570	8858.22	82,552,957	-	-	34.19	193,436	30.48	66,144
(1016, 5)	11.00	10620.20	93,030,080	9296.01	82,552,786	-	-	32.16	193,419	**22.25**	66,128
		out		out		out		30.39	72,798	27.94	18,795
		-	-	-	-	-	-	-	-	-	-
ped33	out	-	-	-	-	-	-	-	-	-	-
(37, 165)	-	1426.99	11,349,475	307.39	2,504,020	1823.43	14,925,943	86.17	453,987	1373.90	10,570,695
(581, 5)	6.86*	1550.76	11,528,022	320.06	2,434,582	1970.72	15,124,932	80.61	453,446	1518.24	10,970,922
		out		140.61	407,387	out		**74.86**	134,068	out	
		-	-	-	-	-	-	-	-	-	-
ped39	out	-	-	-	-	-	-	-	-	-	-
(23, 94)	322.14	-	-	-	-	968.03	7,880,928	61.20	313,496	93.19	83,714
(1272, 5)	10.97*	-	-	-	-	518.04	6,473,615	59.14	313,340	81.24	61,291
		out		out		68.52	218,925	**41.69**	79,356	87.63	14,479
		-	-	-	-	-	-	out		out	
ped42	out	-	-	-	-	-	-				
(25, 76)	561.31	-	-	-	-	2364.67	22,595,247				
(448, 5)	4.25*	-	-	-	-	385.26	3,078,657				
		out		out		**133.19**	93,831				

Table 4.14: CPU time and nodes visited for solving **genetic linkage analysis networks** with static mini-bucket heuristics. Number of flips for GLS was set to 250,000. Time limit 3 hours.

`BB-C+GLS+SMB(12)` found the optimal solution within the first second of the search. `AOBB-C+SMB(12)`, on the other hand, finds the optimal solution after 8 seconds, whereas `BB-C+SMB(12)` reaches a flat region after 18 seconds. In this case, `AOBF-C+SMB(12)` finds the optimal solution after 25 seconds.

Tables 4.13 and 4.14 compare the OR and AND/OR graph search algorithms with and without an initial lower bound, as complete algorithms. Algorithms `AOBB-C+GLS+SMB(`i`)` and `BB-C+GLS+SMB(`i`)` do not include the GLS time, because GLS can be tuned independently for each problem instance to minimize its running time, so we report its time separately (as before, GLS ran for a fixed number of flips). The "*" by the GLS running time indicates that it found the optimal solution to the respective problem instance. We see that indeed `BB-C+GLS+SMB(`i`)` and `AOBB-C+GLS+SMB(`i`)` are sometimes able to

minfill pseudo tree											
grid (w*, h) (n, e)	SamIam GLS	BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=8		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=10		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=12		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=14		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=16	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		0.23	3,297	0.06	373	**0.05**	102	0.06	102	0.06	102
90-10-1	0.13	0.38	3,272	0.19	289	0.19	0	0.19	0	0.20	0
(16, 26)	0.25*	0.14	2,638	0.06	819	0.05	101	0.06	101	0.06	101
(100, 0)		0.28	2,580	0.22	789	0.19	0	0.20	0	0.19	0
		0.27	2,012	0.11	661	**0.05**	100	0.06	100	0.06	100
		126.69	1,233,891	121.00	1,317,992	1.52	16,547	0.42	2,770	0.61	1,450
90-14-1	11.97	21.02	217,185	31.64	339,762	0.88	5,892	0.50	1,122	0.78	1,178
(23, 37)	0.43*	4.22	55,120	3.66	48,513	0.45	5,585	**0.23**	1,361	0.53	1,210
(196, 0)		3.59	45,023	2.77	32,454	0.66	3,684	0.45	1,067	0.78	1,062
		3.20	18,796	2.70	15,764	0.55	2,899	0.30	898	0.63	857
		-	-	-	-	40.05	345,255	2.38	16,942	1.23	5,327
90-16-1	147.19	-	-	1163.43	9,106,361	35.72	306,583	1.97	12,104	1.42	4,614
(26, 42)	0.49*	209.60	2,695,249	35.45	441,364	4.23	50,481	1.19	11,029	**0.95**	4,810
(256, 0)		37.28	453,073	8.14	96,962	4.17	46,138	1.44	10,702	1.23	4,552
		25.70	126,861	10.59	54,796	4.47	22,993	1.42	6,015	1.22	3,067
		i=12		i=14		i=16		i=18		i=20	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	-	-	-	-	-	-
90-24-1	out	-	-	1773.64	6,065,308	609.65	2,008,431	111.58	263,250	632.68	1,705,699
(36, 61)	0.53	-	-	1273.09	9,047,518	596.27	4,923,760	70.42	473,675	74.99	412,291
(576, 20)		3594.60	24,363,798	66.20	425,585	20.16	93,911	11.17	7,850	28.16	27,868
		out		21.94	75,637	10.59	33,770	**6.06**	5,144	23.80	17,291
		-	-	-	-	395.67	1,635,447	-	-	67.09	277,685
90-26-1	out	-	-	-	-	235.36	922,243	65.39	282,394	41.70	73,616
(35, 64)	0.56	146.97	878,874	152.80	962,484	4.36	15,632	12.92	46,489	22.13	2,242
(676, 40)		43.64	248,603	85.72	495,039	10.83	14,580	14.47	6,226	28.38	1,466
		19.06	65,271	24.39	79,619	**4.27**	7,190	8.05	3,777	22.44	1,435
		-	-	-	-	-	-	-	-	-	-
90-30-1	out	-	-	-	-	-	-	-	-	-	-
(38, 68)	0.72	652.15	3,882,300	165.74	1,070,823	155.20	956,837	40.14	212,963	59.28	174,715
(900, 60)		276.00	1,491,880	84.39	442,754	78.81	376,916	31.69	89,045	64.23	148,540
		158.97	534,385	46.73	157,187	47.27	154,496	**21.06**	45,201	57.97	100,800
		-	-	-	-	-	-	-	-	-	-
90-34-1	out	-	-	-	-	-	-	-	-	-	-
(43, 79)	1.31	-	-	-	-	-	-	-	-	369.36	823,604
(1154, 80)		-	-	-	-	980.51	4,943,817	1751.86	5,516,888	315.38	630,406
		out		out		243.63	596,978	270.88	667,013	**71.19**	67,611
		-	-	-	-	-	-	-	-	-	-
90-38-1	out	-	-	-	-	-	-	-	-	-	-
(47, 86)	1.11	969.02	2,623,971	1753.10	3,794,053	203.67	614,868	165.45	488,873	113.06	214,919
(1444, 120)		819.16	2,450,643	1806.57	3,804,190	224.80	607,453	187.63	482,946	138.64	211,562
		101.69	174,786	103.80	146,237	54.00	95,511	**53.44**	78,431	73.10	59,856

Table 4.15: CPU time and nodes visited for solving **grid networks** with static mini-bucket heuristics. Time limit 1 hour. Number of flips for GLS is 50,000.

improve significantly over BB-C+SMB(i) and AOBB-C+SMB(i), especially at relatively small i-bounds. For example, on the ped37 linkage instance, AOBB-C+GLS+SMB(12) achieves almost an order of magnitude speedup over AOBB-C+SMB(12). Similarly, BB-C+GLS+SMB(12) finds the optimal solution to ped37 in about 35 minutes, whereas BB-C+SMB(12) exceeds the 3 hour time limit.

Figures 4.14 and 4.15 show the search trace of the AND/OR Branch-and-Bound algorithms for solving selected instances of grid networks and UAI'06 Dataset, respectively. We see again that AOBB-C+GLS+SMB(i) and BB-C+GLS+SMB(i) take advantage of the quality of the initial lower bound produced by GLS, and find close to optimal solutions

minfill pseudo tree											
bn (w*, h) (n, d)	SamIam GLS	BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=17		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=18		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=19		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=20		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=21	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	-	-	-	-	-	-
BN_31		-	-	-	-	-	-	-	-	-	-
(46, 160)	out	411.33	1,445,200	486.47	2,131,977	209.80	831,431	210.81	889,782	81.61	94,507
(1156, 2)	9.86*	357.86	1,172,122	375.05	1,573,677	202.66	775,258	187.34	752,284	79.01	56,409
		140.41	293,445	126.23	292,293	85.69	142,650	86.00	114,046	**73.14**	25,392
		-	-	-	-	-	-	-	-	-	-
BN_33		-	-	-	-	-	-	-	-	-	-
(43, 163)	-	429.02	982,130	125.78	210,552	236.42	408,855	160.61	256,191	120.33	89,308
(1444, 2)	12.30*	434.97	980,701	134.47	207,658	244.72	399,206	167.39	245,144	129.35	85,745
		75.92	142,932	**41.14**	41,865	58.14	61,064	73.20	49,760	95.16	22,256
		-	-	-	-	-	-	-	-	-	-
BN_35		-	-	-	-	-	-	-	-	-	-
(41, 168)	-	42.95	126,215	107.17	243,533	81.59	151,632	56.11	65,657	78.27	58,973
(1444, 2)	12.38	49.97	120,205	112.42	224,908	89.85	151,619	66.16	74,585	89.31	71,614
		29.77	29,837	36.58	34,987	43.28	28,088	51.28	15,953	76.28	18,048
		-	-	-	-	-	-	-	-	-	-
BN_37		-	-	-	-	-	-	-	-	-	-
(45, 159)	-	26.42	55,571	20.19	33,475	25.45	14,703	45.61	8,815	94.55	16,400
(1444, 2)	12.70	29.77	48,211	26.17	31,674	32.11	13,808	49.63	7,774	99.00	19,871
		15.83	15,399	19.47	11,046	26.55	6,621	46.84	4,315	90.66	5,610
		-	-	-	-	-	-	-	-	-	-
BN_39		-	-	-	-	-	-	-	-	-	-
(48, 164)	-	1161.65	2,615,679	1370.21	3,448,072	507.18	1,499,020	403.07	1,043,378	220.74	518,011
(1444, 2)	12.88	472.36	1,076,698	782.69	2,026,535	276.27	778,118	190.16	436,932	113.67	168,410
		117.03	340,362	247.08	725,738	131.44	316,862	112.27	213,676	**111.20**	127,872
		-	-	-	-	-	-	-	-	-	-
BN_41		-	-	-	-	-	-	-	-	-	-
(49, 164)	-	56.72	119,737	47.30	77,653	33.81	32,774	50.81	38,467	76.42	31,763
(1444, 2)	12.29*	63.16	117,948	52.52	73,947	40.45	30,930	58.53	37,018	86.72	30,487
		23.50	42,795	**22.05**	20,485	27.22	12,030	43.38	16,549	71.61	11,648

Table 4.16: CPU time and nodes visited for solving **UAI'06 networks** with static mini-bucket heuristics. Time limit 30 minutes. Number of flips for GLS is 500,000.

much earlier than AOBB-C+SMB(i) and BB-C+SMB(i), respectively.

Tables 4.15, 4.16, and 4.17 report detailed results for AOBB-C+GLS+SMB(i) and BB-C+GLS+SMB(i) on grid networks and UAI'06 Dataset networks, respectively. We see that the lower bound computed by GLS was in many cases equal to the optimal solution and therefore AOBB-C+GLS+SMB(i) and BB-C+GLS+SMB(i) improved considerably over AOBB-C+SMB(i) and BB-C+SMB(i), respectively.

4.5.4 The Impact of Determinism in Bayesian Networks

In general, when the functions of the graphical model express both hard constraints and general cost functions, it is beneficial to exploit the computational power of the constraints explicitly via constraint propagation as described in Chapter 3.

We evaluated the AND/OR Branch-and-Bound algorithm with static mini-bucket heuris-

minfill pseudo tree											
bn (w*, h) (n, d)	SamIam GLS	BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=17		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=18		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=19		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=20		BB-C+SMB(i) BB-C+GLS+SMB(i) AOBB-C+SMB(i) AOBB-C+GLS+SMB(i) AOBF-C+SMB(i) i=21	
		time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
		301.56	2,085,673	823.32	6,662,948	512.27	3,189,855	55.16	257,866	54.39	70,027
BN_126		**9.83**	63,674	15.78	85,215	19.31	76,346	27.69	37,226	51.38	30,317
(54, 70)	-	255.27	2,324,776	816.71	8,423,064	92.85	829,994	47.67	244,943	55.27	99,056
(512, 2)	6.08*	10.91	83,227	17.74	117,859	20.66	99,518	28.66	49,175	54.28	42,873
		16.22	64,202	25.64	85,148	26.88	76,645	30.95	37,666	54.59	30,197
		-	-	-	-	-	-	-	-	130.27	631,093
BN_127		**26.44**	238,020	31.02	250,746	36.19	215,054	44.34	166,176	57.52	83,380
(57, 74)	out	-	-	-	-	-	-	-	-	128.94	860,026
(512, 2)	5.75*	27.59	282,349	31.11	295,100	38.67	280,166	46.03	214,590	57.47	113,743
		51.80	223,327	58.63	251,134	62.75	215,796	64.28	166,741	66.35	84,007
		4.14	2,558	7.59	1,266	14.84	10,244	28.31	471	49.13	3,147
BN_128		4.50	854	8.05	694	14.17	778	29.44	461	48.75	551
(48, 73)	out	4.11	3,636	7.48	1,411	15.00	12,034	28.38	552	49.33	4,203
(512, 2)	5.95*	4.14	1,022	7.91	974	13.92	991	28.75	547	49.64	674
		3.97	883	7.75	925	13.78	808	28.39	478	49.13	575
		-	-	-	-	176.24	1,603,304	1337.90	11,794,805	257.42	1,855,134
BN_129		244.08	2,419,418	150.30	1,408,350	150.56	1,352,916	119.70	923,635	142.14	914,833
(52, 68)	out	573.74	5,730,592	-	-	167.14	1,688,675	1388.01	13,437,762	219.09	1,747,613
(512, 2)	5.89*	245.08	2,443,843	95.64	961,434	142.55	1,412,079	**76.16**	564,895	138.53	979,046
		out		194.56	922,831	out		132.45	537,371	246.39	910,769
		21.56	182,120	-	-	869.44	7,310,190	-	-	57.06	109,669
BN_130		**14.55**	114,610	87.28	751,400	41.73	299,845	42.86	158,612	58.53	107,880
(54, 67)	out	22.49	239,771	-	-	863.15	8,414,475	-	-	58.94	147,085
(512, 2)	5.87*	15.36	158,150	36.24	364,352	43.25	392,961	43.19	211,380	57.91	144,741
		27.72	115,091	68.53	273,987	76.53	299,172	60.55	158,650	69.63	107,771
		17.06	137,631	39.02	323,431	1149.74	10,230,128	47.25	228,703	-	-
BN_131		**15.42**	118,238	26.77	212,338	19.56	82,414	28.69	73,552	51.69	122,085
(48, 72)	out	18.69	176,456	41.63	396,234	1254.88	12,395,143	50.42	303,818	-	-
(512, 2)	5.87*	16.70	150,341	28.22	256,361	20.34	101,662	29.16	91,103	54.12	156,925
		29.03	116,166	50.13	209,748	28.47	79,689	36.89	73,163	65.74	120,153
		-	-	-	-	-	-	756.69	6,584,446	578.99	4,819,402
BN_132		683.65	5,987,145	429.96	3,750,177	838.83	7,484,051	627.50	5,584,689	392.78	3,296,711
(49, 71)	out	-	-	-	-	-	-	778.22	7,456,812	643.96	6,037,908
(512, 2)	5.89*	686.08	6,499,878	439.89	4,252,274	718.66	6,905,710	453.25	4,319,442	**387.02**	3,557,198
		out		out		out		out		out	
		-	-	**16.84**	104,521	31.28	171,645	127.32	929,016	55.33	30,699
BN_133		29.13	258,988	17.09	102,193	22.77	93,433	36.28	90,006	53.97	17,865
(49, 71)	out	-	-	17.19	133,794	31.64	202,954	135.60	1,184,600	55.22	40,483
(512, 2)	5.79*	30.50	329,146	16.50	125,945	22.66	116,553	36.17	112,317	53.92	17,069
		59.61	258,891	27.44	98,148	32.91	93,613	45.09	90,337	55.31	13,491
		-	-	-	-	-	-	-	-	-	-
BN_134		105.61	1,029,072	**43.16**	373,641	115.67	1,065,258	60.94	376,402	75.16	213,954
(52, 70)	out	-	-	-	-	-	-	-	-	-	-
(512, 2)	5.83*	109.97	1,170,028	44.33	439,065	123.91	1,253,376	60.72	401,521	76.38	241,382
		out		85.77	373,081	out		96.19	377,064	97.59	214,591

Table 4.17: CPU time and nodes visited for solving **UAI'06 networks** with static mini-bucket heuristics. Time limit 30 minutes. Number of flips for GLS is 500,000.

minfill pseudo tree									
grid (w*, h) (n, e)	SamIam GLS	AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=12		AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=14		AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=16		AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=18	
		time	nodes	time	nodes	time	nodes	time	nodes
		-	-	1273.09	9,047,518	596.27	4,923,760	70.42	473,675
90-24-1	out	687.96	4,823,044	202.05	1,564,800	172.31	1,370,222	55.52	401,294
(33, 111)		-	-	66.20	425,585	20.16	93,911	11.17	7,850
(576, 20)	0.53	473.64	3,181,352	19.09	131,546	8.41	49,054	**5.45**	6,891
		out		21.94	75,637	10.59	33,770	6.06	5,144
		146.97	878,874	152.80	962,484	4.36	15,632	12.92	46,489
90-26-1	out	32.67	230,030	53.11	360,612	**3.58**	11,620	11.95	40,075
(36, 113)		36.94	252,380	87.02	559,518	4.17	14,580	7.86	6,310
(676, 40)	0.56	15.09	104,775	32.85	219,037	**3.58**	10,932	8.06	8,128
		19.06	65,271	24.39	79,619	4.27	7,190	8.05	3,777
		652.15	3,882,300	165.74	1,070,823	155.20	956,837	40.14	212,963
90-30-1	out	117.25	771,233	66.66	453,095	50.94	341,670	30.69	168,928
(43, 150)		263.32	1,498,756	74.95	446,498	68.16	376,916	23.88	95,136
(900, 60)	0.72	89.94	561,397	38.92	247,271	28.67	176,330	**15.50**	52,260
		158.97	534,385	46.73	157,187	47.27	154,496	21.06	45,201
		-	-	-	-	-	-	-	-
90-34-1	out	-	-	-	-	-	-	-	-
(45, 153)		-	-	-	-	1096.14	5,569,276	1772.51	5,516,888
(1154, 80)	1.31	-	-	-	-	550.55	2,944,055	651.04	2,614,171
		out		out		**243.63**	596,978	270.88	667,013
		969.02	2,623,971	1753.10	3,794,053	203.67	614,868	165.45	488,873
90-38-1	out	141.89	577,763	204.69	593,809	86.16	319,185	102.03	312,473
(47, 163)		854.61	2,498,702	1822.71	3,792,826	212.63	647,089	164.43	484,815
(1444, 120)	1.11	138.44	573,923	204.68	597,751	96.27	339,729	98.21	311,072
		101.69	174,786	103.80	146,237	54.00	95,511	**53.44**	78,431

Table 4.18: CPU time and nodes visited for solving **deterministic grid networks** with static mini-bucket heuristics. Number of flips for GLS is 50,000. Time limit 1 hour.

tics on selected classes of Bayesian networks containing deterministic conditional probability tables ($i.e.$, zero probability tuples). The algorithm, denoted by `AOBB-C+SAT+SMB(`i`)` exploits the determinism present in the networks by applying unit resolution over the CNF encoding of the zero-probability tuples, at each node in the search tree. We used a unit resolution scheme similar to the one employed by `zChaff`, a the state-of-the-art SAT solver introduced by [94]. We also consider the extension called `AOBB-C+SAT+GLS+SMB(`i`)` which uses GLS to compute the initial lower bound, in addition to the constraint propagation scheme.

Table 4.18 shows the results for 5 deterministic grid networks from Section 4.5.2. These networks have a high degree of determinism encoded in their CPTs. We observe that `AOBB-C+SAT+SMB(`i`)` improves significantly over `AOBB-C+SMB(`i`)`, especially at small i-bounds. On grid `90-30-1`, for example, `AOBB-C+SAT+SMB(12)` is 6 times faster than `AOBB-C+SMB(12)`. As the i-bound increases and the search space is pruned more effectively, the difference between `AOBB-C+SMB(`i`)` and `AOBB-C+SAT+SMB(`i`)` de-

minfill pseudo tree									
iscas89 (w*, h) (n, d)	SamIam GLS	AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=6		AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=8		AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=10		AOBB-C+SMB(i) AOBB-C+SAT+SMB(i) AOBB-C+GLS+SMB(i) AOBB-C+SAT+GLS+SMB(i) AOBF-C+SMB(i) i=12	
		time	nodes	time	nodes	time	nodes	time	nodes
		-	-	-	-	182.53	2,316,024	0.16	432
c432	out	374.29	4,336,403	189.13	2,043,475	1.02	9,512	0.16	432
(27, 45)		**0.05**	0	0.06	0	0.09	0	0.13	0
(432, 2)	0.08*	0.06	0	0.08	0	0.09	0	0.13	0
		out		out		106.27	488,462	0.20	432
		899.63	7,715,133	17.99	155,865	48.13	417,924	17.00	132,139
s953	out	0.19	829	0.16	667	0.20	685	0.31	623
(66, 101)		**0.12**	0	0.13	0	0.17	0	0.28	0
(440, 2)	0.05*	0.13	0	0.13	0	0.17	0	0.30	0
		out		41.03	150,598	110.45	408,828	36.50	113,322
		18.05	104,316	124.53	686,069	3.69	26,847	14.23	94,985
s1196	out	0.19	565	0.19	565	0.23	565	0.38	565
(54, 97)		0.14	0	0.16	0	0.20	0	0.34	0
(560, 2)	0.08*	**0.13**	0	0.14	0	0.20	0	0.34	0
		26.16	77,019	158.19	372,129	7.22	23,348	26.97	80,264
		13.22	82,294	1.02	5,920	2.50	15,621	1.19	6,024
s1488	out	0.20	708	0.20	667	0.25	667	0.44	667
(47, 67)		0.14	0	0.16	0	0.22	0	0.44	0
(667, 2)	0.13*	**0.13**	0	0.16	0	0.20	0	0.47	0
		21.75	74,658	1.67	5,499	4.22	14,445	1.84	5,372
		7.30	41,798	19.69	108,768	4.81	27,711	7.00	41,977
s1494	out	0.20	665	0.22	665	0.27	665	0.45	665
(48, 69)		**0.16**	0	0.17	0	0.22	0	0.41	0
(661, 2)	0.11*	**0.16**	0	0.17	0	0.22	0	0.42	0
		9.67	24,849	27.28	65,859	7.86	19,678	11.48	28,793

Table 4.19: CPU time and nodes visited for solving belief networks derived from **ISCAS'89 circuits** using static mini-bucket heuristics. Time limit 30 minutes.

creases because the heuristics are strong enough to cut the search space significantly. The mini-bucket heuristic already does some level of constraint propagation.

When looking at impact of the initial lower bound on AOBB-C+SAT+SMB(i) we see that AOBB-C+SAT+GLS+SMB(i) is sometimes able to improve even more. For example, on the 90-34-1 grid, AOBB-C+SAT+GLS+SMB(16) finds the optimal solution in about 9 minutes (550.55 seconds) whereas AOBB-C+SAT+SMB(16) exceeds the time limit.

Table 4.19 shows the results for experiments with 5 belief networks derived from ISCAS'89 circuits. We see that constraint propagation via unit resolution plays a dramatic role on this domain, rendering the search space almost backtrack-free across i-bounds. For instance, on the s953, AOBB-C+SAT+SMB(6) is 3 orders of magnitude faster than AOBB-C+SMB(6), while AOBF-C+SMB(6) exceeded the memory limit. When looking at the AND/OR Branch-and-Bound algorithms that exploit the local search initial lower bound, namely AOBB-C+GLS+SMB(i) and AOBB-C+SAT+GLS+SMB(i), we see that they did not expand any nodes. This is because the lower bound obtained by GLS, which

was the optimal solution in this case, was equal to the mini-bucket upper bound computed at the root node. Here, the best performance was achieved by AOBB-C+SAT+SMB(i) and AOBB-C+SAT+GLS+SMB(i), respectively, for the smallest reported i-bound (namely $i = 6$). Notice also the poor performance of SAMIAM which ran out of memory on all tests.

4.5.5 Results for Empirical Evaluation of Weighted CSPs

In Chapter 3 we showed that the best performance on this domain was obtained by the AND/OR Branch-and-Bound tree search algorithm with static mini-bucket heuristics, at relatively large i-bounds, especially on non-binary WCSPs with relatively small domain sizes ($e.g.$, SPOT5 networks, ISCAS'89 circuits, Mastermind instances). AOBB+SMB(i) dominated all its competitors, including the classic OR Branch-and-Bound BB+SMB(i) as well as the OR and AND/OR algorithms that enforce EDAC during search, namely toolbar and the AOEDAC family of algorithms (AOEDAC+PVO, DVO+AOEDAC and AOEDAC+DSO, respectively). The AND/OR Branch-and-Bound with dynamic mini-bucket heuristics AOBB+SMB(i) was shown to be competitive only for relatively small i-bounds. In this section we continue the evaluation of the AND/OR algorithms with mini-bucket heuristics, focusing on memory intensive depth-first and best-first search strategies.

Earth Observing Satellites

SPOT5 benchmark contains a collection of large real scheduling problems for the daily management of Earth observing satellites [7]. They can be easily formulated as WCSPs with binary and ternary constraints, as described in Chapter 3.

Tables 4.20 and 4.21 show detailed results on experiments with 7 SPOT5 networks using min-fill pseudo trees, as well as static and dynamic mini-bucket heuristics. The networks 42b, 408b and 505b are sub-networks of the original ones and contain only binary constraints.

minfill pseudo tree										
spot5 (w*, h) (n, k, c)	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF+SMB(i) i=4		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF+SMB(i) i=6		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF+SMB(i) i=8		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF+SMB(i) i=12		toolbar toolbar-BTD AOEDAC+PVO DVO+AOEDAC AOEDAC+DSO	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	0.01		0.05		0.33		21.66		4.56	218,846
29	-	-	-	-	6313.73	50,150,302	22.30	2,322	**0.35**	984
(14, 42)	8.77	86,058	5.05	45,509	0.66	2,738	22.02	246	545.43	7,837,447
(83, 4, 476)	5.53	48,995	3.66	29,702	0.56	2,267	21.67	110	0.81	8,698
	6.42	36,396	2.23	12,801	0.47	757	21.77	96	11.36	92,970
	0.11		0.17		0.56		28.83		-	-
42b	-	-	-	-	2159.26	9,598,763	145.77	684,109	9553.06	249,053,196
(18, 62)	-	-	-	-	1842.32	9,606,846	134.39	689,402	-	-
(191, 4, 1341)	-	-	-	-	1804.76	9,410,729	116.98	584,838	-	-
	35.42	118,085	29.11	106,648	**20.80**	82,611	38.91	43,127	6825.40	27,698,614
	0.02		0.03		0.11		1.24		0.31	21,939
54	664.48	5,715,457	2.06	17,787	0.38	2,289	1.27	236	0.18	779
(11, 33)	113.19	1,106,598	1.59	17,757	0.39	3,616	1.27	329	9.11	90,495
(68, 4, 283)	18.42	198,712	0.23	2,477	0.16	591	1.25	120	0.06	688
	0.41	2,714	**0.11**	631	0.16	312	0.69	68	0.75	6,614
	0.01		0.02		0.09		1.11		151.11	6,215,135
404	-	-	-	-	-	-	4336.37	32,723,215	5.09	139,968
(19, 42)	430.99	3,969,398	151.99	1,373,846	14.83	144,535	1.44	3,273	152.81	1,984,747
(100, 4, 710)	174.09	1,396,321	51.88	529,002	2.55	23,565	1.16	598	12.09	88,079
	1.45	7,251	1.20	6,399	**1.02**	5,140	1.22	576	1.74	14,844
	0.01		0.09		0.33		8.37		-	-
408b	-	-	-	-	-	-	-	-	-	-
(24, 59)	-	-	-	-	-	-	715.35	4,784,407	-	-
(201, 4, 1847)	-	-	-	-	7507.10	54,826,929	75.08	408,619	-	-
	208.41	185,935	52.53	175,366	44.99	145,901	**16.97**	39,238	747.71	2,134,472
	0.02		0.05		0.14		0.41		-	-
503	-	-	-	-	-	-	0.50	566	0.65	18,800
(9, 39)	-	-	435.26	5,102,299	421.10	4,990,898	0.44	641	-	-
(144, 4, 639)	-	-	189.39	2,442,998	291.72	4,050,474	**0.42**	256	10005.00	44,495,545
	5.28	16,114	1.56	9,929	1.59	9,186	**0.42**	144	53.72	231,480
	0.05		0.11		0.66		47.19		-	-
505b	-	-	-	-	-	-	-	-	33.62	1,119,538
(16, 98)	-	-	-	-	-	-	-	-	-	-
(240, 1721)	-	-	-	-	-	-	1180.48	8,905,473	-	-
	51.86	149,928	42.73	144,723	**29.25**	111,223	54.09	31,692	-	-

Table 4.20: CPU time in seconds and number of nodes visited for solving the **SPOT5 benchmarks**, using **static mini-bucket heuristics** and min-fill based pseudo trees. Time limit 3 hours.

Tree vs. graph AOBB. We notice again the benefit of using caching within depth-first AND/OR Branch-and-Bound search. The differences, in running time and number of nodes visited, between `AOBB-C+SMB(`i`)` and `AOBB+SMB(`i`)` are more prominent at relatively small i-bounds. For example, on `408b`, `AOBB-C+SMB(12)` outperforms `AOBB+SMB(12)` by 1 order of magnitude in terms of both running time and size of the search space explored. When looking at the impact of caching when using dynamic mini-bucket heuristics (Table 4.21) we see that the difference between `AOBB-C+DMB(`i`)` and `AOBB+DMB(`i`)`, across i-bounds, is not that pronounced as in the static case. This is because the dynamic mini-bucket heuristics are far more accurate than the pre-compiled ones and prune the search space more effectively, thus not leaving room for additional improvements due to caching. Notice that `toolbar` and `DVO+AOEDAC` are able to solve relatively

min-fill pseudo tree								
spot5 (w*, h) (n, k, c)	BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=4		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=6		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=8		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=12	
	time	nodes	time	nodes	time	nodes	time	nodes
29	44.24	11,637	125.72	9,417	54.86	354	627.30	320
(14, 42)	65.24	14,438	52.92	11,850	121.83	364	627.29	330
(83, 4, 476)	56.58	6,017	53.06	4,638	122.17	170	636.16	136
	7.25	942	21.83	537	38.83	114	308.71	83
42b	-	-	-	-	-	-	-	-
(18, 62)	-	-	-	-	-	-	-	-
(191, 4, 1341)	-	-	-	-	-	-	-	-
	1455.62	101,453	-	-	-	-	6002.69	212
54	886.51	118,219	32.59	938	24.97	236	320.81	236
(11, 33)	202.14	69,362	26.73	2,188	22.19	329	271.81	329
(68, 4, 283)	84.27	15,214	8.80	357	10.86	120	137.39	120
	4.16	1,056	**3.66**	163	5.95	68	77.78	68
404	-	-	-	-	4895.25	78,692	3459.31	3,008
(19, 42)	240.36	156,338	257.20	39,144	199.67	5,612	563.02	1,327
(100, 4, 710)	65.52	20,457	98.83	6,152	99.78	952	320.49	286
	23.41	4,928	65.80	2,946	101.30	847	351.37	291
408b	-	-	-	-	-	-	-	-
(24, 59)	-	-	-	-	-	-	-	-
(201, 4, 1847)	-	-	-	-	-	-	-	-
	655.41	70,655	2447.91	69,434	-	-	-	-
503	-	-	-	-	-	-	246.65	566
(9, 39)	-	-	-	-	-	-	64.95	641
(144, 4, 639)	-	-	-	-	-	-	49.95	256
	78.69	9,143	324.09	8,175	1025.40	5,984	**25.14**	144
505b	-	-	-	-	-	-	-	-
(16, 98)	-	-	-	-	-	-	-	-
(240, 1721)	-	-	-	-	-	-	-	-
	681.40	33,969	2766.08	28,157	3653.66	12,455	-	-

Table 4.21: CPU time in seconds and number of nodes visited for solving the **SPOT5 benchmarks**, using **dynamic mini-bucket heuristics** and min-fill based pseudo trees. Time limit 3 hours.

efficiently only the first 3 test instances.

AOBB vs. AOBF. When comparing the best-first against the depth-first AND/OR search algorithms we observe again here that `AOBF-C+SMB(`i`)` improves significantly (up to several orders of magnitude) in terms of both CPU time and number of nodes visited, especially for relatively small i-bounds. For example, on `505b`, one of the hardest instances, `AOBF-C+SMB(8)` finds the optimal solution in less than 30 seconds, whereas `AOBB-C+SMB(8)` exceeds the 3 hour time limit. As the mini-bucket i-bound increases and the heuristics become strong enough to cut the search space substantially, the difference between Branch-and-Bound and best-first search decreases, because Branch-and-Bound finds almost optimal solutions fast, and therefore will not explore solutions whose cost is above the optimal one, like best-first search. Notice that `toolbar-BTD` fails only on one instance, namely `408b`, and is competitive with `AOBF-C+SMB(`i`)` on 4 test instances,

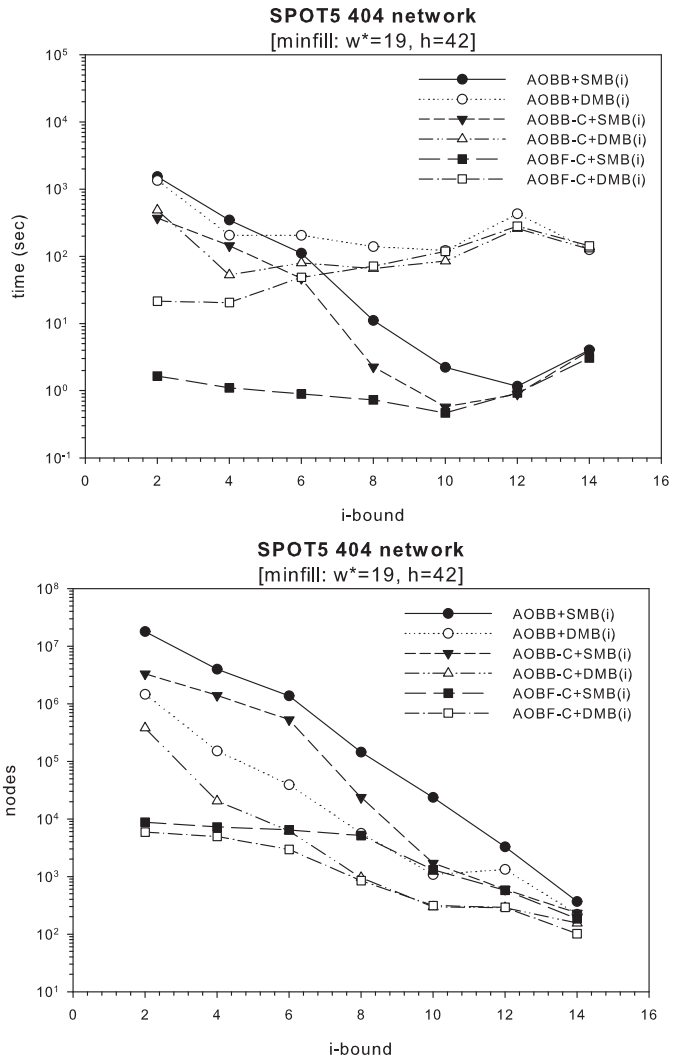


Figure 4.16: Comparison of the impact of static and dynamic mini-bucket heuristics on the **404 SPOT5 network** from Tables 4.20 and 4.21. We show CPU time (top) and number of nodes (bottom).

namely on 29, 54, 503 and 505b.

Static vs. dynamic mini-bucket heuristics. Figure 4.16 displays the running time and number of nodes visited by the AND/OR search algorithms with static and dynamic mini-bucket heuristics, as a function of the i-bound, on the 404 network ($i.e.$, corresponding to the fourth horizontal block from Tables 4.20 and 4.21, respectively). We see that the power of the dynamic mini-bucket heuristics is again more prominent for small i-bounds ($e.g.$, $i = 2$). At larger i-bounds, the static mini-bucket heuristics are cost effective. For instance, the the difference in running time between AOBB-C+SMB(10) and AOBB-C+DMB(10) is about 2 orders of magnitude. Notice that in this case, AOBF-C+SMB(i) outperforms AOBF-C+DMB(i) across all reported i-bounds.

Impact of the pseudo tree. Figure 4.17 plots the runtime distribution of AOBB-C+SMB(i) and AOBF-C+SMB(i) using hypergraph based pseudo trees, over 20 independent runs. We see that the hypergraph based pseudo trees are sometimes able to improve the performance of Branch-and-Bound search, especially for relatively small i-bounds ($e.g.$, 404, 503) for which the heuristic estimates are less accurate. For best-first search however, the min-fill pseudo trees offer the overall best performance, because the mini-bucket heuristics computed along this ordering, rather than the hypergraph based one, are relatively accurate thus bounding the horizon of best-first search more effectively.

ISCAS'89 Benchmark Circuits

Tables 4.22 and 4.23 report the results for experiments with 10 WCSPs derived from ISCAS'89 circuits as described in Chapter 3, using static and dynamic mini-bucket heuristics as well as min-fill based pseudo trees.

Tree vs. graph AOBB. When comparing the tree versus the graph AND/OR Branch-and-Bound search algorithms, we see again the benefit of caching, especially when using pre-

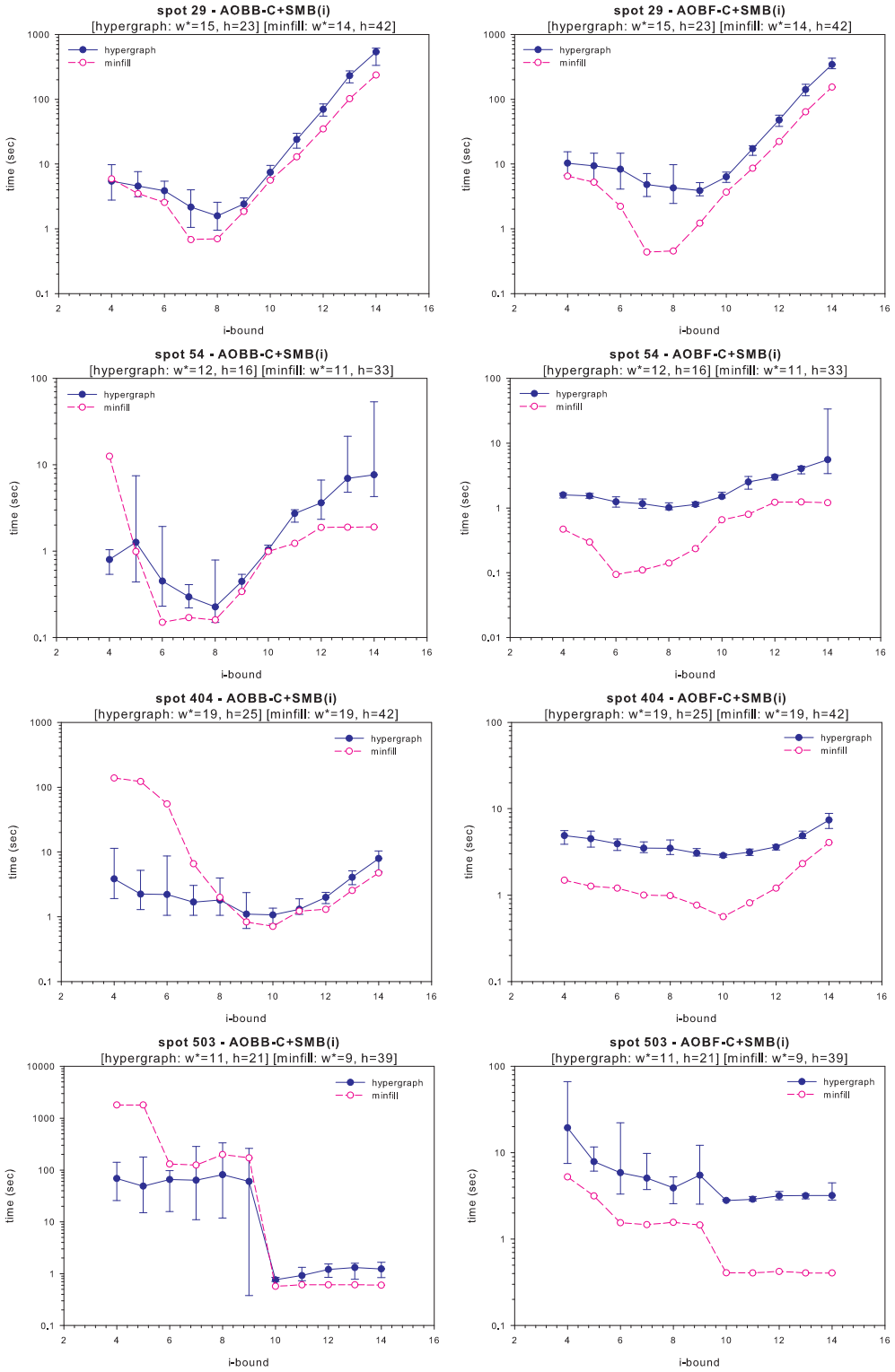


Figure 4.17: Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving SPOT5 networks with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

minfill pseudo tree										
iscas89 (w*, h) (n, d)	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF+SMB(i) i=8		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF+SMB(i) i=10		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF+SMB(i) i=14		toolbar toolbar-BTD	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	0.08		0.09		0.14		0.22			
c432	-	-	-	-	9.27	52,778	9.17	52,240	-	-
(27, 45)	2010.53	23,355,897	148.39	1,713,265	5.94	76,346	5.84	75,420	-	-
(432, 2)	422.08	2,945,230	40.91	337,574	0.89	6,254	0.89	6,010		
	39.33	196,892	0.52	2,154	**0.31**	1,007	0.38	847		
	0.08		0.08		0.14		0.28			
c499	-	-	-	-	1.53	4,495	6.20	35,314	-	-
(23, 55)	96.46	1,265,425	39.65	526,517	1.42	18,851	37.26	486,656	100.96	1,203,734
(499, 2)	19.28	99,906	7.36	40,285	0.47	2,401	5.83	34,708		
	3.91	14,049	2.45	8,816	**0.34**	1,032	2.52	8,755		
	0.16		0.19		0.22		0.45			
c880	-	-	-	-	-	-	-	-	-	-
(27, 67)	1698.08	19,992,512	1316.73	15,247,946	505.75	5,835,825	1134.61	13,568,696	-	-
(881, 2)	100.66	516,056	91.66	446,893	31.06	169,138	59.35	316,124		
	1.36	4,454	0.91	2,792	**0.81**	2,231	1.19	2,862		
	0.02		0.03		0.06		0.14			
s386	0.33	2,015	0.33	2,281	0.30	1,734	0.31	1,191	-	-
(19, 44)	0.14	2,073	0.33	4,867	0.22	2,699	0.22	1,420	0.19	738
(172, 2)	0.06	592	0.17	1,334	0.12	755	0.16	446		
	0.05	187	0.08	304	0.08	203	0.16	172		
	0.13		0.17		0.30		0.73			
s935	-	-	-	-	-	-	-	-	-	-
(66, 101)	2559.30	21,438,706	342.80	3,074,516	-	-	41.34	348,699	1.51	11,368
(441, 2)	1285.07	6,623,608	143.53	763,933	-	-	22.28	128,372		
	6.16	25,493	1.22	4,087	**1.19**	3,319	1.22	2,216		
	0.16		0.19		0.38		0.94			
s1196	-	-	-	-	-	-	-	-	-	-
(54, 97)	-	-	1347.95	12,392,442	-	-	1949.37	15,775,180	376.35	1,276,514
(562, 2)	3347.38	13,554,137	503.30	2,425,152	2299.72	11,488,366	734.66	3,524,780		
	22.67	72,075	**2.89**	9,336	13.02	40,210	7.27	21,989		
	0.16		0.22		0.38		0.92			
s1238	-	-	-	-	-	-	-	-	-	-
(59, 94)	-	-	-	-	1722.53	18,302,873	1394.86	14,213,319	-	-
(541, 2)	1897.37	8,386,634	1682.99	7,431,223	281.05	1,350,933	248.27	1,220,658		
	34.09	137,960	29.41	111,205	12.31	53,095	**6.64**	26,101		
	0.12		0.14		0.17		0.31			
s1423	-	-	-	-	-	-	-	-	-	-
(19, 44)	71.63	648,520	25.58	228,634	7.56	68,102	7.92	70,043	-	-
(749, 2)	7.61	37,244	2.75	11,423	1.48	7,164	1.39	5,868		
	1.16	3,873	0.70	2,193	**0.53**	1,683	0.69	1,663		
	0.16		0.24		0.41		1.05			
s1488	-	-	-	-	10.75	23,620	13.75	25,420	-	-
(47, 67)	6.67	50,613	46.83	430,141	4.00	29,729	5.19	33,827	1.80	9,315
(667, 2)	3.33	15,998	13.14	45,560	2.22	9,337	3.11	10,640		
	0.36	778	0.41	724	0.56	688	1.22	710		
	0.19		0.25		0.45		1.16			
s1494	-	-	191.36	366,822	52.47	140,792	19.86	44,190	-	-
(48, 69)	132.62	833,720	17.70	455,131	376.65	3,207,718	15.49	83,318	2.41	12,122
(661, 2)	62.87	127,934	5.64	17,279	27.64	80,895	6.92	23,131		
	1.44	5,694	**0.59**	1,472	0.95	2,311	1.50	1,476		

Table 4.22: CPU time in seconds and number of nodes visited for solving **ISCAS'89 circuits**, using **static mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour.

min-fill pseudo tree								
iscas89 (w*, h) (n, d)	BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=8		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=10		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=12		BB-C+DMB(i) AOBB+DMB(i) AOBB-C+DMB(i) AOBF-C+DMB(i) i=14	
	time	nodes	time	nodes	time	nodes	time	nodes
c432	403.44	33,506	191.69	14,303	13.56	1,026	8.86	627
(27, 45)	45.59	34,904	25.83	16,482	6.94	1,070	4.55	692
(432, 2)	35.19	3,861	20.69	2,302	6.69	860	4.53	627
	1.53	448	2.28	444	3.02	434	4.92	432
c499	40.99	3,502	31.85	3,102	9.91	987	42.66	2,848
(23, 55)	26.13	13,529	14.44	6,101	4.33	1,002	25.91	3,353
(499, 2)	24.44	2,485	13.42	1,726	4.28	742	25.25	1,251
	1.39	931	2.25	579	3.73	541	9.08	499
c880	-	-	-	-	547.35	18,112	648.52	19,546
(27, 67)	1078.04	796,699	762.16	569,471	85.64	32,748	170.55	36,187
(881, 2)	786.49	31,788	560.80	16,546	68.36	2,486	153.36	2,736
	8.77	1,378	9.94	1,304	**7.28**	956	16.83	958
s386	2.58	1,191	2.91	1,191	3.41	1,191	4.28	1,191
(19, 44)	0.81	1,420	1.14	1,420	1.61	1,420	2.52	1,420
(172, 2)	0.69	446	1.02	446	1.53	446	2.44	446
	0.30	172	0.50	172	0.86	172	1.53	172
s935	49.27	6,217	264.99	9,028	301.39	7,842	957.57	8,080
(66, 101)	18.27	7,400	234.47	10,250	267.02	9,164	915.57	11,164
(441, 2)	16.55	1,568	228.71	3,682	263.58	2,279	903.12	2,528
	5.47	479	23.87	553	27.19	454	140.51	490
s1196	233.39	18,040	335.50	15,525	670.04	13,677	1362.32	11,939
(54, 97)	61.64	21,849	114.16	17,524	246.02	15,443	921.08	13,687
(562, 2)	50.80	3,787	97.53	3,160	217.97	2,888	857.35	2,772
	6.80	688	11.58	586	32.11	635	102.45	632
s1238	784.04	34,905	521.27	15,685	1395.39	17,852	2021.31	11,264
(59, 94)	266.45	39,493	188.83	21,252	566.96	20,945	913.24	13,857
(541, 2)	242.16	8,792	174.80	4,265	544.35	4,511	887.65	3,078
	18.69	827	22.47	666	57.59	591	192.10	632
s1423	-	-	71.39	3,629	134.36	8,132	62.39	3,045
(19, 44)	38.36	26,772	35.02	17,801	36.19	19,719	22.27	3,513
(749, 2)	28.97	3,078	28.64	2,492	30.31	2,361	22.08	1,477
	5.97	1,191	6.25	1,141	9.48	1,126	12.39	762
s1488	146.03	14,365	139.83	12,475	181.58	12,748	306.35	12,748
(47, 67)	20.64	15,064	31.34	13,279	67.78	13,762	193.88	13,762
(667, 2)	18.33	2,824	29.20	2,634	65.34	2,576	190.94	2,576
	2.86	670	5.61	668	13.80	667	41.81	667
s1494	276.49	23,931	267.91	21,032	246.30	14,898	228.83	9,465
(48, 69)	71.52	25,104	84.92	22,082	112.49	15,698	151.00	9,706
(661, 2)	66.25	4,794	78.97	4,018	110.36	3,059	149.30	2,386
	10.42	758	**9.88**	679	20.38	667	58.75	666

Table 4.23: CPU time in seconds and number of nodes visited for solving **ISCAS'89 circuits**, using **dynamic mini-bucket heuristics** and min-fill based pseudo trees. Time limit 1 hour.

compiled mini-bucket heuristics. For example, on the `s1238` circuit, `AOBB-C+SMB(12)` is 6 times faster than `AOBB+SMB(12)` and explored 14 times fewer nodes. The diference between the tree and graph AND/OR algorithms is not too prominent when using dynamic mini-bucket heuristics (Table 4.23), because these heuristics are far more accurate than the static version and the search graph is very close to a tree in this case. The performance of `toolbar` that is designed specifically for the WCSP domain was very poor on this dataset and it was not able to solve the any of the problem instances within the 1 hour time limit. On the other hand, `toolbar-BTD`, which traverses and AND/OR search graph, is competitive on this dataset and solves 6 out of the 10 test instances.

AOBB vs. AOBF. When comparing the depth-first versus the best-first AND/OR algorithms with static and dynamic mini-bucket heuristics we see again that AOBF-C+SMB(i) outperforms significantly AOBB-C+SMB(i), especially for relatively small i-bounds. The same picture can be observed when comparing AOBF-C+DMB(i) with AOBB-C+DMB(i). For instance, on the s1196 circuit, AOBF-C+SMB(10) is 2 orders of magnitude faster than AOBB-C+SMB(10). Similarly, on the s1238 circuit, AOBF-C+DMB(8) outperforms AOBB-C+DMB(8) by one order of magnitude in terms of both running time and size of the search space explored. Overall, AOBF-C+SMB(i) is the best performing algorithm on this dataset.

Static vs. dynamic mini-bucket heuristics. Figure 4.18 plots the CPU time and number of nodes visited by the AND/OR algorithms with static and dynamic mini-bucket heuristics, as a function of the i-bound, on the c880 network from Tables 4.22 and 4.23, respectively. It shows explicitly how the performance of Branch-and-Bound and best-first search changes with the mini-bucket i-bound. Focusing for example on AOBF-C+SMB(i) we notice again the U-shaped curve formed by the running time. At small i-bounds ($i = 4$) the time is high, then as i increases the runnning time decreases ($e.g.$, for $i = 12$ the time is 0.91), but then as i increases further the time starts to increase again. The same behavior can be observed for AOBF-C+DMB(i), as well.

Impact of the level of caching. Figure 4.19 displays the CPU time, as a function of the cache bound j, on 4 ISCAS'89 networks from Tables 4.22 using AOBB-C+SMB(i,j) (naive caching) and AOBB-AC+SMB(i,j) (adaptive caching), respectively. The spectrum of results is similar to what we observed before. Namely, adaptive caching is more powerful than naive caching at smaller j bounds. As the cache bound increases, the two schemes approach full caching.

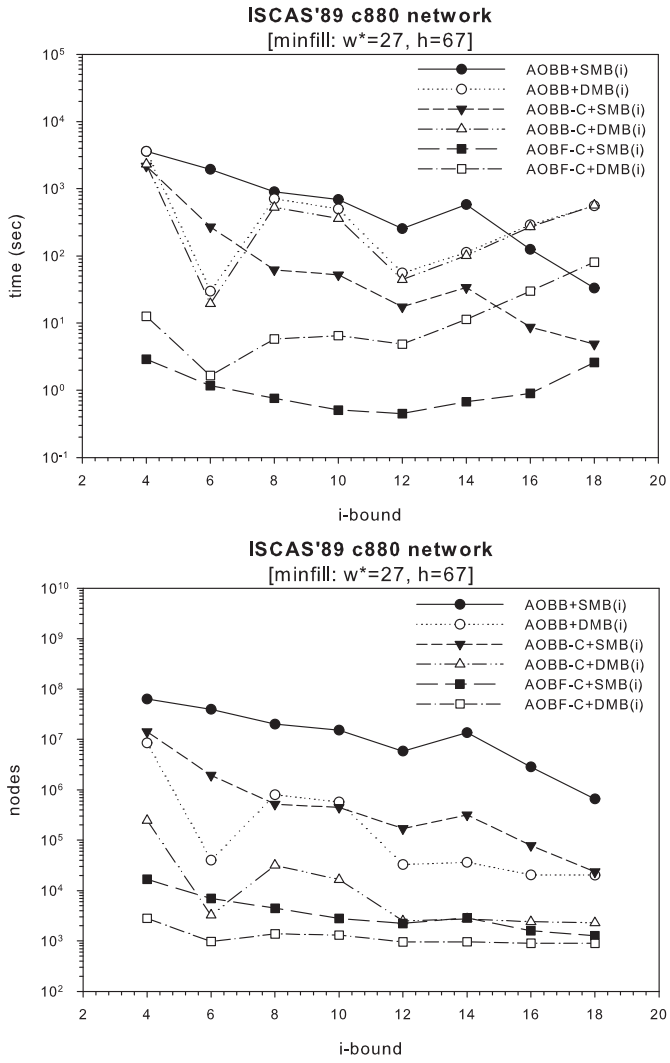


Figure 4.18: Comparison of the impact of static and dynamic mini-bucket heuristics on the **`c880` ISCAS'89 network** from Tables 4.22 and 4.23, respectively.

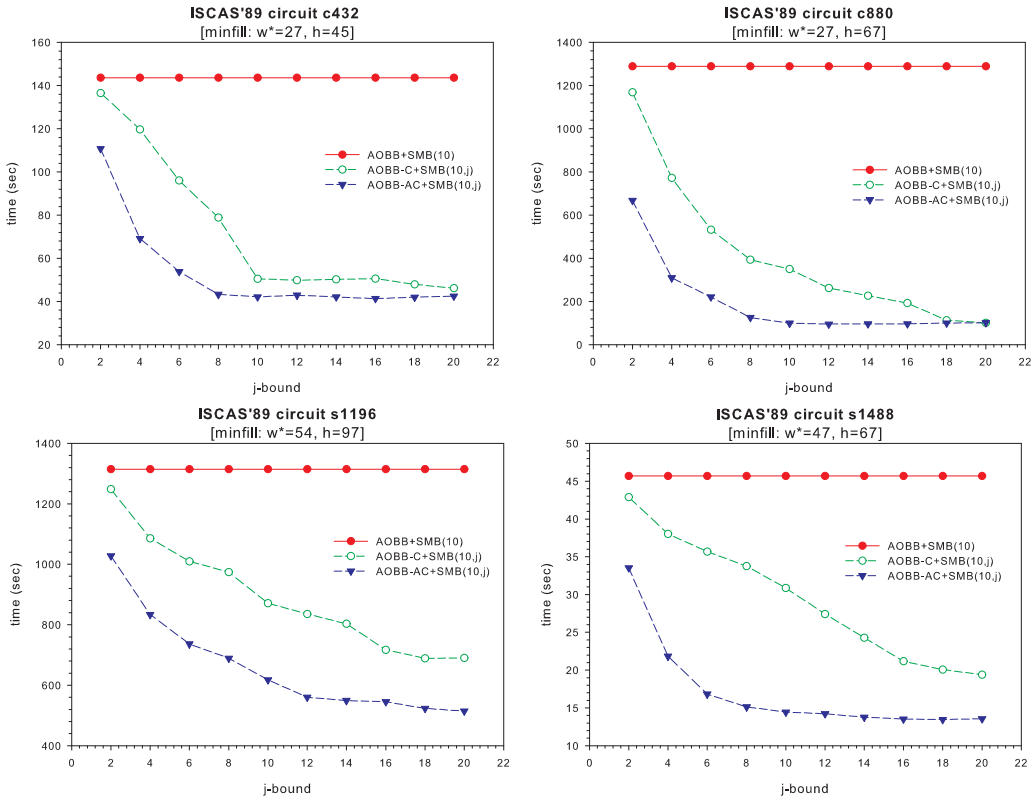


Figure 4.19: Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **ISCAS'89 circuits**. Shown is CPU time in seconds.

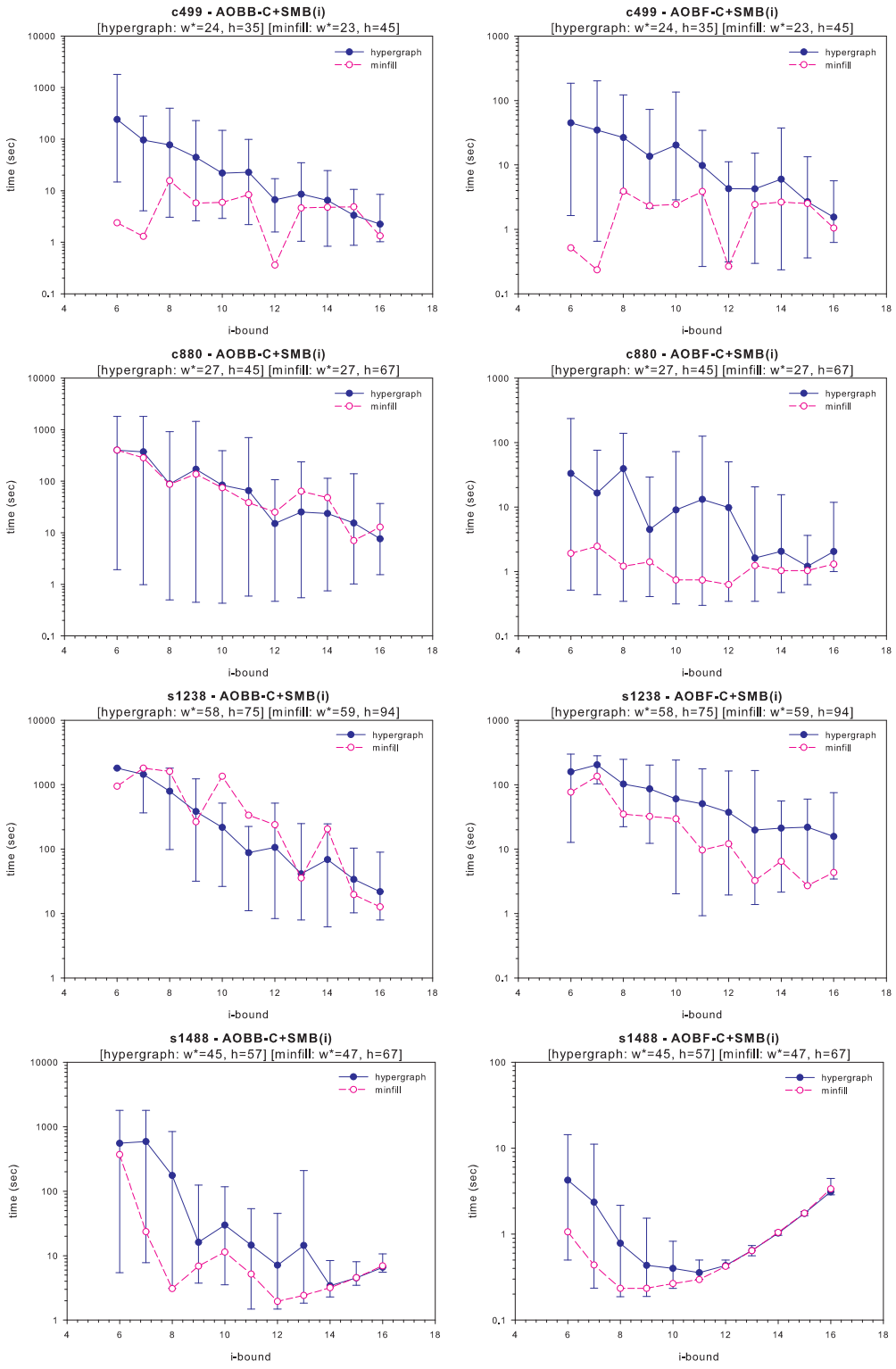


Figure 4.20: Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **ISCAS'89 networks** with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

minfill pseudo trees												
mastermind (w*, h) (n, r, k)	MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=8		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=10		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=12		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=14		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=16		MBE(i) BB-C+SMB(i) AOBB+SMB(i) AOBB-C+SMB(i) AOBF-C+SMB(i) i=18	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	0.30		0.34		0.44		0.80		2.00		5.31	
mm-03-08-03	59.14	49,376	19.39	9,576	51.83	41,282	8.42	3,377	9.17	3,068	12.80	2,980
(20, 57)	1.58	10,396	1.64	7,075	1.50	6,349	1.38	3,830	2.53	3,420	5.73	3,153
(1220, 3, 2)	1.05	2,770	1.22	3,299	1.14	3,010	1.22	2,273	2.39	2,114	5.56	2,031
	0.72	1,366	1.14	2,196	1.22	2,202	1.20	1,311	2.36	1,247	5.66	1,220
	0.75		0.83		1.02		1.75		4.38		15.77	
mm-03-08-04	-	-	-	-	-	-	-	-	-	-	-	-
(33, 87)	92.64	150,642	110.45	193,805	64.13	71,622	17.17	31,177	36.14	63,669	22.38	13,870
(2288, 3, 2)	21.50	20,460	34.75	28,631	15.94	14,101	9.56	8,747	16.03	11,971	19.45	5,376
	10.53	9,693	10.88	9,143	10.06	8,925	**3.89**	2,928	9.08	4,855	19.52	4,266
	0.34		0.41		0.51		0.91		2.44		7.83	
mm-04-08-03	-	-	981.26	726,162	51.42	32,948	32.53	16,633	29.19	14,151	28.11	9,881
(26, 72)	15.64	68,929	6.02	26,111	8.06	34,445	5.05	17,255	6.09	15,443	10.16	10,570
(1418, 3, 2)	3.85	7,439	1.63	3,872	2.49	5,367	2.75	4,778	4.44	4,824	9.06	3,444
	0.94	1,578	0.94	1,475	1.05	1,472	1.42	1,462	2.95	1,453	8.36	1,450
	i=12		i=14		i=16		i=18		i=20		i=22	
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes
	1.36		2.08		4.86		16.53		65.19		246.45	
mm-04-08-04	-	-	-	-	-	-	-	-	-	-	-	-
(39, 103)	494.50	744,993	270.60	447,464	506.74	798,507	80.86	107,463	206.58	242,865	280.07	62,964
(2616, 3, 2)	114.02	82,070	66.84	61,328	93.50	79,555	30.80	13,924	91.08	28,648	253.25	11,650
	38.55	33,069	29.19	26,729	44.95	38,989	**20.64**	3,957	74.67	8,716	250.00	3,491
	2.34		8.52		8.31		24.94		84.52		out	
mm-03-08-05	-	-	-	-	-	-	-	-	-	-		
(41, 111)	-	-	-	-	-	-	1084.48	1,122,008	1283.04	1,185,327		
(3692, 3, 2)	-	-	-	-	-	-	117.39	55,033	282.35	86,588		
	out		out		473.07	199,725	**36.99**	8,297	131.88	21,950		
	1.64		3.09		7.55		21.08		77.81		out	
mm-10-08-03	-	-	-	-	-	-	-	-	-	-		
(51, 132)	161.35	290,594	99.09	326,662	89.06	151,128	84.16	127,130	144.03	133,112		
(2606, 3, 2)	19.86	14,518	19.47	14,739	22.34	13,557	29.80	9,388	89.75	12,362		
	4.80	3,705	8.16	4,501	11.17	3,622	24.67	3,619	81.52	3,573		

Table 4.24: CPU time and number of nodes visited for solving **Mastermind game instances**, using static mini-bucket heuristics and min-fill based pseudo trees. Time limit 1 hour. `toolbar` and `toolbar-BTD` were not able to solve any of the test instances within the time limit. The top part of the table shows the results for i-bounds between 8 and 18, while the bottom part shows i-bounds between 12 and 22.

Impact of the pseudo tree. Figure 4.20 plots the runtime distribution of `AOBB-C+SMB(`i`)` and `AOBF-C+SMB(`i`)`, over 20 independent runs, using hypergraph based pseudo trees. We observe again that, in some cases, the hypergraph trees are able to improve significantly the performance of Branch-and-Bound as well as best-first search ($e.g.$, `c880`, `s1238`).

Mastermind Game Instances

Table 4.24 shows the results for experiments with 6 Mastermind game instances of increasing difficulty, from Chapter 3 using static mini-bucket heuristics and min-fill based pseudo trees. The performance of the AND/OR algorithms with dynamic mini-buckets was quite poor in this case due to prohibitively high computational overhead at large i-bounds.

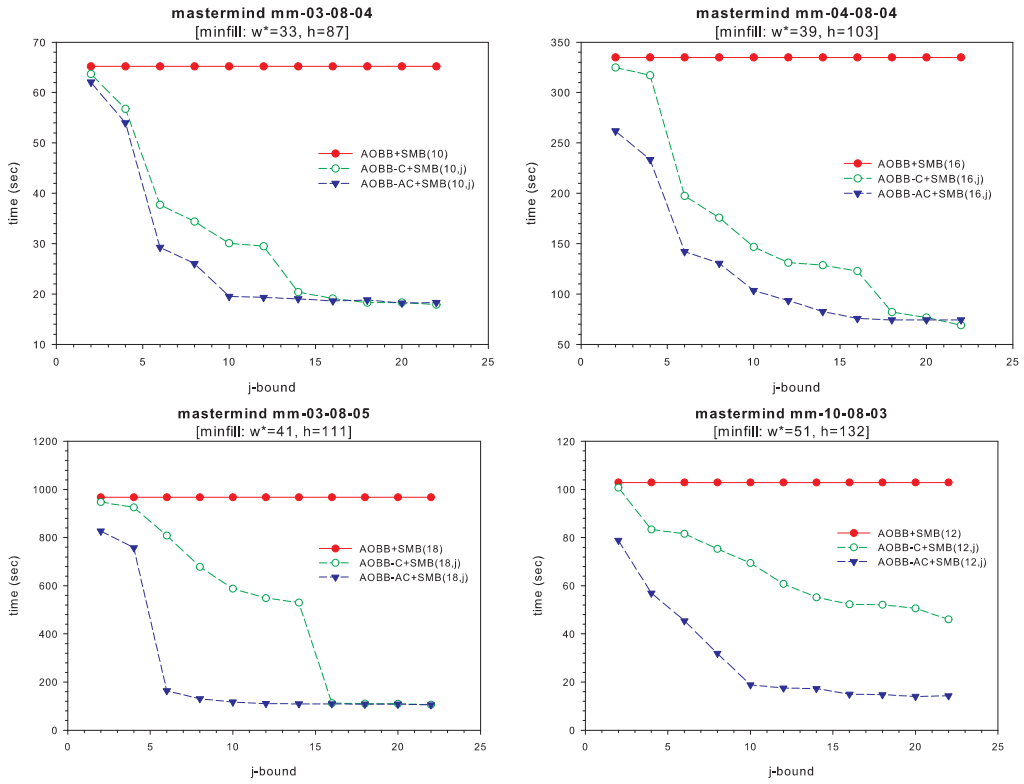


Figure 4.21: Naive versus adaptive caching schemes for AND/OR Branch-and-Bound with static mini-bucket heuristics on **Mastermind networks**. Shown is CPU time in seconds.

Tree vs. graph AOBB. We see again that using caching improves considerably the performance of AND/OR Branch-and-Bound search. On `mm-03-08-05`, for example, `AOBB-C+SMB(18)` is 9 times faster than `AOBB+SMB(18)` and explores a search space 20 times smaller. We also note that `toolbar` and `toolbar-BTD` were not able to solve any of these instances within the time limit.

AOBB vs. AOBF. When comparing best-first against depth-first AND/OR search, we see that `AOBF-C+SMB(`i`)` offers the overall best performance on this domain as well. On the `mm-03-08-05` instance, for example, `AOBF-C+SMB(18)` is about 3 times faster than `AOBB-C+SMB(18)` and about 30 times faster than `AOBB+SMB(18)`, respectively. As before, the time savings are more pronounced at relatively small i-bounds when the heuristic estimates are less accurate.

Impact of the level of caching. Figure 4.21 illustrates the CPU time, as a function of the cache bound j, obtained with the naive and adaptive caching schemes on 4 problem instances from Table 4.24. We notice again the superiority of adaptive caching at relatively small j-bounds.

Impact of the pseudo tree. Figure 4.22 plots the runtime distribution of AOBB-C+SMB(i) and AOBF-C+SMB(i), over 20 independent runs, using hypergraph pseudo trees. The hypergraph trees are sometimes able to improve slightly the performance of AND/OR Branch-and-Bound, at relatively small i-bounds ($e.g.$, mm-04-08-04). For best-first search however, the min-fill based pseudo trees offer the best performance in this case.

Memory usage of AND/OR graph search. In Figure 4.23 we emphasize again the significant memory requirements of best-first AND/OR search compared with those of the depth-first AND/OR Branch-and-Bound search with full caching. We see for example that on the mm-03-08-05 network AOBF-C+SMB(i) with relatively small i-bounds ($e.g.$, $i \in \{12, 14\}$) uses about 2 orders of magnitude more memory than AOBB-C+SMB(i).

4.6 Conclusion to Chapter 4

The chapter continues to investigate the impact of the AND/OR search spaces perspective to solving general constraint optimization problems in graphical models. In contrast to the traditional OR space, the AND/OR search space is sensitive to problem decomposition. The size of the AND/OR search tree can be bounded exponentially by the depth of its guiding pseudo tree. This implies exponential time savings for any linear space search algorithms traversing the AND/OR search tree, in particular AND/OR Branch-and-Bound search, as we showed in Chapter 3. Specifically, if the graphical model has treewidth w^*, the depth of the pseudo tree is $O(w^* \cdot logn)$. The AND/OR search tree can be extended into a graph by merging identical subtrees using graph information only. The size of the context minimal

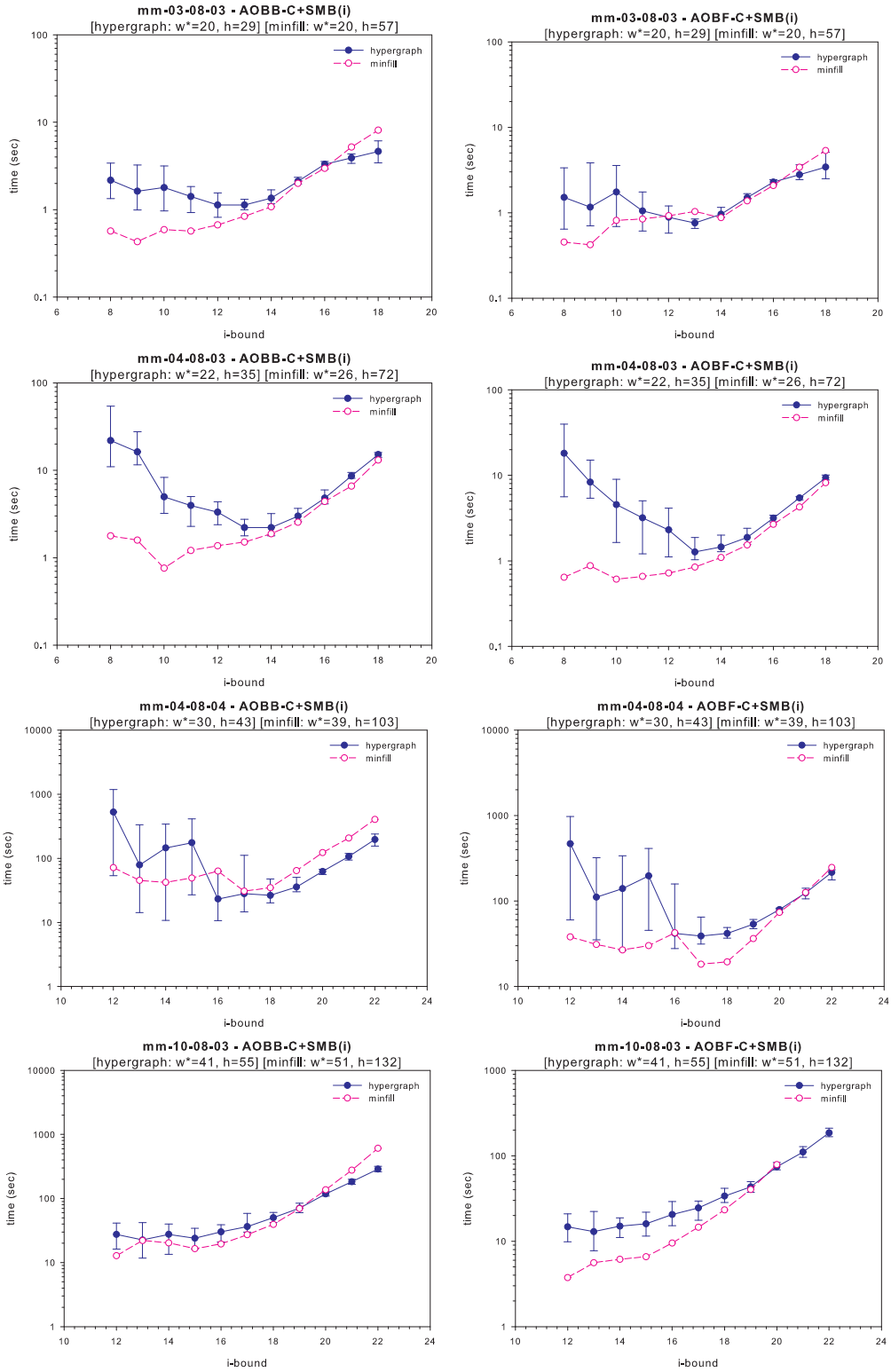


Figure 4.22: Min-fill versus hypergraph partitioning heuristics. CPU time in seconds for solving **Mastermind networks** with AOBB-C+SMB(i) (left side) and AOBF-C+SMB(i) (right side). The header of each plot records the average induced width (w^*) and pseudo tree depth (h) obtained with the hypergraph partitioning heuristic. We also show the induced width and pseudo tree depth for the min-fill heuristic.

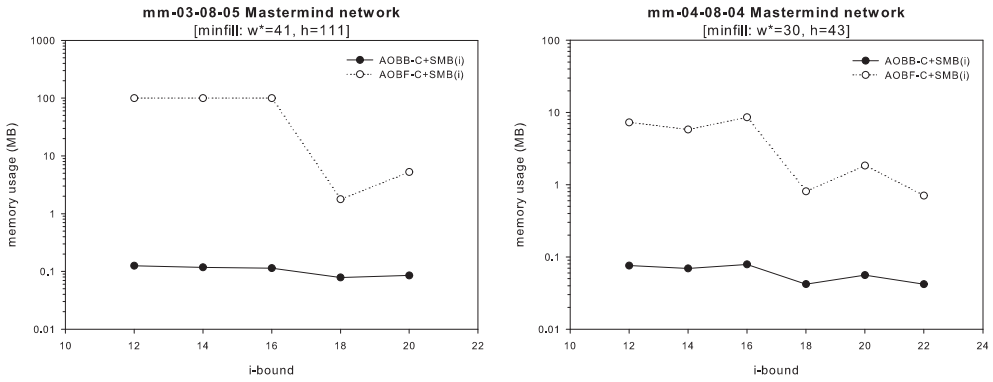


Figure 4.23: Memory usage of the AOBB-C+SMB(i) and AOBF-C+SMB(i) algorithms on the **Mastermind networks** from Table 4.24.

AND/OR search graph is exponential in the treewidth while the size of the context minimal OR search graph is exponential in the pathwidth. Since for some graphs the difference between treewidth and pathwidth is substantial ($e.g.$, balanced pseudo trees) the AND/OR representation implies substantial time and space savings for memory intensive algorithms traversing the AND/OR graph. Searching the AND/OR search graph can be implemented by goods caching during search.

We therefore extended the AND/OR Branch-and-Bound algorithm to traversing a search graph rather than a search tree by equipping it with an efficient caching mechanism. We investigated two flexible context-based caching schemes that can adapt to the current memory restrictions. Since best-first search strategies are known to be superior to depth-first ones when memory is utilized, we also introduced a best-first AND/OR search algorithm that traverses the context minimal AND/OR search graph.

All these algorithms can be guided by any heuristic function. We investigated extensively the mini-bucket heuristics introduced earlier [65] and shown to be effective in the context of OR search trees [65]. The mini-bucket heuristics can be either pre-compiled (static mini-buckets) or generated dynamically during search at each node in the search space (dynamic mini-buckets). They are parameterized by the Mini-Bucket i-bound which allows for a controllable trade-off between heuristic strength and computational overhead.

We focused our empirical evaluation on two common optimization problems in graph-

ical models: finding the MPE in Bayesian networks and solving WCSPs. Our results demonstrated conclusively that the depth-first and best-first memory intensive AND/OR search algorithms guided by mini-bucket heuristics improve dramatically over traditional memory intensive OR search as well as over AND/OR Branch-and-Bound algorithms without caching. We summarize next the most important aspects reflecting the better performance of AND/OR graph search, such as the impact of the level of cachin, the mini-bucket i-bound, constraint propagation, informed initial upper bounds and the quality of the guiding pseudo trees.

- **Impact of the level of caching.** We proposed two parameterized context-based caching schemes that can adapt to the memory limitations. The naive caching records contexts with size smaller or equal to the cache bound j. The adaptive caching saves also nodes whose context size is beyond j, based on adjusted contexts. Our results showed that for small j-bounds, adaptive caching is more powerfull than the naive scheme ($e.g.$, grid networks from Figure 4.6, genetic linkage networks from Figure 4.11, ISCAS'89 circuits from Figure 4.19). As more space becomes available and the j-bound increases, the two schemes gradually approach full caching. The savings in number of nodes due to caching are more pronounced at relatively small i-bounds of the mini-bucket heuristics. When the heuristics are strong enough to prune the search space substantially ($i.e.$, large i-bounds), the context minimal graph traversed by AND/OR Branch-and-Bound is very close to a tree and the effect of caching is diminished.

- **Impact of the mini-bucket i-bound.** Our results show conclusively that when enough memory is available the static mini-bucket heuristics with relatively large i-bounds are cost effective ($e.g.$, genetic linkage analysis networks from Tables 4.6 and 4.7, Mastermind game instances from Table 4.24). However, if the space is severly restricted, the dynamic mini-bucket heuristics appear to be the preferred choice, especially for relatively small i-bounds ($e.g.$, ISCAS'89 networks from Tables 4.22).

This is because these heuristics are far more accurate for the same i-bound than the pre-compiled version.

- **Impact of determinism.** When the graphical model contains both deterministic information (hard constraints) as well as general cost functions, we demonstrated that it is beneficial to exploit the computational power of the constraints via constraint propagation. Our experiments on selected classes of deterministic Bayesian networks showed that enforcing unit resolution over the CNF encoding of the determinism present in the network was able in some cases to render the search space almost backtrack-free ($e.g.$, ISCAS'89 networks from Table 4.19). This caused in some cases a tremendous reduction in running time for the corresponding AND/OR Branch-and-Bound algorithms ($e.g.$, see for example the `s953` network from Table 4.19).

- **Impact of good initial upper bounds.** The AND/OR Branch-and-Bound algorithm assumed a trivial initial upper bound (resp. initial lower bound for maximization tasks). We incorporated a more informed upper bound (resp. lower bound for maximization), obtained by first solving the initial problem via local search. Our results showed that in some cases it causes a tremendeous speed-up over the initial approach (see for example the grid network from Table 4.15, and the ISCAS'89 networks from Table 4.19).

- **Impact of pseudo tree quality.** The performance of the depth-first and best-first memory intensive AND/OR search algorithms is influenced significantly by the quality of the guiding pseudo tree. We investigated two heuristics for generating small induced width/depth pseudo trees. The min-fill based pseudo trees usually have smaller induced width but significantly larger depth, whereas the hypergraph partitioning heuristic produces much smaller depth trees but with larger induced widths. Our experiments demonstrated that when the induced width is small enough, which is typi-

cally the case for min-fill based pseudo trees, the strength of the mini-bucket heuristics compiled along these orderings determines the performance of the AND/OR search algorithms ($e.g.$, SPOT5 networks from Figure 4.17). However, when the graph is highly connected, the relatively large induced width causes the AND/OR algorithms to traverse a search space that is very close to a tree and, therefore, the hypergraph partitioning based pseudo trees, which have far smaller depths than the min-fill based ones, improve performance substantially ($e.g.$, genetic linkage networks from Figure 4.10 and Table 4.8). This is because for tree search the depth of the pseudo tree matters more than the induced width.

Chapter 5

AND/OR Search for 0-1 Integer Programming

5.1 Introduction

A *constraint optimization problem* is the minimization (or maximization) of an objective function subject to a set of constraints on the possible values of a set of independent decision variables. An important class of optimization problems in operations research and computer science are the 0-1 Integer Linear Programming problems (0-1 ILP) [95] where the objective is to optimize a linear function of bi-valued integer decision variables, subject to a set of linear equality or inequality constraints defined on subsets of variables. The classical approach to solving 0-1 ILPs is the *Branch-and-Bound* method [74] which traverses a search tree defined by the problem while maintaining the best solution found so far and discarding partial solutions which cannot improve on the best.

The AND/OR Branch-and-Bound (AOBB) introduced in Chapter 3 is a Branch-and-Bound algorithm that explores an AND/OR search tree in a depth-first manner for solving optimization tasks in graphical models. The AND/OR Branch-and-Bound algorithm with caching (AOBB-C) from Chapter 4 improves AOBB by allowing the algorithm to save previously computed results and retrieve them when the same subproblems are encountered again. The algorithm explores the context minimal AND/OR search graph. A *best-first* AND/OR search algorithm (AOBF-C) that traverses the AND/OR search graph was in-

troduced subsequently in Chapter 4. The algorithms were initially restricted to a static variable ordering determined by the underlying pseudo tree, but subsequent extensions to dynamic variable ordering heuristics were also introduced Chapter 3. One extension, called AND/OR Branch-and-Bound with Partial Variable Ordering (AOBB+PVO) (AOBF+PVO) was shown to have significant impact on several domains.

5.1.1 Contribution

In this chapter we extend the principles of AND/OR search and the ideas of context-based caching to solving 0-1 Integer Linear Programs. We explore both depth-first and best-first control strategies. Under conditions of admissibility and monotonicity of the guiding heuristic function, best-first search is known to expand the minimal number of nodes, at the expense of using additional memory [40]. We also extend dynamic variable ordering heuristics for AND/OR search and explore their impact on 0-1 ILPs.

We demonstrate empirically the benefit of the AND/OR search approach on several benchmarks for 0-1 ILP problems, including combinatorial auctions, random uncapacitated warehouse location problems and MAX-SAT problem instances. Our results show conclusively that the new AND/OR search approach improves dramatically over the traditional OR search on this domain, in some cases with several orders of magnitude of improved performance. We illustrate the tremendous gain obtained by exploiting problem's decomposition (using AND nodes), equivalence (by caching), branching strategy (via dynamic variable ordering heuristics) and control strategy. We also show that the AND/OR algorithms are sometimes able to outperform significantly commercial solvers like CPLEX.

The research presented in this chapter is based in part on [80, 83].

5.1.2 Chapter Outline

The chapter is organized as follows. Sections 5.2 and 5.3 provide background on 0-1 ILP and AND/OR search spaces for 0-1 ILPs. In Sections 5.4 and 5.5 we present the extensions

of the depth-first AND/OR Branch-and-Bound and the best-first AND/OR search algorithms to 0-1 ILP. Section 5.6 discusses the AND/OR search approach that incorporates dynamic variable ordering heuristics. Section 5.7 shows our empirical evaluation, while Section 5.8 provides concluding remarks.

5.2 Background

5.2.1 Integer Programming

DEFINITION **42 (linear program)** *A* linear program (LP) *consists of a set of* n *continuous variables* $\mathbf{X} = \{X_1, ..., X_n\}$ *and a set of* m *linear constraints (equalities or inequalities)* $\mathbf{F} = \{F_1, ..., F_m\}$ *defined on subsets of variables. The goal is to minimize a global linear cost function, denoted* $z(\mathbf{X})$*, subject to the constraints. One of the standard forms of a linear program is:*

$$\textit{min} \quad z(\mathbf{X}) = \sum_{i=1}^{n} c_i \cdot X_i \tag{5.1}$$

$$\textit{s.t.} \quad \sum_{i=1}^{n} a_{ij} \cdot X_i \leq b_j, \quad \forall\, 1 \leq j \leq m \tag{5.2}$$

$$X_i \geq 0, \quad \forall\, 0 \leq i \leq n \tag{5.3}$$

where (5.1) *represents the linear objective function, and* (5.2) *defines the set of linear constraints. In addition,* (5.3) *ensures that all variables are positive.*

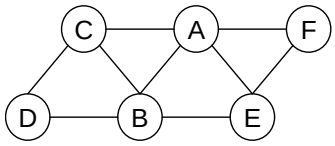
A linear program can also be expressed in a matrix notation, as follows:

$$min\{\mathbf{c}^\top \mathbf{X} \mid \mathbf{A} \cdot \mathbf{X} \leq \mathbf{b}, \mathbf{X} \geq 0\} \tag{5.4}$$

where $\mathbf{c} \in \mathbb{R}^n$*,* $\mathbf{b} \in \mathbb{R}^m$*,* $\mathbf{A} \in \mathbb{R}^{m \times n}$ *and* $\mathbf{X} \in \mathbb{R}^n_+$*. Namely,* $\mathbf{c}$ *represents the cost vector and*

$$\text{minimize}: z = 7A + 3B - 2C + 5D - 6E + 8F$$
$$\text{subject to}:$$
$$3A - 12B + C \leq 3$$
$$-2B + 5C - 3D \leq -2$$
$$2A + B - 4E \leq 2$$
$$A - 3E + F \leq 1$$
$$A, B, C, D, E, F \in \{0,1\}$$

(a) 0-1 Integer Linear Program



(b) Constraint graph

Figure 5.1: Example of a 0-1 Integer Linear Program.

$\mathbf{X}$ *is the vector of decision variables. The vector* $\mathbf{b}$ *and the matrix* $\mathbf{A}$ *define the* m *linear constraints.*

One of the most important constraint optimization problems in operations research and computer science is *integer programming*. Applications of integer programming include scheduling, routing, VLSI circuit design, combinatorial auctions, and facility location [95]. Formally:

DEFINITION **43 (0-1 integer linear programming)** *A* 0-1 integer linear programming (0-1 ILP) *problem is a linear program where all the decision variables are constrained to have integer values 0 or 1 at the optimal solution. Formally,*

$$\textit{min} \quad z(\mathbf{X}) = \sum_{i=1}^{n} c_i \cdot X_i \tag{5.5}$$

$$\textit{s.t.} \quad \sum_{i=1}^{n} a_{ij} \cdot X_i \leq b_j, \quad \forall\, 1 \leq j \leq m \tag{5.6}$$

$$X_i \in \{0, 1\} \quad \forall\, 0 \leq i \leq n \tag{5.7}$$

Example 19 *Figure 5.1(a) shows a 0-1 ILP instance with 6 binary decision variables* $(A,\ B,\ C,\ D,\ E,\ F)$ *and 4 linear constraints* $F_1(A, B, C)$, $F_2(B, C, D)$, $F_3(A, B, E)$, $F_4(A, E, F)$. *The objective function to be minimized is defined by* $z = 7A + B - 2C +$

$5D - 6E + 8F$. *Figure 5.1(b) displays the constraint graph associated with this 0-1 ILP, where nodes correspond to the variables and there is an edge between any two nodes whose corresponding variables appear in the scope of the same linear constraint.*

If some variables are constrained to be integers (not necessarily binary), then the problem is simply called *integer programming*. If not all variables are constrained to be integral (they can be real), then the problem is called *mixed integer programming* (MIP). Otherwise, the problem is called *0-1 integer programming*.

While 0-1 integer programming, and thus integer programming and MIP are all NP-hard [63], there are many sophisticated techniques that can solve very large instances in practice. We next briefly review the existing search techniques upon which we build our methods.

5.2.2 Branch-and-Bound Search for Integer Programming

In *Branch-and-Bound* search, the best solution found so far (the *incumbent*) is kept in memory. Once a node in the search tree is generated, a lower bound (also known as a heuristic evaluation function) on the solution value is computed by solving a relaxed version of the problem, while honoring the commitments made on the search path so far. The most common method for doing this is to solve the problem while relaxing only the integrality constraints of all undecided variables. The resulting *linear program* (LP) can be solved fast in practice, for example using the *simplex* algorithm [23] (and in polynomial worst-case time using integer-point methods [95]). A path terminates when the lower bound is at least the value of the incumbent, or when the subproblem is infeasible or yields an integer solution. Once all paths have terminated, the incumbent is a provably optimal solution.

There are several ways to decide which leaf node of the search tree to expand next. For example, in *depth-first* Branch-and-Bound, the most recent node is expanded next. In *best-first search* ($i.e.$, A^* search [99]), the leaf with the lowest lower bound is expanded next. A^* search is desirable because for any fixed branching variable ordering, no tree search algorithm that finds a provably optimal solution can guarantee expanding fewer nodes [40].

Therefore, of the known node-selection strategies, A^* seems to be best suited when the goal is to find a provably optimal solution. A variant of a best-first node-selection strategy, called *best-bound search*, is often used in MIP [125]. While in general A^* the children are evaluated when they are generated, in best-bound search the children are queued for expansion based on their parents' values and the LP of each child is only solved if the child comes up for expansion from the queue. Thus best-bound search needs to continue until each node on the queue has value no better than the incumbent. Best-bound search generates more nodes, but may require fewer (or more) LPs to be solved.

5.2.3 Branch-and-Cut Search for Integer Programming

A modern algorithm for solving MIPs is *Branch-and-Cut*, which first achieved success in solving large instances of the traveling salesman problem [100, 101], and is now the core of the fastest commercial general-purpose integer programming packages. It is a *Branch-and-Bound*, except that in addition, the algorithm may generate *cutting planes* [95]. They are linear constraints that, when added to the subproblem at a search node, may result in a smaller feasible space for the LP, while not cutting off the optimal integer solution, and thus a higher lower bound. The higher lower bound in turn can cause earlier termination of the search path, and thus yields smaller search trees.

5.2.4 State-of-the-art Software Packages

CPLEX[1] is a leading commercial software product for solving MIPs. It uses Branch-and-Cut, and it can be configured to support many different branching algorithms ($i.e.$, variable ordering heuristics). It also makes available low-level interfaces ($i.e.$, APIs) for controlling the search, as well as other components such as the pre-solver, the cutting plane engine and the LP solver.

[1] http://www.ilog.com/cplex/

`lp_solve`[2] is an open source linear (integer) programming solver based on the simplex and the Branch-and-Bound methods. We chose to develop our AND/OR search algorithms in the framework of `lp_solve`, because we could have access to the source code. Unlike CPLEX, `lp_solve` does not provide a cutting plane engine.

5.3 Extending AND/OR Search Spaces to 0-1 ILPs

As mentioned earlier, the common way of solving 0-1 ILPs is by search, namely to instantiate variables one at a time following a static/dynamic variable ordering. In the simplest case, this process defines an OR search tree, whose nodes represent states in the space of partial assignments. However, this search space does not capture independencies that appear in the structure of the problem. The AND/OR search space for graphical models presented in Chapters 3 and 4 remedies this problem and it can be extended to 0-1 ILPs in a straightforward manner. For completeness sake, we describe it next briefly.

5.3.1 AND/OR Search Trees for 0-1 ILPs

Given a 0-1 ILP instance, its constraint graph G and a pseudo tree $\mathcal{T}$ of G, the associated AND/OR search tree $S_{\mathcal{T}}$ has alternating levels of OR nodes and AND nodes. The OR nodes are labeled by X_i and correspond to the variables. The AND nodes are labeled by $\langle X_i, x_i \rangle$ (or simply x_i) and correspond to value assignments in the domains of the variables that are consistent relative to the constraints. The structure of the AND/OR tree is based on the underlying pseudo tree $\mathcal{T}$ of G. The root of the AND/OR search tree is an OR node, labeled with the root of $\mathcal{T}$. The children of an OR node X_i are AND nodes labeled with assignments $\langle X_i, x_i \rangle$, consistent along the path from the root. The children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of variable X_i in $\mathcal{T}$.

Example 20 *Consider the 0-1 ILP from Figure 5.2(a). A pseudo tree of the constraint*

[2] http://lpsolve.sourceforge.net/5.5/

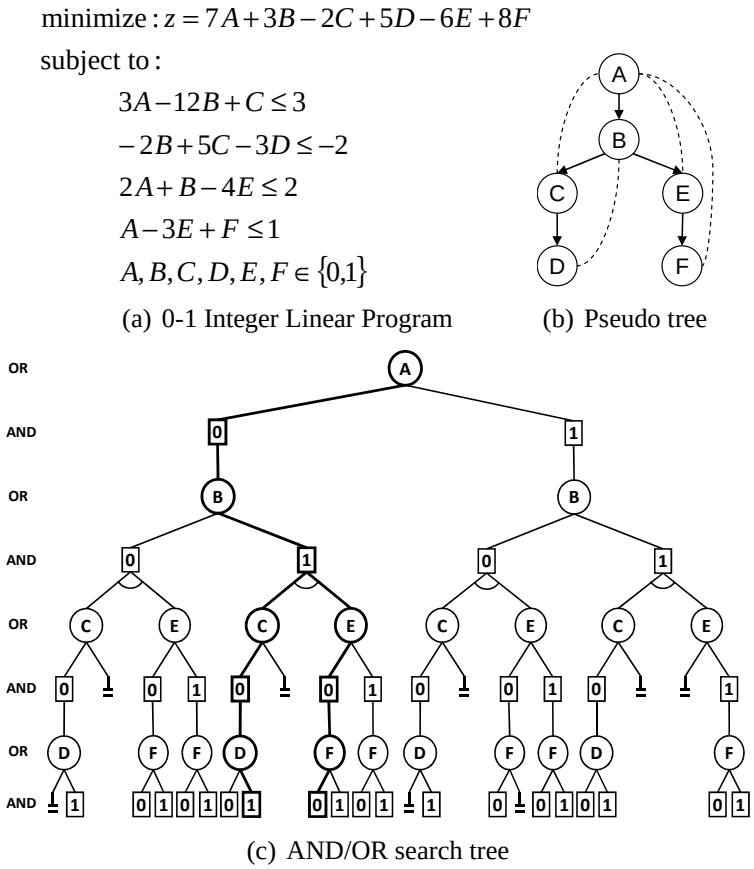
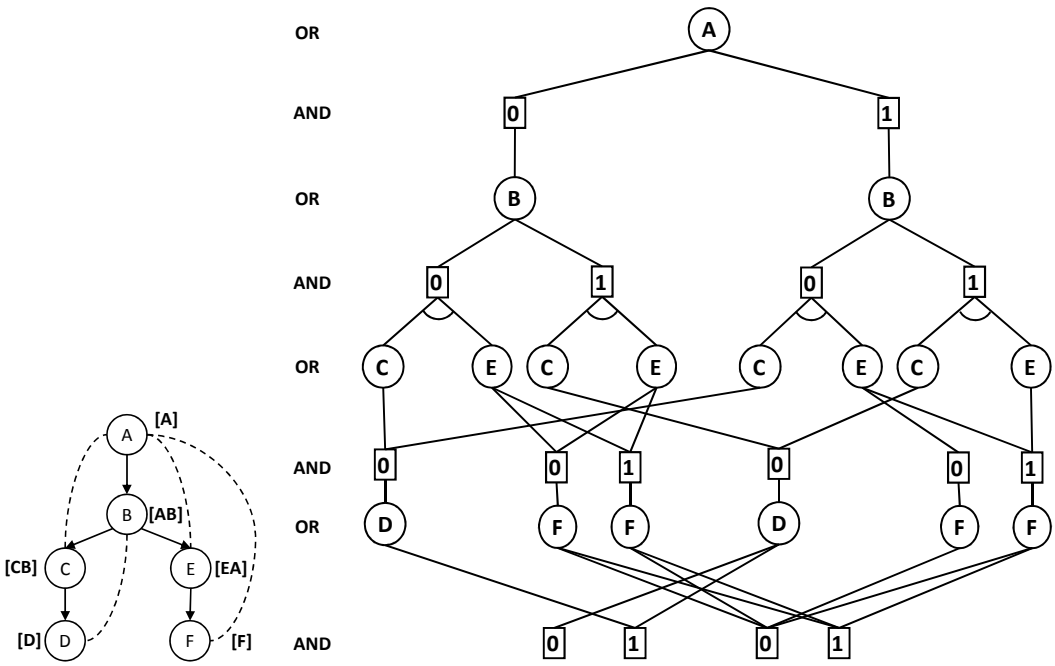


Figure 5.2: AND/OR search tree for a 0-1 Integer Linear Program instance.

graph, together with the back-arcs (dotted lines) are given in Figure 5.2(b). Figure 5.2(c) shows the corresponding AND/OR search tree. Notice that the partial assignment $(A = 0, B = 0, C = 0, D = 0)$ *which is represented by the path* $\{A, \langle A, 0\rangle, B, \langle B, 0\rangle, C, \langle C, 0\rangle, D, \langle D, 0\rangle\}$ *in the AND/OR search tree, is inconsistent because the constraint* $-2B + 5C - 3D \leq -2$ *is violated. Similarly, the partial assignment* $(A = 0, B = 0, C = 1)$ *is also inconsistent due to the violation of the same constraint for any value assignment to* D*.*

The arcs in the AND/OR search tree of a 0-1 ILP are associated with *weights* that are derived from the objective function $\sum_{i=1}^{n} c_i \cdot X_i$. The *weight* $w(n, m)$ of the arc from the OR node n, labeled X_i to the AND node m, labeled $\langle X_i, x_i\rangle$, is $w(n, m) = c_i \cdot x_i$.

Given a weighted AND/OR search tree of a 0-1 ILP, each of its nodes can be associated with a *value*. The value $v(n)$ of a node n is the minimal cost solution to the subproblem



(a) Pseudo tree (b) Context minimal AND/OR search graph

Figure 5.3: Context minimal AND/OR search graph for the 0-1 ILP from Figure 5.2.

rooted at n, subject to the current variable instantiation along the path from the root to n. It can be computed recursively using the values of n's successors, as shown in Chapter 3.

5.3.2 AND/OR Search Graphs for 0-1 ILPs

Often different nodes in the search tree root identical subtrees, and correspond to identical subproblems. Any two such nodes can be *merged*, reducing the size of the search space and converting it into a graph. Some of these mergeable nodes can be identified based on *contexts*, as described in Chapter 4.

Example 21 *Figure 5.3(b) shows the context minimal AND/OR search graph, relative to the pseudo tree from Figure 5.3(a), corresponding to the 0-1 ILP from Figure 5.2. The square brackets indicate the AND contexts of the variables.*

5.4 Depth-First AND/OR Branch-and-Bound Search

In Chapter 4 we introduced a new generation of depth-first Branch-and-Bound and best-first AND/OR search algorithms for solving constraint optimization tasks in graphical models. Our extensive empirical evaluations on a variety of probabilistic and deterministic graphical models demonstrated the power of these new algorithms over competitive approaches exploring traditional OR search spaces. We next revisit the depth-first Branch-and-Bound algorithm for searching AND/OR graphs, focusing on the specific properties for 0-1 ILPs.

The **Depth-First AND/OR Branch-and-Bound Search** algorithm, `AOBB-C-ILP`, that traverses the context minimal AND/OR graph via full caching is described by Algorithm 10 and shown here for completeness. It specializes the Branch-and-Bound algorithm introduced in Chapter 4 to 0-1 ILPs. If the caching mechanism is disabled then the algorithm uses linear space only and traverses an AND/OR search tree (see also Chapter 3 for more details).

As we showed in Chapter 4, the context based caching is done using tables. For each variable X_i, a table is reserved in memory for each possible assignment to its context. Initially, each entry has a predefined value, in our case `NULL`. The fringe of the search is maintained on a stack called `OPEN`. The current node is denoted by n, its parent by p, and the current path by π_n. The children of the current node are denoted by $succ(n)$. The flag `caching` is used to enable the caching mechanism.

Each node n in the search graph maintains its current value $v(n)$, which is updated based on the values of its children. For OR nodes, the current $v(n)$ is an upper bound on the optimal solution cost below n. Initially, $v(n)$ is set to ∞ if n is OR, and 0 if n is AND, respectively. The heuristic function $h(n)$ of $v(n)$ associated with each node n in the search graph is computed by solving the LP relaxation of the subproblem rooted at n, P_n, conditioned on the current partial assignment along π_n ($i.e.$, $asgn(\pi_n)$) (lines 11 and 28, respectively). Notice that if the LP relaxation of P_n is infeasible, then we assign $h(n) = \infty$ and $v(n) = \infty$. Similarly, if P_n has an integer solution, then $h(n)$ equals $v(n)$. In both

Algorithm 10: AOBB-C-ILP: AND/OR Branch-and-Bound Search for 0-1 ILP

Input: A 0-1 ILP instance with objective function $\sum_{i=1}^{n} c_i X_i$, pseudo tree $\mathcal{T}$ rooted at X_1, AND contexts pas_i for every variable X_i, caching set to $true$ or $false$.
Output: Minimal cost solution.

```
1  v(s) ← ∞; OPEN ← {s}                                              // Initialize search stack
2  if caching == true then
3      Initialize cache tables with entries "NULL"                   // Initialize cache tables
4  while OPEN ≠ ∅ do
5      n ← top(OPEN); remove n from OPEN; succ(n) ← ∅                // EXPAND
6      if n is marked INFEASIBLE or INTEGER then
7          v(n) ← ∞ (if INFEASIBLE) or v(n) ← h(n) (if INTEGER)
8      else if n is an OR node, labeled X_i then
9          foreach x_i ∈ D_i do
10             create an AND node n', labeled ⟨X_i, x_i⟩
11             v(n') ← 0; h(n') ← LP(P_n')                           // Solve the LP relaxation
12             w(n, n') ← c_i · x_i                                  // Compute the arc weight
13             mark n' as INFEASIBLE or INTEGER if the LP relaxation is infeasible or has an integer solution
14             succ(n) ← succ(n) ∪ {n'}
15     else if n is an AND node, labeled ⟨X_i, x_i⟩ then
16         cached ← false; deadend ← false
17         if caching == true and Cache(asgn(π_n)[pas_i]) ≠ NULL then
18             v(n) ← Cache(asgn(π_n)[pas_i])                        // Retrieve value
19             cached ← true                                         // No need to expand below
20         foreach OR ancestor m of n do
21             lb ← evalPartialSolutionTree(T'_m)
22             if lb ≥ v(m) then
23                 deadend ← true                                    // Pruning
24                 break
25         if deadend == false and cached == false then
26             foreach X_j ∈ children_T(X_i) do
27                 create an OR node n' labeled X_j
28                 v(n') ← ∞; h(n') ← LP(P_n')                       // Solve the LP relaxation
29                 mark n' as INFEASIBLE or INTEGER if the LP relaxation is infeasible or has an integer solution
30                 succ(n) ← succ(n) ∪ {n'}
31         else if deadend == true then
32             succ(p) ← succ(p) − {n}
33     Add succ(n) on top of OPEN                                    // PROPAGATE
34     while succ(n) == ∅ do
35         if n is an OR node, labeled X_i then
36             if X_i == X_1 then
37                 return v(n)                                       // Search is complete
38             v(p) ← v(p) + v(n)                                    // Update AND node value (summation)
39         else if n is an AND node, labeled ⟨X_i, x_i⟩ then
40             if caching == true and v(n) ≠ ∞ then
41                 Cache(asgn(π_n)[pas_i]) ← v(n)                    // Save AND node value in cache
42             if v(p) > (w(p, n) + v(n)) then
43                 v(p) ← w(p, n) + v(n)                             // Update OR node value (minimization)
44         remove n from succ(p)
45         n ← p
```

Algorithm 11: Recursive computation of the heuristic evaluation function.

function: `evalPartialSolutionTree(`T'_n`)`
Input: Partial solution subtree T'_n rooted at node n.
Output: Heuristic evaluation function $f(T'_n)$.

```
1  if succ(n) == ∅ then
2      return h(n)
3  else
4      if n is an AND node then
5          let m_1, ..., m_k be the OR children of n in T'_n
6          return Σ_{i=1}^{k} evalPartialSolutionTree(T'_{m_i})
7      else if n is an OR node then
8          let m be the AND child of n in T'_n
9          return w(n, m) + evalPartialSolutionTree(T'_m)
```

cases, $succ(n)$ is set to the empty set, thus avoiding n's expansion (lines 6–7).

Before expanding the current AND node n, its cache table is checked (line 18). If the same context was encountered before, it is retrieved from the cache, and $succ(n)$ is set to the empty set, which will trigger the PROPAGATE step. Otherwise, the node is expanded in the usual way, depending on whether it is an AND or OR node (lines 8–32). The algorithm also computes the heuristic evaluation function for every partial solution subtree rooted at the OR ancestors of n along the path from the root (lines 20–24). The search below n is terminated if, for some OR ancestor m, $f(T'_m) \geq v(m)$, where $v(m)$ is the current upper bound on the optimal cost below m. The recursive computation of $f(T'_m)$ based on Definition 30 in Chapter 3 is described in Algorithm 11.

The node values are updated by the PROPAGATE step (lines 34–45). It is triggered when a node has an empty set of descendants (note that as each successor is evaluated, it is removed from the set of successors in line 44). This means that all its children have been evaluated, and their final values are already determined. If the current node is the root, then the search terminates with its value (line 37). If n is an OR node, then its parent p is an AND node, and p updates its current value $v(p)$ by summation with the value of n (line 38). An AND node n propagates its value to its parent p in a similar way, by minimization (lines 42–43). Finally, the current node n is set to its parent p (line 45), because n was completely evaluated. Search continues either with a *propagation* step (if conditions are met) or with an *expansion* step. We give next an example of the pruning mechanism used

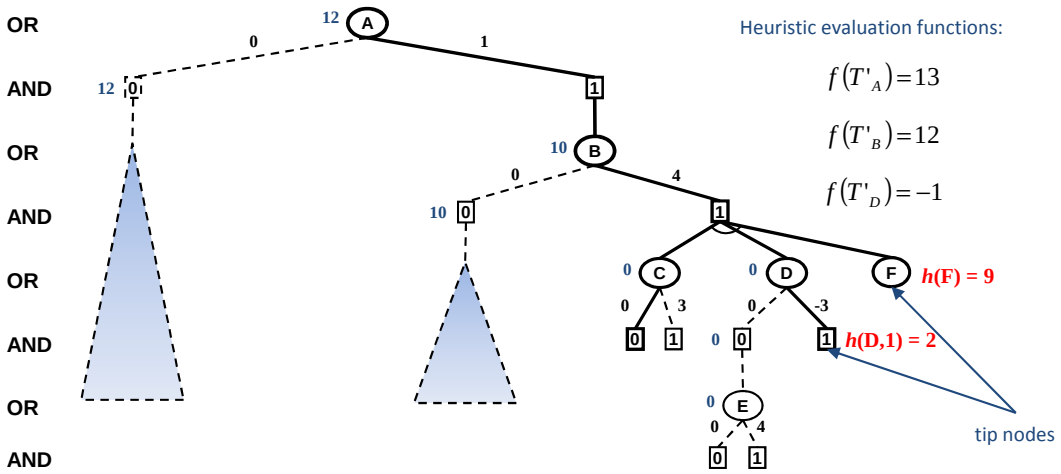


Figure 5.4: Illustration of the pruning mechanism.

by AOBB-C-ILP.

Example 22 *Consider the partially explored weighted AND/OR search tree in Figure 5.4. The current partial solution tree T' is highlighted. It contains the following nodes: A, $\langle A,1\rangle$, B, $\langle B,1\rangle$, C, $\langle C,0\rangle$, D, $\langle D,1\rangle$ and F. The nodes labeled by $\langle D,1\rangle$ and by F are non-terminal tip nodes and their corresponding heuristic estimates are $h(\langle D,1\rangle)=2$ and $h(F)=9$, respectively. The subtrees rooted at the AND nodes labeled $\langle A,0\rangle$, $\langle B,0\rangle$ and $\langle D,0\rangle$ are fully evaluated, and therefore the current upper bounds of the OR nodes labeled A, B and D, along the active path, are $ub(A)=12$, $ub(B)=10$ and $ub(D)=0$, respectively. Moreover, the heuristic evaluation functions of the partial solution subtrees rooted at the OR nodes along the current path can be computed recursively based on Definition 30 in Chapter 3, namely $f(T'_A)=13$, $f(T'_B)=12$ and $f(T'_D)=-1$, respectively. Notice that while we could prune below $\langle D,1\rangle$ because $f(T'_A)>ub(A)$, we could discover this pruning earlier by looking at node B only, because $f(T'_B)>ub(B)$. Therefore, the partial solution tree T'_A need not be consulted in this case.*

AOBB-C-ILP is restricted to a static variable ordering determined by the pseudo tree and explores the context minimal AND/OR search graph via *full caching*. However, if the memory requirements are prohibitive, rather than using full caching, AOBB-C-ILP can

be modified to use a memory bounded caching scheme that saves only those nodes whose context size can fit in the available memory, as described in Chapter 4.

5.5 Best-First AND/OR Search

We now direct our attention to a *best-first* rather than depth-first control strategy for traversing the context minimal AND/OR graph and present a best-first AND/OR search algorithm for 0-1 ILP. The algorithm uses similar amounts of memory as the depth-first AND/OR Branch-and-Bound with full caching. It was described in Chapter 4 and evaluated for general constraint optimization problems. By specializing it to 0-1 ILP using the LP relaxation for the heuristic function h, we get `AOBF-C-ILP`. For completeness sake, we describe the algorithm again including minor modifications for the 0-1 ILP case.

The algorithm, denoted by `AOBF-C-ILP` (Algorithm 12), specializes Nilsson's `AO`* algorithm [97] to AND/OR search spaces for 0-1 ILPs. It interleaves forward expansion of the best partial solution tree (`EXPAND`) with a cost revision step (`REVISE`) that updates node values, as detailed in [97]. The explicated AND/OR search graph is maintained by a data structure called $\mathcal{G}'_{\mathcal{T}}$, the current node is n, s is the root of the search graph and the current best partial solution subtree is denoted by T'. The children of a node n are denoted by $succ(n)$.

First, a top-down, graph-growing operation finds the best partial solution tree by tracing down through the marked arcs of the explicit AND/OR search graph $\mathcal{G}'_{\mathcal{T}}$ (lines 4–9). These previously computed marks indicate the current best partial solution tree from each node in $\mathcal{G}'_{\mathcal{T}}$. Before the algorithm terminates, the best partial solution tree, T', does not yet have all of its leaf nodes terminal. One of its non-terminal leaf nodes n is then expanded by generating its successors, depending on whether it is an OR or an AND node. Notice that when expanding an OR node, the algorithm does not generate AND children that are already present in the explicit search graph $\mathcal{G}'_{\mathcal{T}}$ (lines 13–15). All these identical AND

Algorithm 12: AOBF-C-ILP: Best-First AND/OR Search for 0-1 ILP

Input: A 0-1 ILP instance with objective function $\sum_{i=1}^{n} c_i X_i$, pseudo tree $\mathcal{T}$ rooted at X_1, AND contexts pas_i for every variable X_i

Output: Minimal cost solution.

$v(s) \leftarrow h(s)$; $\mathcal{G}'_{\mathcal{T}} \leftarrow \{s\}$ // Initialize
while *s is not labeled SOLVED* **do**
  $S \leftarrow \{s\}$; $T' \leftarrow \emptyset$; // Create the marked partial solution tree
  while $S \neq \emptyset$ **do**
    $n \leftarrow$ top(S); remove n from S
    $T' \leftarrow T' \cup \{n\}$
    let L be the set of marked successors of n
    if $L \neq \emptyset$ **then**
      add L on top of S
  let n be any nonterminal tip node of the marked T' (rooted at s) // **EXPAND**
  if *n is an OR node, labeled X_i* **then**
    foreach $x_i \in D_i$ **do**
      let n' be the AND node in $\mathcal{G}'_{\mathcal{T}}$ having context equal to pas_i
      if $n' == NULL$ **then**
        create an AND node n' labeled $\langle X_i, x_i \rangle$
        $h(n') \leftarrow LP(P_{n'})$; $v(n') \leftarrow h(n')$ // Solve the LP relaxation
        $w(n, n') \leftarrow c_i \cdot x_i$ // Compute the arc weight
        label n' as INFEASIBLE or INTEGER if the LP relaxation is infeasible or has an integer solution
        if *n' is INTEGER or TERMINAL* **then**
          label n' as SOLVED
        else if *n' is INFEASIBLE* **then**
          $v(n') \leftarrow \infty$
      $succ(n) \leftarrow succ(n) \cup \{n'\}$
  else if *n is an AND node, labeled $\langle X_i, x_i \rangle$* **then**
    foreach $X_j \in children_{\mathcal{T}}(X_i)$ **do**
      create an OR node n' labeled X_j
      $h(n') \leftarrow LP(P_{n'})$; $v(n') \leftarrow h(n')$ // Solve the LP relaxation
      label n' as INFEASIBLE or INTEGER if the LP relaxation is infeasible or has an integer solution
      if *n' is INTEGER* **then**
        mark n' as SOLVED
      else if *n' is INFEASIBLE* **then**
        $v(n') \leftarrow \infty$
      $succ(n) \leftarrow succ(n) \cup \{n'\}$
  $\mathcal{G}'_{\mathcal{T}} \leftarrow \mathcal{G}'_{\mathcal{T}} \cup succ(n)$
  $S \leftarrow \{n\}$ // **REVISE**
  while $S \neq \emptyset$ **do**
    let m be a node in S such that m has no descendants in $\mathcal{G}'_{\mathcal{T}}$ still in S; remove m from S
    if *m is an AND node, labeled $\langle X_i, x_i \rangle$* **then**
      $v(m) \leftarrow \sum_{m' \in succ(m)} v(m')$
      mark all arcs to the successors
      label m as SOLVED if all its children are labeled SOLVED
    else if *m is an OR node, labeled X_i* **then**
      $v(m) = min_{m' \in succ(m)}(w(m, m') + v(m'))$
      mark the arc through which this minimum is achieved
      label m as SOLVED if the marked successor is labeled SOLVED
    if *m changes its value **or** m is labeled SOLVED* **then**
      add to S all those parents of m such that m is one of their successors through a marked arc.
return $v(s)$ // Search terminates

nodes in $\mathcal{G}'_{\mathcal{T}}$ are easily recognized based on their contexts. Upon node's n expansion, a heuristic underestimate $h(n')$ of $v(n')$ is assigned to each of n's successors $n' \in succ(n)$ (lines 12–25). Again, $h(n')$ is obtained by solving the LP relaxation of the subproblem rooted at n', conditioned on the current partial assignment of the path to the root. As before, `AOBF-C-ILP` avoids expanding those nodes for which the corresponding LP relaxation is infeasible or yields an integer solution (lines 18–22 and 28–32).

The second operation in `AOBF-C-ILP` is a bottom-up, cost revision, arc marking, SOLVE-labeling procedure (lines 26–40). Starting with the node just expanded n, the procedure revises its value $v(n)$, using the newly computed values of its successors, and marks the outgoing arcs on the estimated best path to terminal nodes. This revised value is then propagated upwards in the graph. The revised value $v(n)$ is an updated lower bound estimate of the cost of an optimal solution to the subproblem rooted at n. During the bottom-up step, `AOBF-C-ILP` labels an AND node as `SOLVED` if all of its OR child nodes are solved, and labels an OR node as `SOLVED` if its marked AND child is also solved. The algorithm terminates with the optimal solution when the root node s is labeled `SOLVED`. We next summarize the complexity of both depth-first and best-first AND/OR graph search:

THEOREM **12 (complexity)** *The depth-first and best-first AND/OR graph search algorithms guided by a pseudo tree $\mathcal{T}$ are sound and complete for solving 0-1 ILPs. Their time and space complexity is $O(n \cdot 2^{w^*})$, where w^* is the induced width of the pseudo tree.*

Proof. Immediate from Theorem 9, which bounds the size of the context minimal AND/OR search graph. □

5.6 Dynamic Variable Orderings

The depth-first and best-first AND/OR search algorithms presented in the previous sections assumed a static variable ordering determined by the underlying pseudo tree of the con-

straint graph. However, the mechanism of identifying unifiable AND nodes based solely on their contexts is hard to extend when variables are instantiated in a different order than that dictated by the pseudo tree. In this section we discuss a strategy that allows dynamic variable orderings in depth-first and best-first AND/OR search, when both algorithms traverse an AND/OR search tree. The approach called *Partial Variable Ordering (PVO)*, which combines the static AND/OR decomposition principle with a dynamic variable ordering heuristic, was described and tested also for general constraint optimization over graphical models in Chapter 3. For completeness sake, we review it briefly next.

Variable Orderings for Integer Programming. At every node in the search tree, the search algorithm has to decide what variable to instantiate next. One common method in operations research is to select next the *most fractional variable*, $i.e.$, the variable whose LP value is furthest from being integral [125]. Finding a candidate variable under this rule is fast and the method yields small search trees on many problem instances.

A more sophisticated approach, which is better suited for certain hard problems is *strong branching* [21]. This method performs a one-step lookahead for each variable that is non-integral in the LP at the node. The one-step lookahead computation solves the LP relaxations for each of the children of the candidate variable, and a score is computed based on the LP values of the children. The next variable to instantiate is selected as the one with the highest score among the candidates.

Partial Variable Ordering (PVO). *AND/OR Branch-and-Bound with Partial Variable Ordering* (resp. *Best-First AND/OR Search with Partial Variable Ordering*), denoted by `AOBB+PVO-ILP` (resp. `AOBF+PVO-ILP`), uses the static graph-based decomposition given by a pseudo tree with a dynamic semantic ordering heuristic applied over chain portions of the pseudo tree. For simplicity and without loss of generality we consider the *most fractional variable* as our semantic variable ordering heuristic. Clearly, it can be replaced

Problem classes	Tree search AOBB-ILP AOBF-ILP	Tree search AOBB+PVO-ILP AOBF+PVO-ILP	Graph search AOBB-C-ILP AOBF-C-ILP	ILP solvers BB (lp_solve) CPLEX 11.0
Combinatorial Auctions	√	√	√	√
Warehouse Location Problems	√	√	√	√
MAX-SAT Instances	√	√	√	√

Table 5.1: Detailed outline of the experimental evaluation for 0-1 ILP.

by any other heuristic.

Consider the pseudo tree from Figure 5.2(b) inducing the following variable groups (or chains): $\{A, B\}$, $\{C, D\}$ and $\{E, F\}$, respectively. This implies that variables $\{A, B\}$ should be considered before $\{C, D\}$ and $\{E, F\}$. The variables in each group can be dynamically ordered based on a second, independent heuristic.

`AOBB+PVO-ILP` (resp. `AOBF+PVO-ILP`) can be derived from Algorithm 10 (resp. Algorithm 12) with some simple modifications. The algorithm traverses an AND/OR search tree in a depth-first (resp. best-first) manner, guided by a pre-computed pseudo tree $\mathcal{T}$. When the current AND node n, labeled $\langle X_i, x_i \rangle$, is expanded in the forward step, the algorithm generates its OR successor m, labeled X_j, based on the semantic ordering heuristic. Specifically, m corresponds to the most fractional variable in the current pseudo tree chain. If there are no uninstantiated variables left in the current chain, namely variable X_i was instantiated last, then the OR successors of n are labeled by the most fractional variables from the variable groups rooted by X_i in $\mathcal{T}$.

5.7 Experimental Results

We evaluated the performance of the depth-first and best-first AND/OR search algorithms on 0-1 ILP problem classes such as combinatorial auction, uncapacitated warehouse location problems and MAX-SAT problem instances. We implemented our algorithms in C++ and carried out all experiments on a 2.4GHz Pentium IV with 2GB of RAM, running Windows XP.

Algorithms. The detailed outline of the experimental evaluation is given in Table 5.1. We evaluated the following 6 classes of AND/OR search algorithms:

1 Depth-first and best-first search algorithms using a static variable ordering and exploring the AND/OR tree, denoted by `AOBB-ILP` and `AOBF-ILP`, respectively.

2 Depth-first and best-first search algorithms using dynamic partial variable orderings and exploring the AND/OR tree, denoted by `AOBB+PVO-ILP` and `AOBF+PVO-ILP`, respectively.

3 Depth-first and best-first search algorithms with caching that explore the context minimal AND/OR graph and use static variable orderings, denoted by `AOBB-C-ILP` and `AOBF-C-ILP`, respectively.

All of these AND/OR algorithms use a *simplex* implementation based on the open-source `lp_solve 5.5` library to compute the guiding LP relaxation. For this reason, we compare them against the OR Branch-and-Bound algorithm available from the `lp_solve` library, denoted by `BB`. The pseudo tree used by the AND/OR algorithms was constructed using the hypergraph partitioning heuristic described in Chapter 3. `BB`, `AOBB+PVO-ILP` and `AOBF+PVO-ILP` used a dynamic variable ordering heuristic based on *reduced costs* ($i.e.$, dual values) [95]. Specifically, the next fractional variable to instantiate has the smallest reduced cost in the solution of the LP relaxation. Ties are broken lexicographically.

We note however that the `AOBB-ILP` and `AOBB-C-ILP` algorithms support a restricted form of dynamic variable and value ordering. Namely, there is a dynamic internal ordering of the successors of the node just expanded, before placing them onto the search stack. Specifically, in line 33 of Algorithm 10, if the current node n is AND, then the independent subproblems rooted by its OR children can be solved in decreasing order of their corresponding heuristic estimates (variable ordering). Alternatively, if n is OR, then

its AND children corresponding to domain values can also be sorted in decreasing order of their heuristic estimates (value ordering).

For reference, we also ran the ILOG CPLEX version 11.0 solver (with default settings), which uses a best-first control strategy, dynamic variable ordering heuristic based on strong branching, as well as cutting planes to tighten the LP relaxation. It explores however an OR search tree.

In the MAX-SAT domain we ran, in addition, three specialized solvers:

1 `MaxSolver` [126], a DPLL-based algorithm that uses a 0-1 non-linear integer formulation of the MAX-SAT problem,

2 `toolbar` [26], a classic OR Branch-and-Bound algorithm that solves MAX-SAT as a Weighted CSP problem [9], and

3 `PBS` [2], a DPLL-based solver capable of propagating and learning pseudo-boolean constraints as well as clauses.

`MaxSolver` and `toolbar` were shown to perform very well on random MAX-SAT instances with high graph connectivity [26], whereas `PBS` exhibits better performance on relatively sparse MAX-SAT instances [126]. These algorithms explore an OR search space.

Throughout our empirical evaluation we will address the following questions that govern the performance of the proposed algorithms:

1 The impact of AND/OR versus OR search.

2 The impact of best-first versus depth-first AND/OR search.

3 The impact of caching.

4 The impact of dynamic variable orderings.

Measures of Performance. We report CPU time (in seconds) and number of nodes visited

(which is equivalent to the number of times *simplex* was called to solve the LP relaxation of the current subproblem). We also specify the number of variables (n), the number of constraints (c), the depth of the pseudo trees (h) and the induced width of the graphs (w^*) obtained for each problem instance. The best performance points are highlighted. In each table, "-" denotes that the respective algorithm exceeded the time limit. Similarly, "out" stands for exceeding the 2GB memory limit.

5.7.1 Combinatorial Auctions

In **combinatorial auctions** (CA), an auctioneer has a set of goods, $M = \{1, 2, ..., m\}$ to sell and the buyers submit a set of bids, $\mathcal{B} = \{B_1, B_2, ..., B_n\}$. A bid is a tuple $B_j = (S_j, p_j)$, where $S_j \subseteq M$ is a set of goods and $p_j \geq 0$ is a price. The winner determination problem is to label the bids as winning or losing so as to maximize the sum of the accepted bid prices under the constraint that each good is allocated to at most one bid. The problem can be formulated as a 0-1 ILP, as follows:

$$max \sum_{j=1}^{n} p_j x_j \tag{5.8}$$

$$\begin{aligned} \text{s.t.} \quad & \textstyle\sum_{j|i \in S_j} x_j \leq 1 \quad i \in \{1..m\} \\ & x_j \in \{0, 1\} \qquad j \in \{1..n\} \end{aligned}$$

Combinatorial auctions can also be formulated as binary Weighted CSPs [9], as described in [34]. Therefore, in addition to the 0-1 ILP solvers, we also ran `toolbar` which is a specialized OR Branch-and-Bound algorithm that maintains a level of local consistency called *existential directional arc-consistency* [25].

regions-upv and arbitrary-upv Combinatorial Auctions

Figures 5.5 and 5.6 display the results for experiments with combinatorial auctions drawn from the *regions-upv* (Figure 5.5) and *arbitrary-upv* (Figure 5.6) distributions of CATS 2.0

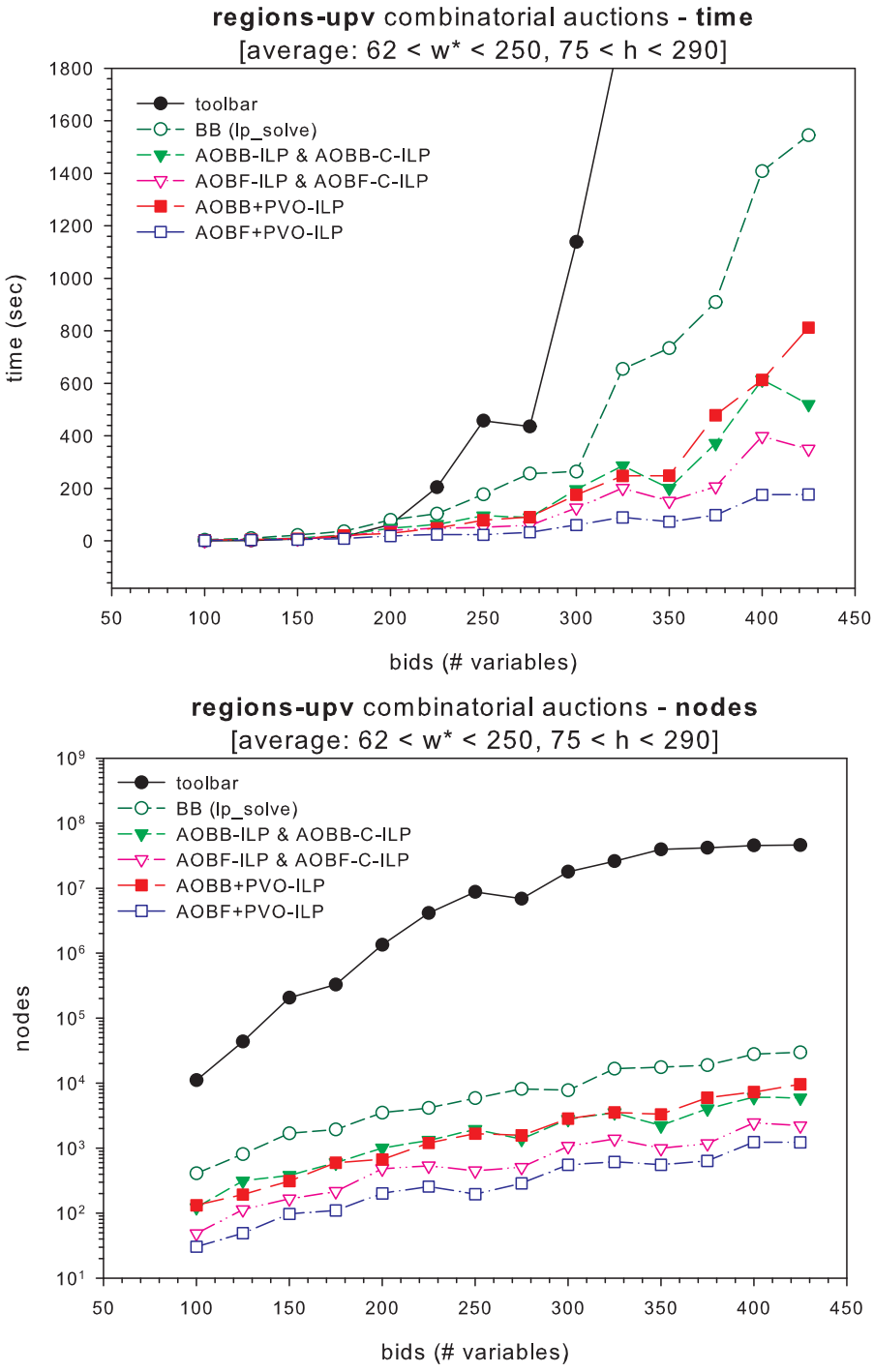


Figure 5.5: Comparing depth-first and best-first AND/OR search algorithms with static and dynamic variable orderings. CPU time in seconds (top) and number of nodes visited (bottom) for solving combinatorial auctions from the ***regions-upv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours.

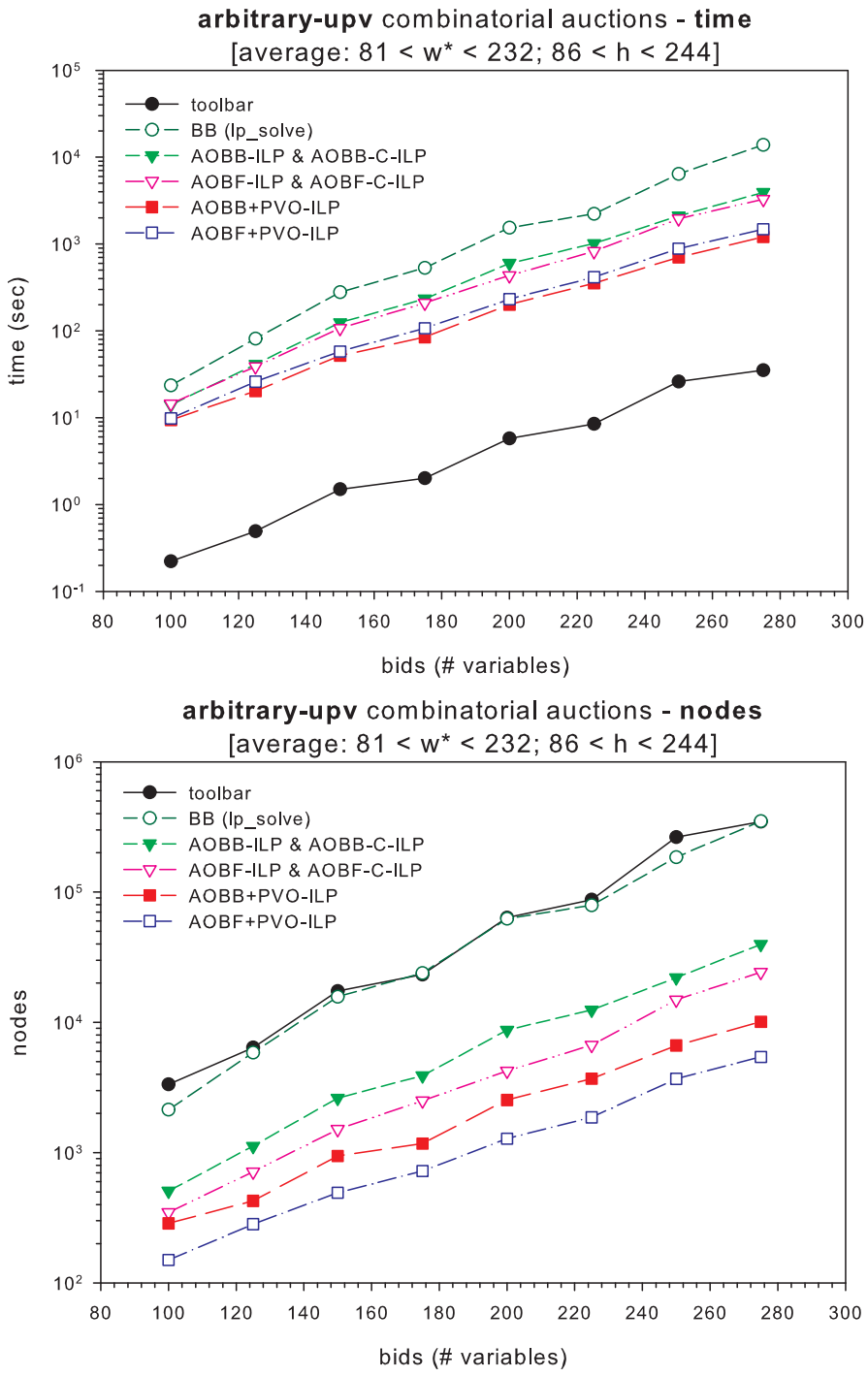


Figure 5.6: Comparing depth-first and best-first AND/OR search algorithms with static and dynamic variable orderings. CPU time in seconds (top) and number of nodes visited (bottom) for solving combinatorial auctions from the ***arbitrary-upv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours.

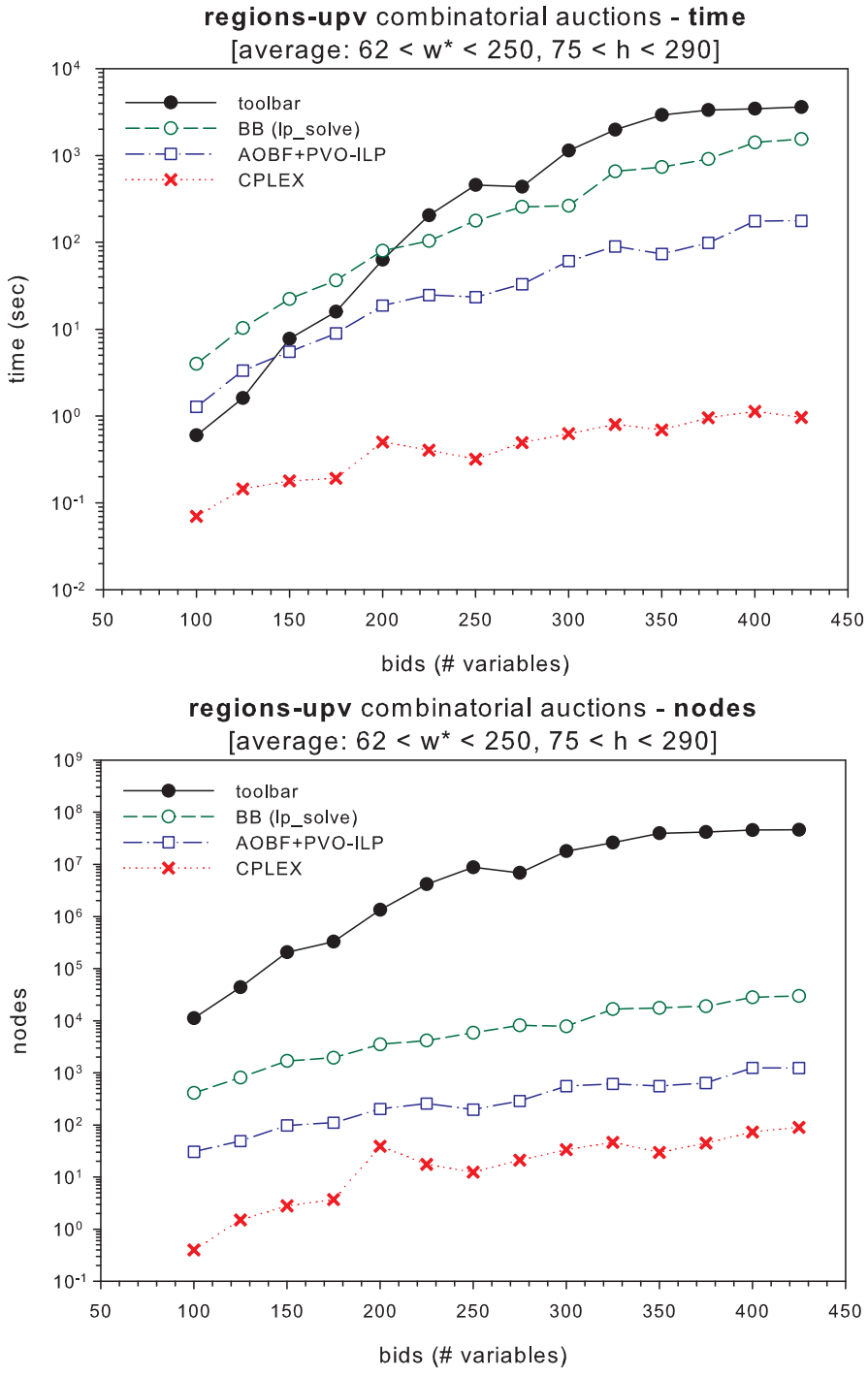


Figure 5.7: Comparison with CPLEX. CPU time in seconds (top) and number of nodes (bottom) visited for solving combinatorial auctions from the ***regions-upv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours.

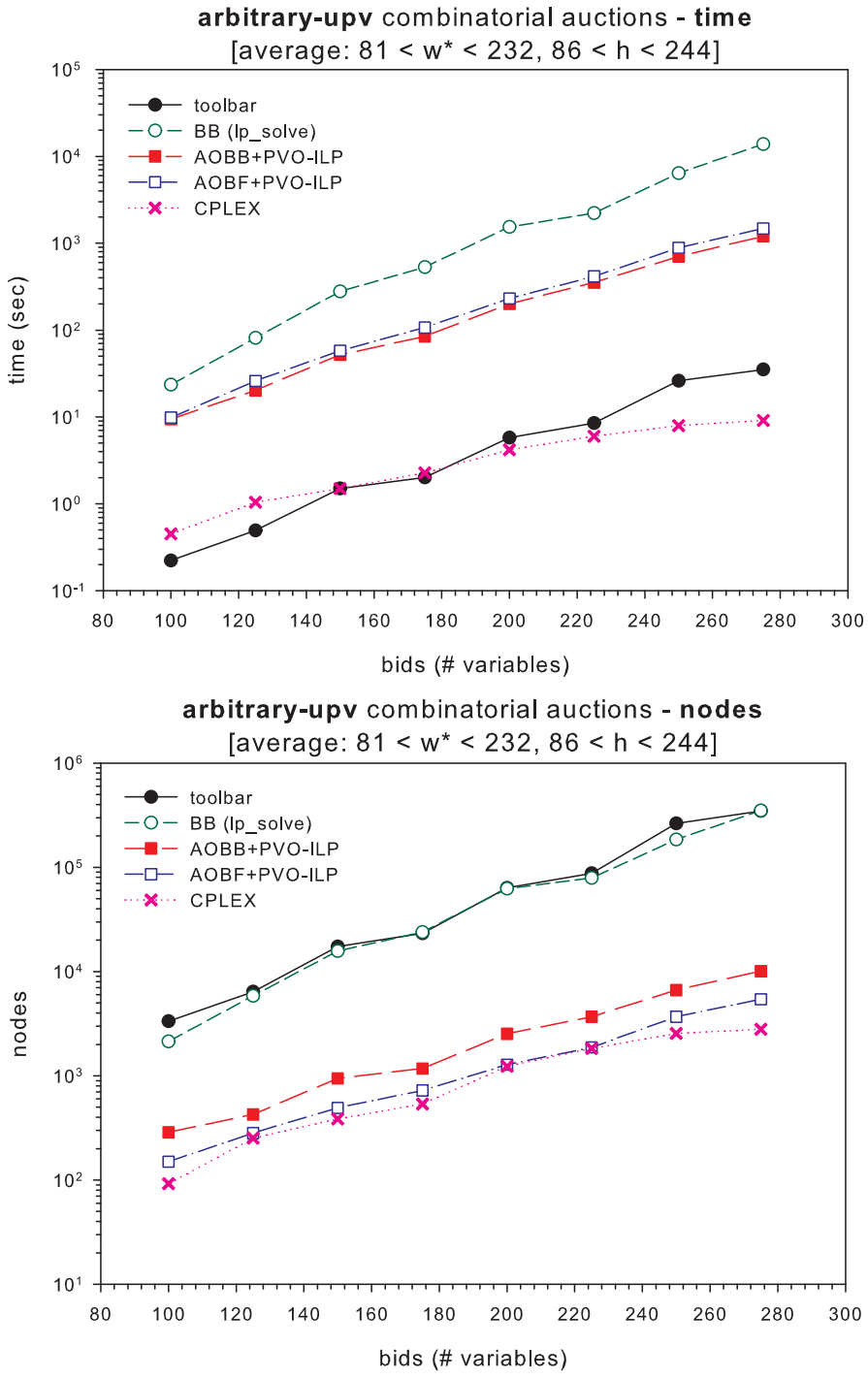


Figure 5.8: Comparison with CPLEX. CPU time in seconds (top) and number of nodes visited (bottom) for solving combinatorial auctions from the ***arbitrary-upv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours.

test suite [75]. The *regions-upv* problem instances simulate the auction of radio spectrum in which a government sells the right to use specific segments of spectrum in different geographical areas. The *arbitrary-upv* problem instances simulate the auction of various electronic components. The suffix *upv* indicates that the bid prices were drawn from a *uniform* distribution. We looked at moderate size auctions having 100 goods and increasing number of bids. The number of bids is also the number of variables in the 0-1 ILP model. Each data point represents an average over 10 instances drawn uniformly at random from the respective distribution. The header of each plot in Figures 5.5 and 5.6 shows the average induced width and depth of the pseudo trees.

AND/OR vs. OR search. When comparing the AND/OR versus OR search regimes, we observe that both depth-first and best-first AND/OR search algorithms improve considerably over the OR search algorithm, `BB`, especially when the number of bids increases and the problem instances become more difficult. In particular, the depth-first and best-first AND/OR search algorithm using partial variable orderings, `AOBB+PVO-ILP` and `AOBF+PVO-ILP`, are the winners on this domain, among the `lp_solve` based solvers. For example, on the *regions-upv* auctions with 400 bids (Figure 5.5), `AOBF+PVO-ILP` is on average about 8 times faster than `BB`. Similarly, on the *arbitrary-upv* auctions with 280 bids (Figure 5.6), the difference in running time between `AOBB+PVO-ILP` and `BB` is about 1 order of magnitude. Notice that on the *regions-upv* dataset, `toolbar` is outperformed significantly by `BB` as well as the AND/OR algorithms. On the *arbitrary-upv* dataset, `toolbar` outperforms dramatically the `lp_solve` based solvers. However, the size of the search space explored by `toolbar` is significantly larger than the ones explored by the AND/OR algorithms. Therefore, `toolbar`'s better performance in this case can be explained by the far smaller computational overhead of the arc-consistency based heuristic used, compared with the LP relaxation based heuristic.

AOBB vs. AOBF. When comparing further best-first versus depth-first AND/OR search, we see that `AOBF-ILP` (resp. `AOBF+PVO-ILP`) improve considerably over `AOBB-ILP` (resp. `AOBB+PVO-ILP`), especially on the *regions-upv* dataset. The gain observed when moving from depth-first AND/OR Branch-and-Bound to best-first AND/OR search is primarily due to the optimal cost, which bounds the horizon of best-first more effectively than for depth-first search.

Impact of caching. When looking at the impact of caching on AND/OR search, we notice that the graph search algorithms `AOBB-C-ILP` and `AOBF-C-ILP` expanded the same number of nodes as the tree search algorithms `AOBB-ILP` and `AOBF-ILP`, respectively (see Figures 5.5 and 5.6). This indicates that, for this domain, the context minimal AND/OR search graph explored is a tree. Or, the LP relaxation is very accurate in this case and the AND/OR algorithms only explore a small part of the pseudo tree, for which the corresponding context-based cache entries are actually dead-caches.

Impact of dynamic variable orderings. We can see that using dynamic variable ordering heuristics improves the performance of best-first AND/OR search only. For depth-first AND/OR search, the performance deteriorated sometimes (see for example `AOBB-ILP` vs. `AOBB+PVO-ILP` on *regions-upv* auctions in Figure 5.5).

Comparison with CPLEX. In Figures 5.7 and 5.8 we contrast the results obtained with CPLEX, `toolbar`, `BB`, `AOBB+PVO-ILP` and `AOBF+PVO-ILP` on the *regions-upv* (Figure 5.7) and *arbitrary-npv* (Figure 5.8) distributions, respectively. Clearly, we can see that CPLEX is the best performing solver on these datasets. In particular, it is several orders of magnitude faster than the `lp_solve` based solvers, especially the baseline `BB` solver. Its excellent performance is leveraged by the powerful cutting planes engine as well as the proprietary variable ordering heuristic used. Note that on the *arbitrary-upv* dataset, `toolbar`

is competitive with CPLEX only for relatively small number of bids.

regions-npv and arbitrary-npv Combinatorial Auctions

Figures 5.9 and 5.10 show the results for experiments with combinatorial auctions generated from the *regions-npv* (Figure 5.9) and *arbitrary-npv* (Figure 5.10) distributions of the CATS 2.0 suite. The bid prices of these auctions were drawn from a *normal* rather than the uniform distribution. As before, each data point represents an average over 10 random instances.

The spectrum of results is similar to what we observed for the *regions-upv* and *arbitrary-upv* auctions. The AND/OR algorithms outperformed `BB` by a significant margin. Caching had no impact on these datasets as well, namely the context minimal AND/OR graph explored was a tree (in Figures 5.9 and 5.10 for example, the curves corresponding to graph search `AOBB-C-ILP` and `AOBF-C-ILP` overlap with those corresponding to tree search `AOBB-ILP` and `AOBF-ILP`). On the *arbitrary-npv* dataset, `toolbar` outperformed again the `lp_solve` based solvers, indicating that in this case the EDAC heuristic had a far smaller overhead than the LP based one.

Figures 5.11 and 5.12 show the results obtained with CPLEX on the *regions-npv* (Figure 5.11) and *arbitrary-npv* (Figure 5.12) distributions, respectively. Clearly, we can see that CPLEX is the best performing solver on these datasets. It is several orders of magnitude faster than all other ILP solvers. On the *arbitrary-npv* auctions, `toolbar` is competitive with CPLEX only for relatively small number of bids.

5.7.2 Uncapacitated Warehouse Location Problems

In the **uncapacitated warehouse location problem** (UWLP) a company considers opening m warehouses at some candidate locations in order to supply its n existing stores. The objective is to determine which warehouse to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is

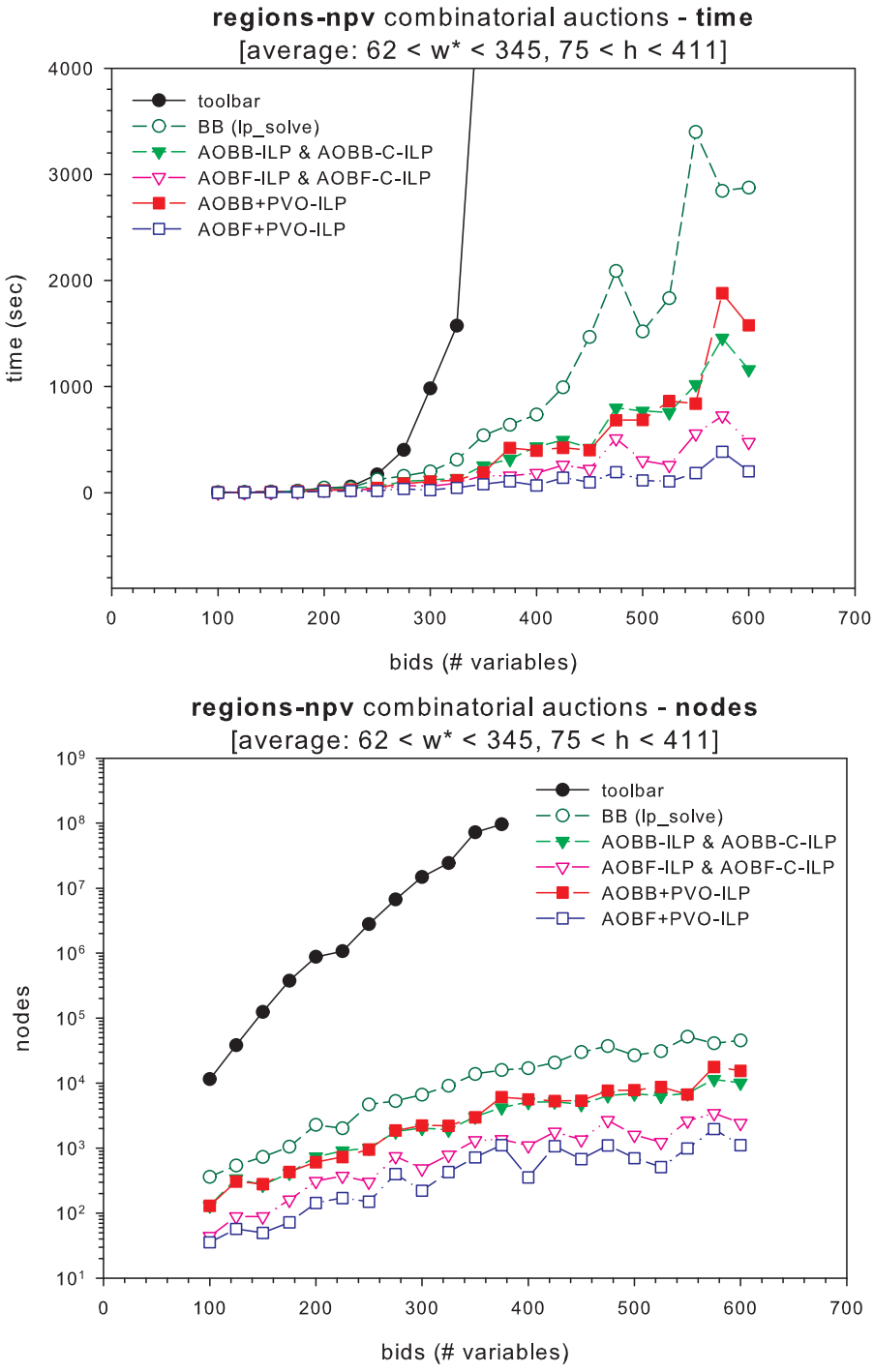


Figure 5.9: Comparing depth-first and best-first AND/OR search algorithms with static and dynamic variable orderings. CPU time in seconds (top) and number of nodes visited (bottom) for solving combinatorial auctions from the ***regions-npv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours.

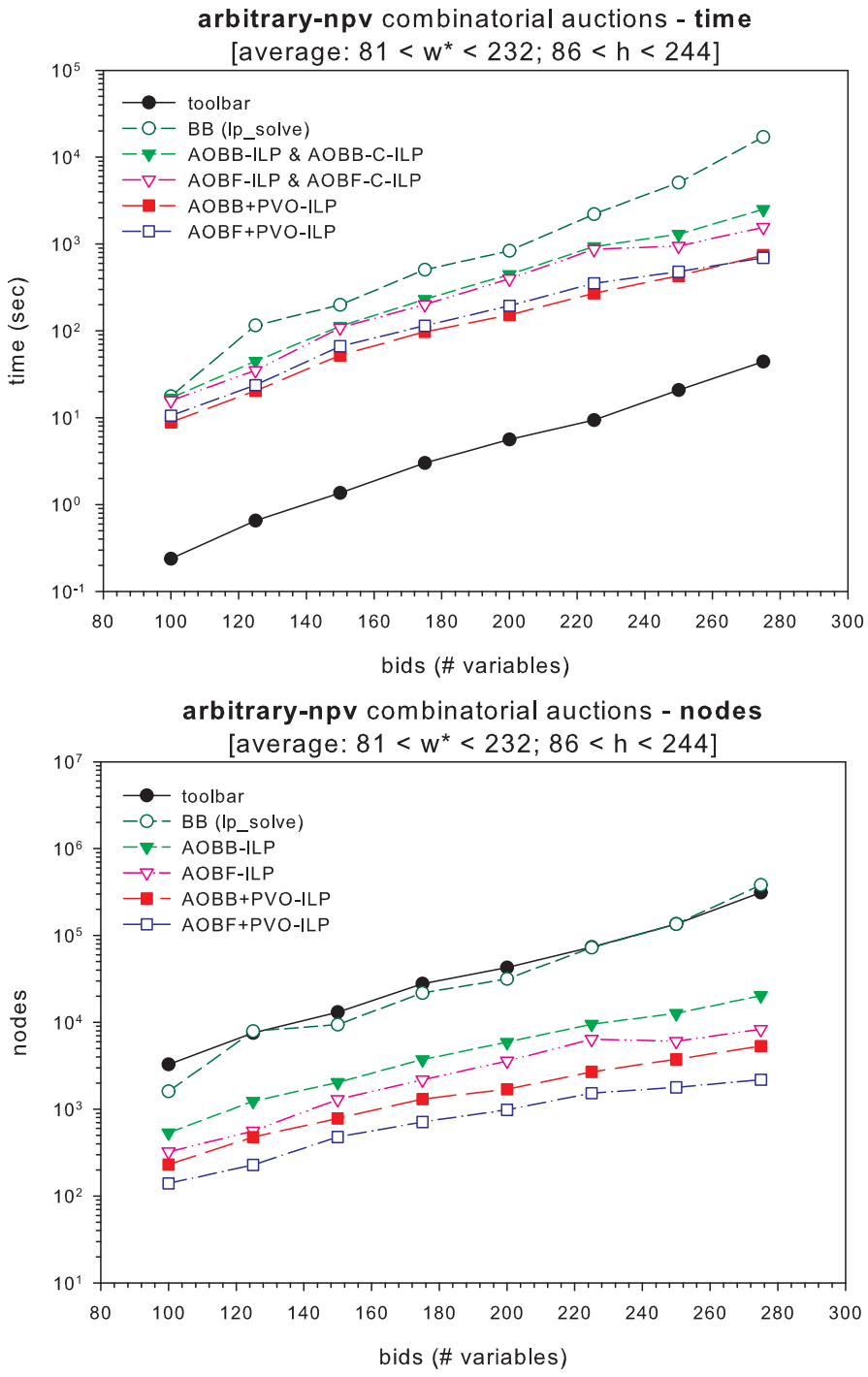


Figure 5.10: Comparing depth-first and best-first AND/OR search algorithms with static and dynamic variable orderings. CPU time in seconds (top) and number of nodes visited (bottom) for solving combinatorial auctions from the ***arbitrary-npv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours.

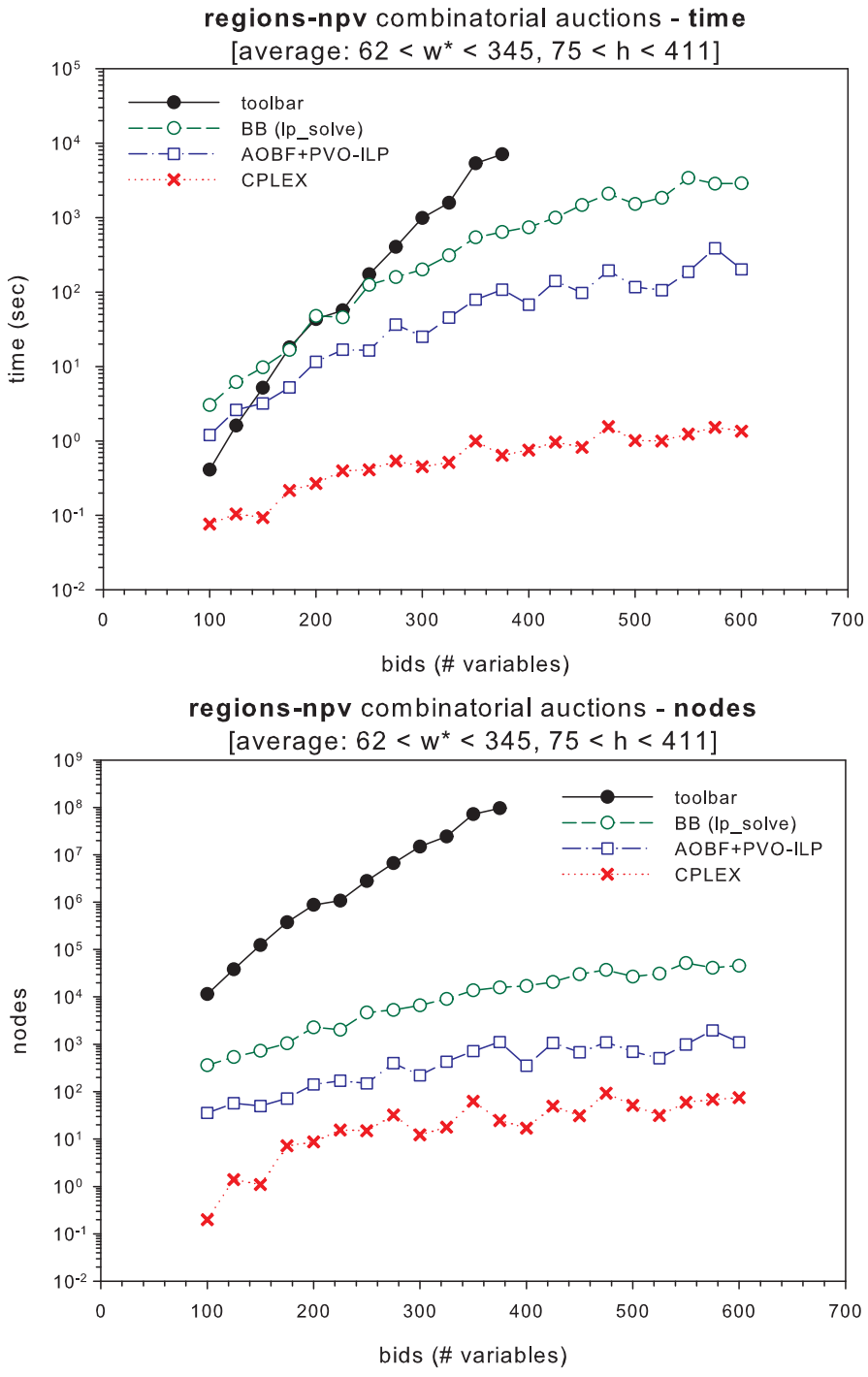


Figure 5.11: Comparison with CPLEX. CPU time in seconds (top) and number of nodes visited (bottom) for solving combinatorial auctions from the ***regions-npv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours.

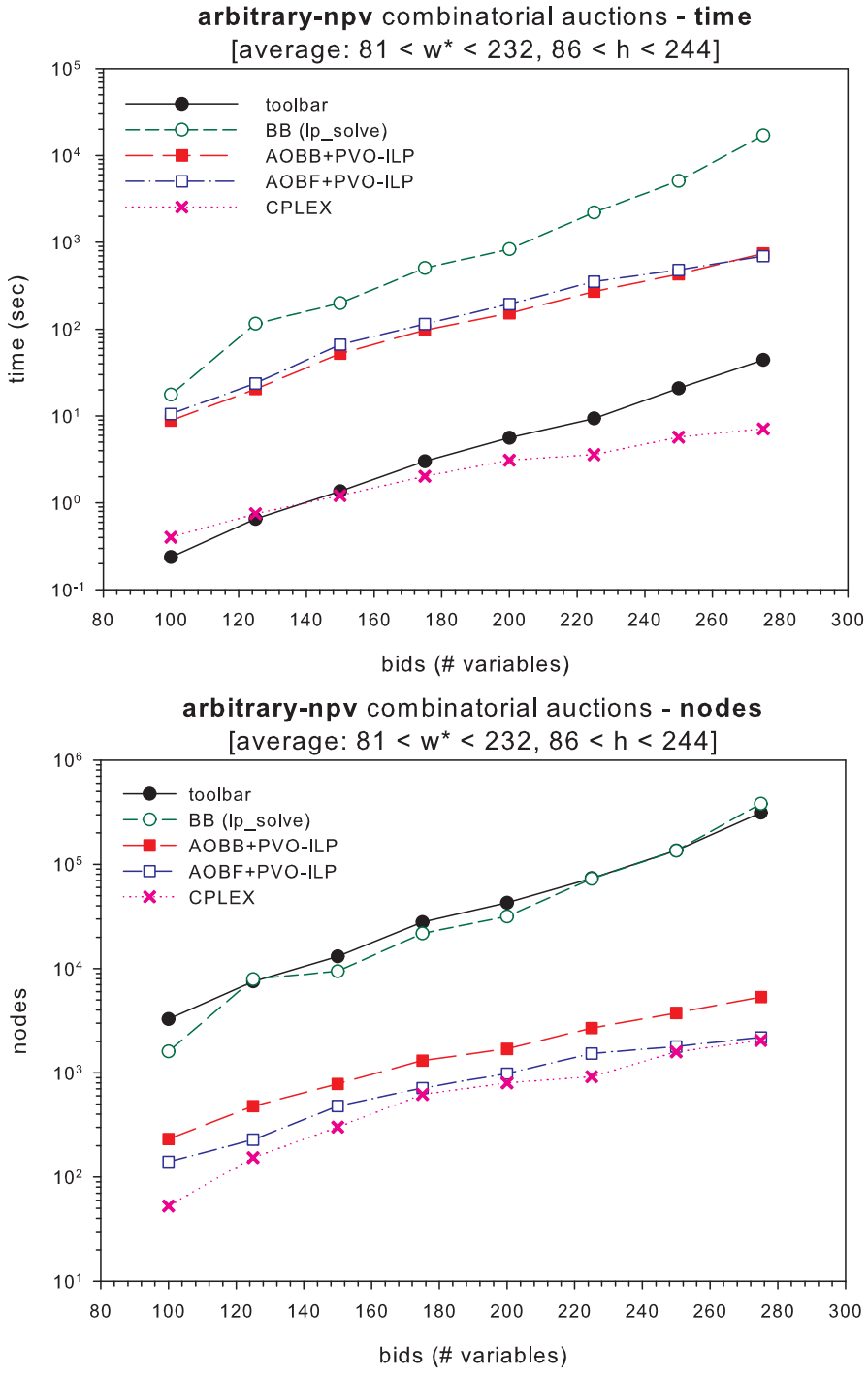


Figure 5.12: Comparison with CPLEX. CPU time in seconds (top) and number of nodes visited (bottom) for solving combinatorial auctions from the ***arbitrary-npv*** distribution with 100 goods and increasing number of bids. Time limit 3 hours.

minimized. Each store must be supplied by exactly one warehouse. The typical 0-1 ILP formulation of the problem is as follows:

$$min \sum_{j=1}^{n} \sum_{i=1}^{m} c_{ij} x_{ij} + \sum_{i=1}^{m} f_i y_i \tag{5.9}$$

$$\begin{aligned} \text{s.t.} \quad & \textstyle\sum_{i=1}^{m} x_{ij} = 1 && \forall j \in \{1..n\} \\ & x_{ij} \leq y_i && \forall j \in \{1..n\}, \forall i \in \{1..m\} \\ & x_{ij} \in \{0,1\} && \forall j \in \{1..n\}, \forall i \in \{1..m\} \\ & y_i \in \{0,1\} && \forall i \in \{1..m\} \end{aligned}$$

where f_i is the cost of opening a warehouse at location i and c_{ij} is the cost of supplying store j from the warehouse at location i.

Tables 5.2 and 5.3 display the results obtained for 30 randomly generated UWLP instances[3] with 50 warehouses, 200 and 400 stores, respectively. The warehouse opening and store supply costs were chosen uniformly randomly between 0 and 1000. These are large problems with 10,050 variables and 10,500 constraints for the `uwlp-50-200` class, and 20,050 variables and 20,400 constraints for the `uwlp-50-400` class, respectively, but having relatively shallow pseudo trees with depths of 123.

AND/OR vs. OR search. When looking at AND/OR versus OR search, we can see that in almost all test cases the AND/OR algorithms dominate `BB`. On the *uwlp-50-200-013* instance, for example, `AOBF+PVO-ILP` causes a speed-up of 186 over `BB`, exploring a search tree 1,142 times smaller. Similarly, on *uwlp-50-400-001*, `AOBB+PVO-ILP` outperforms `BB` by almost 2 orders of magnitude in terms of running time and size of the search space explored. On this domain, the best performing algorithm among the `lp_solve` based solvers is best-first `AOBF+PVO-ILP`.

AOBB vs. AOBF. When comparing best-first against depth-first AND/OR Branch-and-

[3]Problem generator from http://www.mpi-sb.mpg.de/units/ag1/projects/benchmarks/UflLib/

uwlp-50-200	BB (lp_solve)		AOBB-ILP		AOBB+PVO-ILP		AOBB-C-ILP	
(w*, h)	CPLEX		AOBF-ILP		AOBF+PVO-ILP		AOBF-C-ILP	
(n=10,050, c=10,500)	time	nodes	time	nodes	time	nodes	time	nodes
uwlp-50-200-001	48.66	86	69.74	62	25.69	20	69.69	62
(50, 123)	**1.73**	3	44.45	20	**20.25**	7	42.84	20
uwlp-50-200-003	33.14	72	48.56	59	**18.20**	17	48.56	59
(50, 123)	**1.36**	0	34.89	22	22.56	6	34.09	22
uwlp-50-200-004	61.08	142	46.39	46	17.47	10	46.42	46
(50, 123)	**0.80**	0	37.58	24	**15.49**	3	36.27	24
uwlp-50-200-005	1591.89	1,692	404.94	233	**125.81**	50	405.72	233
(50, 123)	**9.91**	81	287.64	97	145.53	37	270.99	97
uwlp-50-200-011	256.19	358	233.96	246	78.74	39	233.21	246
(50, 123)	**7.97**	37	88.22	41	**75.83**	22	83.75	41
uwlp-50-200-013	13693.76	14,846	116.19	44	78.86	24	116.25	44
(50, 123)	**8.94**	37	111.28	26	**74.53**	13	105.72	26
uwlp-50-200-017	711.04	998	123.14	118	18.17	9	124.70	118
(50, 123)	**2.15**	3	48.06	21	**16.84**	2	47.77	21
uwlp-50-200-018	1477.74	2,666	161.03	146	59.52	37	161.05	146
(50, 123)	**5.74**	8	54.58	21	**32.33**	8	52.41	21
uwlp-50-200-020	2179.39	3,668	190.77	138	68.91	36	190.81	138
(50, 123)	**7.47**	28	87.58	33	**48.33**	10	83.70	33
uwlp-50-200-021	3252.60	5,774	609.74	580	**37.63**	9	608.24	580
(50, 123)	**6.66**	25	80.55	30	46.80	7	92.08	30
uwlp-50-200-022	50.70	122	49.08	63	**17.00**	9	49.14	63
(50, 123)	**1.84**	3	38.39	26	18.17	6	37.34	26
uwlp-50-200-023	205.92	204	102.30	50	43.72	16	102.09	50
(50, 123)	**6.05**	6	60.70	19	**34.16**	5	58.50	19
uwlp-50-200-024	2177.67	3,288	125.85	71	28.19	16	125.86	71
(50, 123)	**5.52**	15	86.64	31	**25.89**	4	82.27	31
uwlp-50-200-029	14.94	10	55.33	46	15.06	5	53.28	46
(50, 123)	**1.59**	1	46.56	27	16.33	3	45.36	27
uwlp-50-200-030	21.77	42	127.39	164	15.03	5	127.59	164
(50, 123)	**0.95**	1	31.52	15	**14.09**	1	30.64	15

Table 5.2: CPU time in seconds and number of nodes visited for solving UWLP instances with 50 warehouses 200 stores, respectively. Time limit 10 hours.

uwlp-50-400	BB (lp_solve)		AOBB-ILP		AOBB+PVO-ILP		AOBB-C-ILP	
(w*, h)	CPLEX		AOBF-ILP		AOBF+PVO-ILP		AOBF-C-ILP	
(n=20,050, c=20,400)	time	nodes	time	nodes	time	nodes	time	nodes
uwlp-50-400-001	13638.55	12,548	743.75	374	106.63	29	743.68	374
(50, 123)	**10.76**	12	130.03	20	**81.63**	8	126.39	20
uwlp-50-400-004	820.89	942	1114.47	794	55.10	10	1117.55	794
(50, 123)	**6.52**	6	126.97	25	**51.85**	3	123.19	25
uwlp-50-400-005	57532.67	32,626	2719.09	617	247.03	50	2722.26	617
(50, 123)	**30.55**	58	331.87	36	**131.58**	8	313.09	36
uwlp-50-400-006	365.93	632	48.41	11	**32.31**	1	48.44	11
(50, 123)	**3.59**	0	51.62	8	32.65	1	51.95	8
uwlp-50-400-008	599.49	560	175.60	49	96.66	21	175.67	49
(50, 123)	**3.40**	0	119.28	13	**60.27**	3	116.42	13
uwlp-50-400-009	17608.98	17,262	281.02	76	97.00	9	281.30	76
(50, 123)	**9.02**	6	132.27	14	**78.05**	2	128.58	14
uwlp-50-400-011	22727.61	22,324	193.91	77	**64.28**	5	193.89	77
(50, 123)	**8.07**	7	93.11	12	64.58	4	92.06	12
uwlp-50-400-012	5468.30	4,174	671.90	307	**52.22**	4	671.77	307
(50, 123)	**4.49**	0	164.64	32	52.95	2	159.28	32
uwlp-50-400-014	-		524.69	147	248.27	41	522.25	147
(50, 123)	**22.15**	38	229.88	27	**142.83**	10	220.64	27
uwlp-50-400-019	459.39	436	85.11	18	**41.80**	3	85.13	18
(50, 123)	**3.90**	0	75.52	10	42.28	1	74.83	10
uwlp-50-400-026	232.25	252	182.35	81	59.13	14	182.52	81
(50, 123)	**4.04**	0	94.13	19	**44.05**	2	91.89	19
uwlp-50-400-027	10725.29	12,654	699.86	328	**78.44**	13	698.93	328
(50, 123)	**16.57**	50	292.28	80	84.70	10	276.21	80
uwlp-50-400-028	32669.82	29,166	508.14	175	127.45	30	507.33	175
(50, 123)	**17.16**	54	292.03	55	**127.44**	15	277.33	55
uwlp-50-400-029	22525.77	14,568	721.08	191	260.08	44	720.77	191
(50, 123)	**6.91**	4	162.83	15	**100.96**	5	158.68	15
uwlp-50-400-030	133346.24	95,866	1336.26	313	304.42	69	1339.17	313
(50, 123)	**19.80**	31	787.04	115	**240.14**	28	741.36	115

Table 5.3: CPU time in seconds and number of nodes visited for solving UWLP instances with 50 warehouses 400 stores, respectively. Time limit 10 hours.

Bound search we observe only minor savings in running time in favor of best-first search. This can be explained by the already small enough search space traversed by the algorithms, which does not leave room for additional improvements due to the optimal cost bound exploited by best-first search.

Impact of caching. When looking at the impact of caching we see again that AOBB-C-ILP and AOBF-C-ILP visited the same number of nodes as AOBB-ILP and AOBF-ILP, respectively (see columns 3 and 5 in Tables 5.2 and 5.3). This shows again that the context minimal AND/OR search graph explored by the AOBB-C-ILP and AOBF-C-ILP algorithms was a tree and therefore all cache entries were dead-caches.

Impact of dynamic variable orderings. We also observe that the dynamic variable ordering had a significant impact on performance in this case, especially for depth-first search. For example, on the *uwlp-50-200-021* instance, AOBB+PVO-ILP is 16 times faster than AOBB-ILP and expands 64 times fewer nodes. However, the difference in running time between the best-first search algorithms, AOBF-ILP and AOBF+PVO-ILP, is smaller compared to what we see for depth-first AND/OR search. This is because the search space explored by AOBF-ILP is already small enough and the savings in number of nodes caused by dynamic variable orderings cause only minor time savings.

Comparison with CPLEX. When looking at the results obtained with CPLEX (column 2 in Tables 5.2 and 5.3), we notice again its excellent performance in terms of both running time and size of the search space explored. However, we see that in some cases AOBF+PVO-ILP actually explored fewer nodes than CPLEX ($e.g.$, *uwlp-50-200-021*). This is important because it shows that the relative worse performance of AOBF+PVO-ILP versus CPLEX is due mainly to the much slower *simplex* implementation of the former, lack of cutting planes engine as well as the naive dynamic variable ordering heuristic used.

5.7.3 MAX-SAT Instances

Given a set of Boolean variables the goal of **maximum satisfiability** (MAX-SAT) is to find a truth assignment to the variables that violates the least number of clauses. We experimented with problem classes `pret` and `dubois` from the SATLIB[4] library, which were previously shown to be difficult for 0-1 ILP solvers [26].

MAX-SAT can be formulated as a 0-1 ILP [61] or pseudo-Boolean formula [124, 43]. In the 0-1 ILP model, a Boolean variable v is mapped to an integer variable x that takes value 1 when v is $True$ or 0 when it is $False$. Similarly, $\neg v$ is mapped to $1 - x$. With these mappings, a clause can be formulated as a linear inequality. For example, the clause $(v_1 \vee \neg v_2 \vee v_3)$ can be mapped to $x_1 + (1 - x_2) + x_3 \geq 1$. Here, the inequality means that the clause must be satisfied in order for the left side of the inequality to have a value no less than one.

However, a clause in a MAX-SAT may not be satisfied, so that the corresponding inequality may be violated. To address this issue, an auxiliary integer variable y is introduced to the left side of a mapped inequality. Variable $y = 1$ if the corresponding clause is unsatisfied, making the inequality valid; otherwise, $y = 0$. Since the objective is to minimize the number of violated clauses, it is equivalent to minimize the sum of the auxiliary variables that are forced to take value 1. For example, $(v_1 \vee \neg v_2 \vee v_3)$, $(v_2 \vee v_4)$ can be written as a 0-1 ILP of minimizing $z = y_1 + y_2$, subject to the constraints of $x_1 + (1 - x_2) + x_3 + y_1 \geq 1$ and $x_2 + (1 - x_4) + y_2 \geq 1$.

pret Instances

Table 5.4 shows the results for experiments with 6 *pret* instances. These are unsatisfiable instances of graph 2-coloring with parity constraints. The size of these problems is relatively small (60 variables with 160 clauses for *pret60* and 150 variables with 400 clauses for *pret150*, respectively).

[4] http://www.satlib.org/

pret (w*, h)	CPLEX BB (lp_solve 5.5)		MaxS	toolbar PBS		AOBB-ILP AOBF-ILP		AOBB+PVO-ILP AOBF+PVO-ILP		AOBB-C-ILP AOBF-C-ILP	
	time	nodes	time	time	nodes	time	nodes	time	nodes	time	nodes
pret60-40	676.94	3,926,422	9.47	53.89	7,297,773	7.88	1,255	8.41	1,216	7.38	1,216
(6, 13)	-	-		**0.00**	565	7.56	1,202	8.70	1,326	**3.58**	568
pret60-60	535.05	2,963,435	9.48	53.66	7,297,773	8.56	1,259	8.70	1,247	7.30	1,140
(6, 13)	-	-		**0.00**	495	8.08	1,184	8.31	1,206	**3.56**	538
pret60-75	402.53	2,005,738	9.37	53.52	7,297,773	6.97	1,124	6.80	1,089	6.34	1,067
(6, 13)	-	-		**0.00**	543	7.38	1,145	8.42	1,149	**3.08**	506
pret150-40	out		-	-	-	95.11	6,625	108.84	7,152	75.19	5,625
(6, 15)	-	-		**0.02**	2,592	101.78	6,535	101.97	6,246	**19.70**	1,379
pret150-60	out		-	-	-	98.88	6,851	112.64	7,347	78.25	5,813
(6, 15)	-	-		**0.01**	2,873	106.36	6,723	102.28	6,375	**19.75**	1,393
pret150-75	out		-	-	-	108.14	7,311	115.16	7,452	84.97	6,114
(6, 15)	-	-		**0.02**	2,898	98.95	6,282	103.03	6,394	**20.95**	1,430

Table 5.4: CPU time in seconds and number of nodes visited for solving `pret` MAX-SAT instances. Time limit 10 hours.

AND/OR vs. OR search. When comparing AND/OR versus OR search we see again that the AND/OR algorithms improved dramatically over `BB`. For instance, on the *pret150-75* network, `AOBB-ILP` finds the optimal solution in less than 2 minutes, whereas `BB` exceeds the 10 hour time limit. Similarly, `MaxSolver` and `toolbar` could not solve the instance within the time limit. Overall, `PBS` offers the best performance on this dataset.

AOBB vs. AOBF. The best-first AND/OR search algorithms improve sometimes considerably over the depth-first ones, especially when exploring an AND/OR graph ($e.g.$, see `AOBF-C-ILP` versus `AOBB-C-ILP` in the leftmost column of Table 5.4). Moreover, the search space explored by `AOBF-C-ILP` appears to be the smallest. This indicates that the computational overhead of `AOBF-C-ILP` is mainly due to evaluating its guiding lower bounding heuristic evaluation function.

Impact of caching. When looking at the depth-first AND/OR Branch-and-Bound graph search algorithm we only observe minor improvements due to caching. This is probably because most of the cache entries were actually dead-caches. On the other hand, best-first `AOBF-C-ILP` exploits the relatively small size of the context-minimal AND/OR graph

($i.e.$, in this case the problem structure is captured by a very small context with size 6 and a shallow pseudo tree with depth 13 or 15) and achieves the best performance among the ILP solvers.

Impact of dynamic variable orderings. We also see that the dynamic variable ordering did not have an impact on search performance for both depth-first and best-first algorithms.

Comparison with CPLEX. Both depth-first and best-first AND/OR search algorithms outperformed dramatically CPLEX on this dataset. On the *pret60-40* instance, for example, `AOBF-C-ILP` is 2 orders of magnitude faster than CPLEX. Similarly, on *pret150-40*, CPLEX exceeded the memory limit.

dubois Instances

Figure 5.13 displays the results for experiments with random *dubois* instances with increasing number of variables. These are unsatisfiable 3-SAT instances with $3 \times degree$ variables and $8 \times degree$ clauses, each of them having 3 literals. As in the previous test case, the *dubois* instances have very small contexts of size 6 and shallow pseudo trees with depths ranging from 10 to 20.

AND/OR vs. OR search. As before, we see that the AND/OR algorithms are far superior to `BB`, which could not solve any of the test instances within the 3 hour time limit. `PBS` is again the overall best performing algorithm, however it failed to solve 4 test instances: on instance *dubois130*, for which $degree = 130$, it exceeded the 3 hour time limit, whereas on instances *dubois180*, *dubois200* and *dubois260* the clause/pseudo-boolean constraint learning mechanism caused the solver to run out of memory. We note that `MaxSolver` and `toolbar` were not able to solve any of the test instances within the time limit.

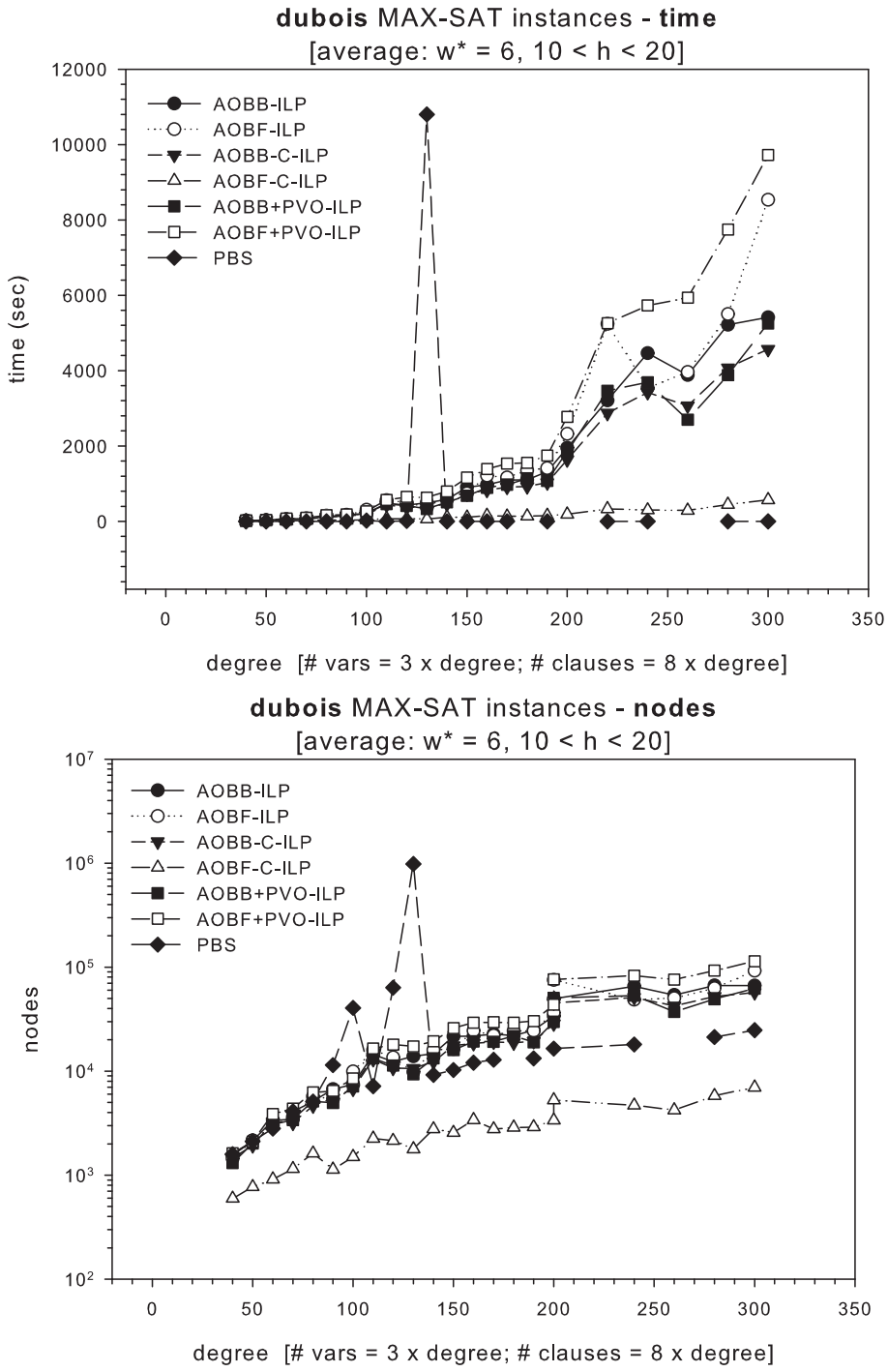


Figure 5.13: Comparing depth-first and best-first AND/OR search algorithms with static and dynamic variable orderings. CPU time in seconds (top) and number of nodes visited (bottom) for solving ***dubois*** MAX-SAT instances. Time limit 3 hours. CPLEX, `BB`, `toolbar` and `MaxSolver` were not able to solve any of the test instances within the time limit.

AOBB vs. AOBF. Best-first search outperforms again depth-first search, especially when exploring the AND/OR graph. However, the depth-first tree search algorithms AOBB-ILP and AOBB+PVO-ILP were better than the best-first tree search counterparts in this case. This was probably caused by the internal dynamic variable ordering used by AOBB-ILP and AOBB+PVO-ILP to solve independent subproblems rooted at the AND nodes in the search tree.

Impact of caching. We can see that AOBF-C-ILP takes full advantage of the relatively small context minimal AND/OR search graph and, on some of the larger instances, it outperforms its ILP competitors with up to one order of magnitude in terms of both running time and number of nodes expanded. On this dataset, AOBF-C-ILP explores the smallest search space, but its computational overhead does not pay off in terms of running time when compared with PBS. The impact of caching on AND/OR Branch-and-Bound is not that pronounced as for best-first search.

Impact of dynamic variable orderings. The dynamic variable ordering had a minor impact on depth-first AND/OR search only ($e.g.$, see AOBB+PVO-ILP versus AOBB-ILP in Figure 5.13).

Comparison with CPLEX. The performance of CPLEX was quite poor on this dataset and could not solve any of the test instances within the time limit.

5.8 Conclusion to Chapter 5

The chapter investigates the impact of the AND/OR search spaces perspective to solving optimization problems from the class of 0-1 Integer Linear Programs. In Chapters 3 and 4 we showed that the AND/OR search paradigm can improve general constraint optimization

algorithms. Here, we demonstrate empirically the benefit of AND/OR search to 0-1 ILPs.

Specifically, we extended and evaluated the depth-first and best-first AND/OR search algorithm traversing the AND/OR search tree or context minimal AND/OR graph to solving 0-1 ILPs. We also extended the algorithms with dynamic variable ordering strategies. Our empirical evaluation demonstrated on a variety of benchmark problems that the AND/OR search algorithms outperform the classic depth-first OR Branch-and-Bound sometimes by several orders of magnitude. We summarize next the most important factors influencing performance, including dynamic variable orderings, caching, as well as the search control strategy ($e.g.$, depth-first versus the best-first).

- **Depth-first versus best-first search.** Our results showed that the AND/OR search algorithms using a best-first control strategy and traversing either an AND/OR search tree or graph were able, in many cases, to improve considerably over the depth-first search ones ($e.g.$, combinatorial auctions from Figures 5.5 and 5.9, *dubois* MAX-SAT instances from Figure 5.13).

- **Impact of caching.** For problems with relatively small contexts (treewidth), the memory intensive best-first AND/OR search algorithms were shown to outperform dramatically the corresponding tree search algorithms ($e.g.$, *dubois* MAX-SAT instances from Figure 5.13). The impact of caching on the depth-first AND/OR Branch-and-Bound search algorithms was less prominent on these types of problems ($e.g.$, *pret* and *dubois* MAX-SAT instances from Table 5.4 and Figure 5.13, respectively) probably because most of the cache entries were dead-caches. For problems with very large contexts ($e.g.$, combinatorial auctions from Figures 5.5 and 5.9, UWLP instances from Tables 5.2 and 5.3) the context minimal AND/OR graph explored was a tree, and therefore caching had no impact.

- **Impact of dynamic variable orderings.** The AND/OR search approach was already shown to be powerful when used in conjunction with dynamic variable ordering

schemes in Chapter 3. Here, for 0-1 ILPs we also show that the AND/OR Branch-and-Bound with partial variable orderings sometimes outperformed the AND/OR Brach-and-Bound restricted to a static variable ordering by one order of magnitude ($e.g.$, UWLP instances from Tables 5.2 and 5.3). Similarly, best-first AND/OR search with partial variable orderings improved considerably over its counterpart using a static ordering ($e.g.$, combinatorial auctions from Figures 5.5 and 5.9).

- **AND/OR solvers versus CPLEX.** Our current implementation of the depth-first and best-first AND/OR search is far from being fully optimized with respect to commercial 0-1 ILP solvers such as CPLEX, as it relies on an open source implementation of the *simplex* algorithm, as well as a naive dynamic variable ordering heuristic. Nevertheless, we demonstrated that on selected classes of 0-1 ILPs the AND/OR algorithms outperformed CPLEX in terms of both the number of nodes explored ($e.g.$, UWLP instances from Tables 5.2 and 5.3) and CPU time ($e.g.$, `pret` MAX-SAT instances from Table 5.4).

Chapter 6

AND/OR Multi-Valued Decision Diagrams for Constraint Optimization

6.1 Introduction

The compilation of graphical models, including constraint and probabilistic networks, has recently been under intense investigation. Compilation techniques are useful when an extended off-line computation can be traded for fast real-time answers. Typically, a tractable compiled representation of the problem is desired. Since the tasks of interest are in general NP-hard, this is not always possible in the worst case. In practice, however, it is often the case that the compiled representation is much smaller than the worst case bound, as was observed for Ordered Binary Decision Diagrams (OBDDs) [13] which are extensively used in hardware and software verification.

In the context of constraint networks, compilation schemes are very useful for interactive solving or product configuration type problems [45, 52]. These are combinatorial problems where a compact representation of the feasible set of solutions is necessary. The system has to be *complete* (to represent all set of solutions), *backtrack-free* (to never encounter dead-ends) and *real-time* (to provide fast answers).

Contribution

In this chapter we present a compilation scheme for constraint optimization, which has been of interest recently in the context of post-optimality analysis [53]. Our goal is to obtain a compact representation of the set of optimal solutions. Our approach is based on three main ideas: (1) AND/OR search spaces for graphical models [38]. Their key feature is the exploitation of problem structure during search, sometimes yielding exponential improvement over structure-blind search methods. (2) Branch-and-Bound search for optimization, applied to AND/OR search spaces [79, 82]. (3) Reduction rules similar to OBDDs, that lead to the compilation of the search algorithm trace into an AND/OR Multi-Valued Decision Diagram (AOMDD) [89].

The novelty over previous results consists in: (1) The treatment of general weighted graphs based on cost functions, rather than constraints. (2) A top down search based approach for generating the AOMDD, rather than Variable Elimination based as in [89]. (3) Extensive experimental evaluation that proves the efficiency of the weighted AOMDD. We show that the compilation scheme can often be accomplished relatively efficiently and that we sometimes get a substantial reduction beyond the initial trace of state-of-the-art search algorithms.

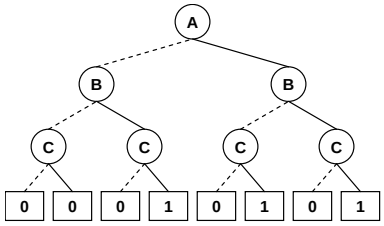
The research presented in this chapter is based in part on [107].

Chapter Outline

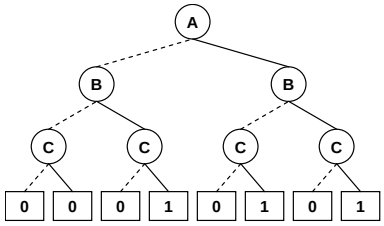
The chapter is structured as follows. Sections 6.2 and 6.3 provide background on Ordered Binary Decision Diagrams and AND/OR Multi-Valued Decision Diagrams. In Section 6.4 we present the new search based compile algorithm for the optimal solution set to a COP. Section 6.5 is dedicated to an extensive empirical evaluation that proves the efficiency of the AOMDD data-structure for optimization, while Section 6.6 provides concluding remarks.

A	B	C	f(ABC)
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**0**
1	1	1	**1**

(a) Table



(b) Unordered tree



(c) Ordered tree

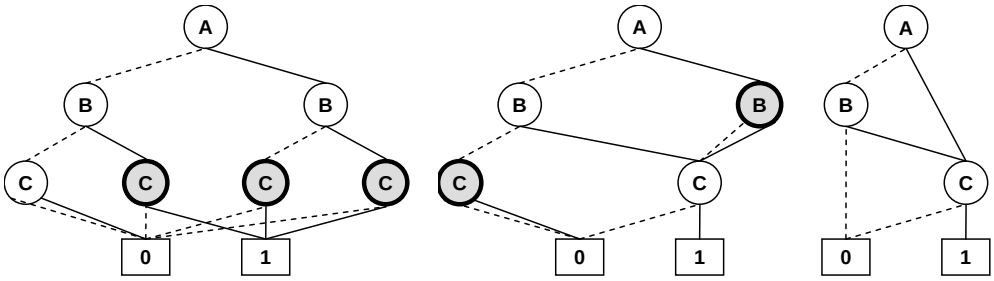
Figure 6.1: Boolean function representations

6.2 Review of Binary Decision Diagrams

Decision diagrams are widely used in many areas of research to represent decision processes. In particular, they can be used to represent functions. Due to the fundamental importance of Boolean functions, a lot of effort has been dedicated to the study of *Binary Decision Diagrams* (BDDs), which are extensively used in formal verification [18, 92].

A BDD is a representation of a Boolean function. Given $\mathbf{B} = \{0, 1\}$, a Boolean function $f : \mathbf{B}^n \rightarrow \mathbf{B}$, has n arguments, $X_1, \cdots, X_n$, which are Boolean variables, and takes Boolean values. A Boolean function can be represented by a table (see Figure 6.1(a)), but this is exponential in n, and so is the binary tree representation in Figure 6.1(b). The goal is to have a compact representation, that also supports efficient operations between functions. *Ordered Binary Decision Diagrams* (OBDDs) [13] provide such a framework by imposing the same order to the variables along each path in the binary tree, and then applying the following two reduction rules exhaustively: (1) *isomorphism*: merge nodes that have the same label and the same respective children; (2) *redundancy*: eliminate nodes whose low (zero) and high (one) edges point to the same node, and connect the parent of removed node directly to the child of removed node.

Example 23 *Figure 6.2(a) shows the binary tree from Figure 6.1(c) after the isomorphic terminal nodes (leaves) have been merged. The highlighted nodes, labeled with C, are also isomorphic, and Figure 6.2(b) shows the result after they are merged. Now, the highlighted nodes labeled with C and B are redundant, and removing them gives the OBDD in Figure*



(a) Isomorphism (b) Redundancy (c) OBDD

Figure 6.2: Reduction rules

6.2(c).

6.3 Weighted AND/OR Multi-Valued Decision Diagrams

The context minimal AND/OR graph described in Chapter 4 offers an effective way of identifying some unifiable nodes during the execution of the search algorithm. However, merging based on context is not complete, $i.e.$ there may still be unifiable nodes in the search graph that do not have identical contexts. The context-based merging uses only information available from the ancestors in the pseudo tree. If all the information from the descendants would also be available, it could lead to the identification of more unifiable nodes. This comes at a higher cost, however, since information from descendants in the pseudo tree means that the entire associated subproblem has to be solved. Orthogonal to the problem of unification, some of the nodes in an AND/OR search graph may be redundant, for example when the set of solutions rooted at variable X_i is not dependent on the specific value assigned to X_i.

The above criteria suggest that once an AND/OR search graph is available ($e.g.$, after search terminates, and its trace is saved) reduction rules based on *isomorphism* and *redundancy* (similar to OBDDs) can be applied further, reducing the size of the AND/OR search graph that was explicated by search. In order to apply the reduction rules, it is convenient to group each OR node and its children into a *meta-node* [89]:

DEFINITION **44 (meta-node)** *A meta-node v in a weighted AND/OR search graph consists of an OR node labeled $var(v) = X_i$ and its k_i AND children labeled $x_{i_1}, ..., x_{i_{k_i}}$ that correspond to its value assignments. Each AND node labeled x_{i_j} points to a list of child meta-nodes, $u.children_j$, and also stores the weight $w(X_i, x_{i_j})$.*

The reduction rules are straightforward. Two meta-nodes are *isomorphic* if they have the same variable label and the same respective lists of children and weights. A meta-node is *redundant* if all its lists of children and weights are respectively identical. Formally,

DEFINITION **45 (isomorphic meta-nodes)** *Given a weighted AND/OR search graph $\mathcal{G}$ represented with meta-nodes, two meta-nodes u and v having $var(u) = var(v) = X$ and $|D(X)| = k$ are isomorphic iff:*

1. $u.childreni = v.children_i, \forall i \in \{1, ..., k\}$ *and*
2. $w^u(X, x_i) = w^v(X, x_i), \forall i \in \{1, ..., k\}$, *where w^u, w^v are the weights of u and v, respectively).*

DEFINITION **46 (redundant meta-node)** *Given a weighted AND/OR search graph $\mathcal{G}$ represented with meta-nodes, a meta-node u with $var(u) = X$ and $|D_X| = k$ is redundant iff:*

1. $u.children_1 = ... = u.children_k$ *and*
2. $w(X, x_1) = ... = w(X, x_k)$.

When reduction rules are applied exhaustively to an AND/OR search graph, the result is an AND/OR Multi-Valued Decision Diagram (AOMDD). The AOMDD data structure for constraint networks (where weights are all 1) was introduced in [89], along with a Variable Elimination type algorithm to generate it, based on the *apply* operator, similar to the OBDD case.

DEFINITION **47 (AOMDD)** *An AND/OR Multi-Valued Decision Diagram (AOMDD) is a weighted AND/OR search graph that is completely reduced by isomorphic merging and redundancy removal, namely:*

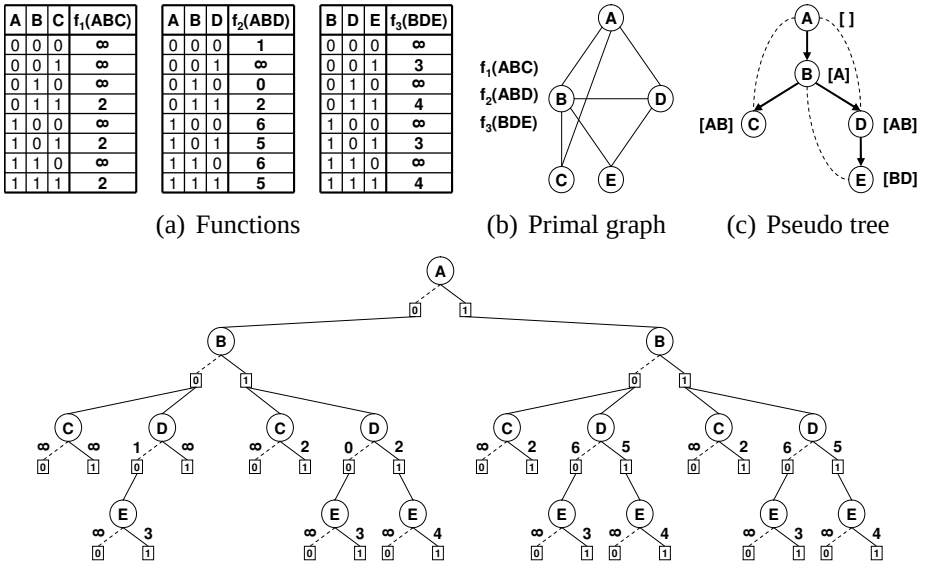
1. *it contains no isomorphic meta-nodes; and*

2. *it contains no redundant meta-nodes.*

An example of a AOMDD appears in Figure 6.4(b), representing the exhaustive reduction of the context minimal AND/OR graph in Figure 6.4(a). The terminal nodes labeled with $\mathbf{0}$ and $\mathbf{1}$ denote inconsistent and consistent assignments, respectively. The AOMDD can be understood as a collection of MDDs (Multi-Valued Decision Diagrams, based on a chain pseudo tree), each based on a path in the underlying pseudo tree, and synchronized on their common variables. In this example, f_1 is identical to the function in Figure 6.1(a), and the portion of the AOMDD corresponding to variables A, B, C is identical to the OBDD in Figure 6.2(c). This is also because when $A = 1$, B is redundant for f_2 and f_3, so the common portion of the OBDDs corresponding to ABC and $ABDE$ (namely that on AB is the same). Note that when $A = 1$, B is redundant and its common list of children becomes the list of children for $A = 1$, namely the problem already splits into two independent components after $A = 1$, even though this can not be read from the pseudo tree in Figure 6.3(c).

6.4 Using AND/OR Search to Generate AOMDDs

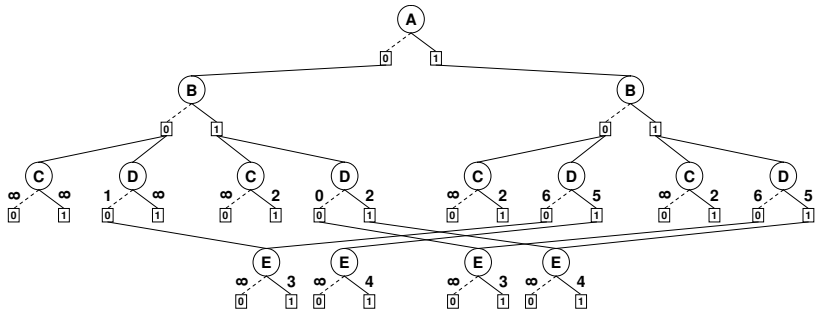
In this section we extend the AOMDD to the case of weighted graphs which captures a COP. We also propose a generation algorithm based on AND/OR Branch-and-Bound search with context based caching. More specifically, we are not interested in an AOMDD that represents all consistent assignments, but rather in one that represents only the optimal assignments (solutions of a COP).



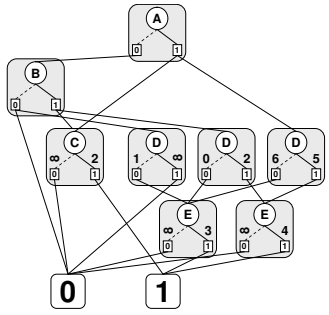
(a) Functions (b) Primal graph (c) Pseudo tree

(d) AND/OR search tree

Figure 6.3: AND/OR search tree for COP



(a) Context minimal AND/OR graph



(b) AOMDD

Figure 6.4: AND/OR graphs for COP

We next define the AOMDD describing the set of optimal solutions to a COP and present a general scheme for generating these compiled data-structures.

DEFINITION **48** *Given a set of tuples S over variables $\mathbf{X}$ and a tree $\mathcal{T}$ over $\mathbf{X}$, $\mathcal{T}$ expresses S iff there exists an AND/OR tree guided by $\mathcal{T}$ that expresses all and only tuples in S.*

It can be shown that:

PROPOSITION **3** *If $\mathcal{T}$ is a pseudo tree of a COP $\mathcal{P}$, then $\mathcal{T}$ can be used to express S^{opt}, the set of optimal solutions of $\mathcal{P}$.*

Proof. Let $\mathcal{T}$ be a pseudo tree with root X and two child nodes Y and Z, respectively. Assume that S^{opt} contains two optimal solutions associated with the tuples $(X = x, Y = y, Z = x)$ and $(X = x, Y = y_1, Z = x_1)$, respectively. The AND/OR tree relative to $\mathcal{T}$ that expresses S^{opt}, $\mathcal{S}_{\mathcal{T}}$, contains the AND node $\langle X, x\rangle$ with two OR child nodes labeled Y and Z, each of them with two AND children, namely $\{\langle Y, y\rangle, \langle Y, y_1\rangle\}$ and $\{\langle Z, z\rangle, \langle Z, z_1\rangle\}$, respectively. Yet, the tuples $(X = x, Y = y, Z = z_1)$ and $(X = x, Y = y_1, Z = z)$ are not optimal solutions, while the AND/OR tree expresses them. This is a contradiction since $\mathcal{S}_{\mathcal{T}}$, by definition, expresses only optimal solutions. □

Therefore, the following is well defined:

DEFINITION **49** *Given a COP $\mathcal{P}$, its set of optimal solutions S^{opt} and a pseudo tree $\mathcal{T}$ of $\mathcal{P}$, its AOMDD$^{opt}_{\mathcal{T}}$ is the AOMDD that expresses all and only S^{opt} relative to $\mathcal{T}$.*

The target is to generate AOMDD$^{opt}_{\mathcal{T}}$ of a COP. The idea is to use a pseudo tree $\mathcal{T}$ that can express all solutions and explore a subset of its context minimal AND/OR graph, $\mathcal{G}_{\mathcal{T}}$ that contains all the optimal solutions and then process it so that it will represent only optimal solutions and be completely reduced relative to isomorphism and redundancy. Therefore, any search algorithm for optimal solutions that explores the context minimal graph can be

used to generate the initial trace. The better the algorithm we use, the more efficient the procedure would be because the initial trace will be tight around the context minimal graph that is restricted to the optimal solutions.

6.4.1 The Search Based Compile Algorithm

The compilation algorithm, called AOBB-COMPILE, is described in Algorithm 13. It extends the AND/OR Branch-and-Bound algorithm with context based caching (AOBB-C) described in Chapter 4 by compiling the trace of the search into an AND/OR Multi-Valued Decision Diagram representing all optimal solutions to the input COP instance.

The algorithm is based on two mutually recursive steps, similar to AOBB-C: EXPAND and PROPAGATE which call each other until the search terminates. The fringe of the search is maintained by a stack called OPEN. The current node is n, its parent p, and the current path π_n. The children of the current node in the AND/OR search graph are denoted by $succ(n)$. The AND/OR decision diagram being constructed is denoted by AOMDD. Each node u in the AND/OR search graph has a pointer, denoted by $u.metanode$, to the corresponding meta-node in AOMDD.

In the EXPAND step, when the current OR node n is expanded, AOBB-COMPILE creates a new meta-node corresponding to n and adds it to AOMDD. If n is already present in cache, then AOBB-COMPILE ensures that the meta-node corresponding to n's parent in the context minimal search graph points to the meta-node that was created when n was first expanded.

In the PROPAGATE step when node values are propagated backwards, the algorithm also attempts to reduce the diagram by removing isomorphic meta-nodes. Specifically, if n is the current OR node being evaluated and if there exists a meta-node m which is isomorphic with $n.metanode$, then the parents of $n.metanode$ in the AOMDD are updated to point to m instead of $n.metanode$, which is then removed from the diagram. Every meta-node in AOMDD also records the optimal cost solution to the problem below it.

Algorithm 13: AOBB-COMPILE

Data: A COP instance $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, pseudo-tree $\mathcal{T}$, root s, heuristic function f_h.
Result: AOMDD containing the optimal solutions to $\mathcal{P}$.

$v(X_1) \leftarrow \infty$; $OPEN \leftarrow \{X_1\}$;$AOMDD \leftarrow \emptyset$; // Initialize
while $OPEN \neq \emptyset$ **do**
  $n \leftarrow top(OPEN)$; remove n from OPEN
  let π_n be the assignment along the path from the root to n
  if *n is an OR node, labeled X_i* **then** // **EXPAND**
    if $Cache(n, context(X_i)) \neq \emptyset$ **then**
      $v(n) \leftarrow Cache(n, context(X_i))$
      $succ(n) \leftarrow \emptyset$
      let $p = \langle X_j, x_j \rangle$ be the AND parent of n in the AND/OR search graph
      $p.metanode.children_{x_j} \leftarrow p.metanode.children_{x_j} \cup \{n.metanode\}$
    else
      $succ(n) \leftarrow \{\langle X_i, x_i \rangle | \langle X_i, x_i \rangle$ is consistent with $\pi_n\}$
      for $\langle X_i, x_i \rangle \in succ(n)$ **do**
        $v(\langle X_i, x_i \rangle) \leftarrow 0$; $h(\langle X_i, x_i \rangle) \leftarrow$ `heuristic(`X_i`,`x_i`)`
        $w(X_i, x_i) \leftarrow \sum_{f \in F, X_i \in scope(f)} f(\pi_n)$
      create a new meta-node m for X_i and add it to AOMDD
      let $p = \langle X_j, x_j \rangle$ be the AND parent of n in the AND/OR search graph
      $p.metanode.children_{x_j} \leftarrow p.metanode.children_{x_j} \cup \{m\}$
    Add $succ(n)$ on top of $OPEN$
  else if *n is an AND node, labeled $\langle X_i, x_i \rangle$* **then**
    for $a \in ancestors(X_i, x_i)$ **do**
      if *(a is OR)* ***and*** *($f(T_a) > v(a)$)* **then**
        $n.deadend \leftarrow true$
        $n.metanode.children_{x_i} \leftarrow UNSOLVED$
        break
    if $n.deadend == false$ **then**
      $succ(n) \leftarrow \{X_j | X_j \in children_T(X_i)\}$
      $v(X_j) \leftarrow \infty$; $h(X_j) \leftarrow$ `heuristic(`X_j`)`
      Add $succ(n)$ on top of $OPEN$
      if $succ(n) == \emptyset$ **then**
        $n.metanode.children_{x_i} \leftarrow SOLVED$
  while $succ(n) == \emptyset$ **do** // **PROPAGATE**
    let p be the parent of n
    if *n is an OR node, labeled X_i* **then**
      if $X_i == X_1$ **then** // Search is complete
        return $AOMDD$
      $Cache(n, context(X_i)) \leftarrow v(n)$
      $v(p) \leftarrow v(p) + v(n)$
      $n.metanode.value \leftarrow v(n)$
      if `findIsomorphism(`$n.metanode$`)` $== true$ **then**
        let m be the meta-node isomorphic with $n.metanode$
        redirect the links of $n.metanode$'s parents in AOMDD to point to m
        $AOMDD \leftarrow AOMDD - \{n.metanode\}$
    if *n is an AND node, labeled $\langle X_i, x_i \rangle$* **then**
      $v(p) \leftarrow min(v(p), w(X_i, x_i) + v(n))$
    remove n from $succ(p)$
    $n \leftarrow p$

The compiled AOMDD may contain sub-optimal solutions that were visited during the Branch-and-Bound search but were not pruned. Therefore, a second pass over the decision diagram is necessary to remove any path which does not appear in any optimal solution.

Specifically, AOBB-COMPILE traverses the AOMDD in a depth-first manner and, for every meta-node u along the current path from the root, it prunes $u.children_j$ from the diagram if $(\sum_{u' \in u.children_j} v(u') + w(X_i, x_{i_j})) > v(u)$, namely the optimal cost solution to the problem below the j child of u is not better than the optimal cost at u.

THEOREM **13** *Given a COP instance $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$ and a pseudo tree $\mathcal{T}$ of $\mathcal{P}$, the AOMDD generated by AOBB-COMPILE along $\mathcal{T}$ is AOMDD$^{opt}_{\mathcal{T}}$.*

Proof. Follows immediately from Proposition 3. □

The complexity of AOBB-COMPILE is bounded time and space by the trace generated, which is $O(n \cdot exp(w^*))$. However, the heuristic evaluation function used by the AND/OR Branch-and-Bound typically restricts the trace far below this complexity bound.

6.5 Experiments

In this section we evaluate empirically the compilation scheme on two common classes of optimization problems: Weighted CSPs (WCSP) [9] and 0-1 Integer Linear Programs (0-1 ILP) [95]. In our experiments we compiled the search trace relative to isomorphic meta-nodes only, without removing redundant nodes. Also we did not perform the second top-down pass over the diagram to remove additional sub-optimal solutions.

6.5.1 Weighted CSPs

We consider the compilation algorithm based on the AND/OR Branch-and-Bound algorithm with pre-compiled mini-bucket heuristics and full caching introduced in Chapter 4 and denoted by AOBB-C+SMB(i). The parameter i represents the mini-bucket i-bound and controls the accuracy of the heuristic.

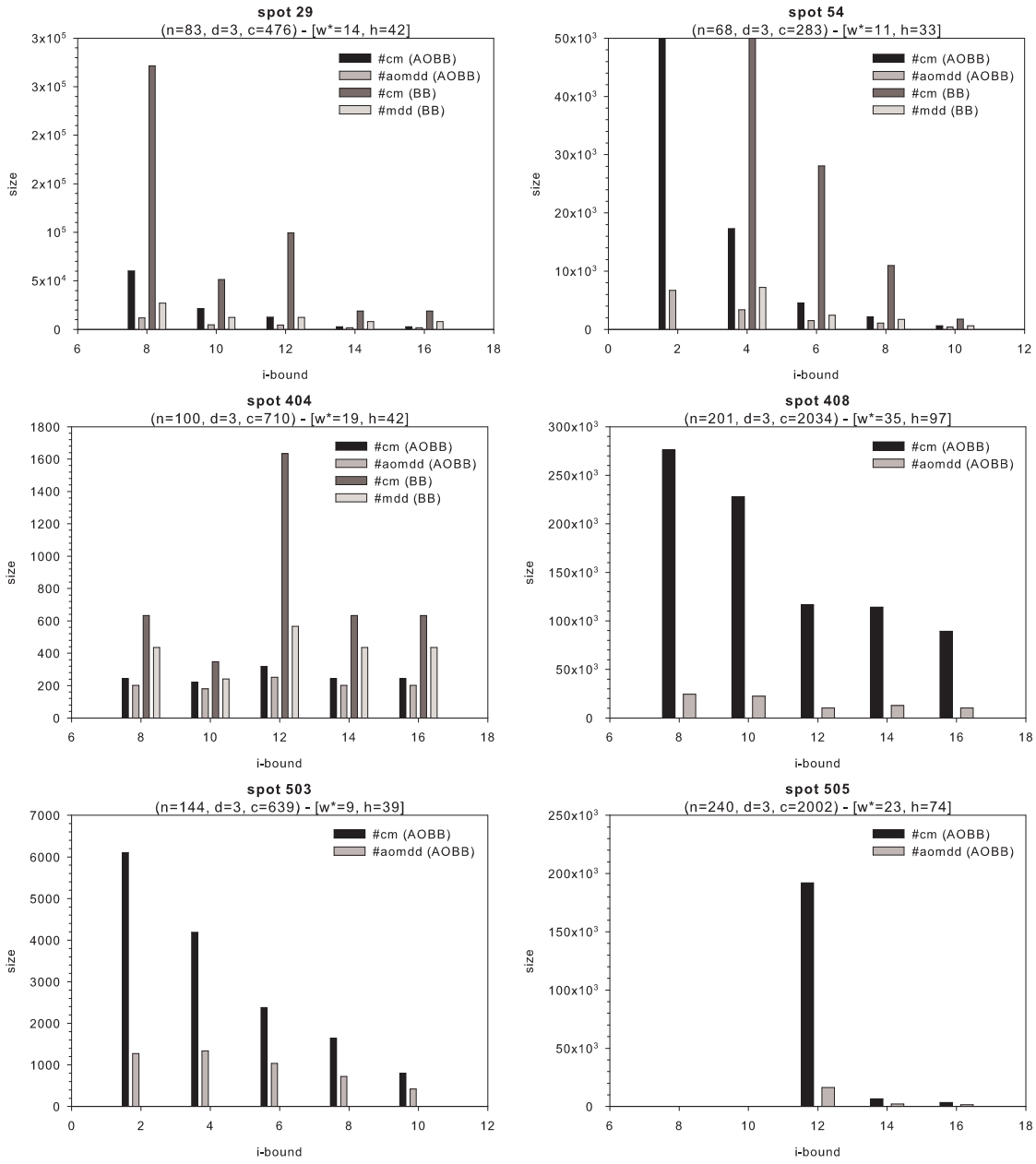


Figure 6.5: The trace of AND/OR Branch-and-Bound search (#cm) versus the AOMDD size (#aomdd) for the SPOT5 networks. Compilation time limit 1 hour.

AOBB-C+SMB(i)															
iscas	n	w*	i=10			i=12			i=14			i=16			
	c	h	time	#cm	#aomdd	time	#cm	#aomdd	time	#cm	#aomdd	time	#cm	#aomdd	
s386	172	19	0.50	2,420	811	0.17	1,132	558	0.21	527	360	0.38	527	360	
	172	44		ratio =	**2.98**		ratio =	**2.03**		ratio =	**1.46**		ratio =	**1.46**	
s953	440	66		-			-		981.20	186,658	37,084	22.46	22,053	9,847	
	464	101								ratio =	**5.03**		ratio =	**2.24**	
s1423	748	24	21.12	21,863	9,389	7.47	13,393	6,515	5.09	10,523	6,043	2.01	5,754	4,316	
	751	54		ratio =	**2.33**		ratio =	**2.06**		ratio =	**1.74**		ratio =	**1.33**	
s1488	667	47	250.18	83,927	20,774	4.48	15,008	3,929	10.72	23,872	5,375	5.54	5,830	3,246	
	667	67		ratio =	**4.04**		ratio =	**3.82**		ratio =	**4.44**		ratio =	**1.80**	
s1494	661	48	138.61	63,856	18,501	387.73	125,030	22,393	37.78	31,355	11,546	39.75	30,610	12,467	
	661	69		ratio =	**3.45**		ratio =	**5.58**		ratio =	**2.72**		ratio =	**2.46**	
c432	432	27	1867.49	395,766	41,964	1.29	7,551	4,024	1.30	7,112	3,693	0.74	1,120	881	
	432	45		ratio =	**9.43**		ratio =	**1.88**		ratio =	**1.93**		ratio =	**1.27**	
c499	499	23	363.78	93,936	33,157	6.66	12,582	7,051	271.26	88,131	23,502	16.75	17,714	9,536	
	499	74		ratio =	**2.83**		ratio =	**1.78**		ratio =	**3.75**		ratio =	**1.86**	

Table 6.1: CPU time in seconds, the trace of AND/OR Branch-and-Bound search (#cm) and the AOMDD size (#aomdd) for the ISCAS'89 circuits. Compilation time limit 1 hour.

For each test instance we report the number of OR nodes in the context minimal AND/OR search graph (#cm) visited by AOBB-C+SMB(i), and the number of meta-nodes in the resulting AND/OR decision diagram (#aomdd), as well as their ratio defined as $ratio = \frac{\#cm}{\#aomdd}$. In some cases we also report the compilation time. We record the number of variables (n), maximum domain size (d), the number of constraints (c), the depth of the pseudo-trees (h) and the induced width of the graphs (w^*) obtained for the test instances. The pseudo trees were generated using the min-fill heuristic described in Chapter 3.

Earth Observing Satellites

Figure 6.5 displays the results for experiments with 6 SPOT5 networks described in Chapter 3. Each subgraph depicts the trace of AOBB-C+SMB(i) and the size of the resulting AND/OR decision diagram as a function of the i-bound of the mini-bucket heuristic. For comparison, we also include the results obtained with the OR version of the compilation scheme that explores the traditional OR search space.

We observe that the resulting AOMDD is substantially smaller than the context minimal AND/OR graph traversed by AOBB-C+SMB(i), especially for relatively small i-bounds that generate relatively weak heuristic estimates. For instance, on the 408 network, we were able to compile an AOMDD 11 times smaller than the AND/OR search graph explored by AOBB-C+SMB(8). As the i-bound increases, the heuristic estimates become

AOBB-C+SMB(i)														
planning	n	w*	i=6			i=8			i=10			i=12		
	c	h	time	#cm	#aomdd	time	#cm	#aomdd	time	#cm	#aomdd	time	#cm	#aomdd
bwt3ac	45	16	77.45	28,558	12,152	45.76	22,475	11,106	8.92	3,878	2,537	99.00	1,775	1,252
d=11	301	34		ratio =	**2.35**		ratio =	**2.02**		ratio =	**1.53**		ratio =	**1.42**
bwt3bc	45	11	54.22	23,560	10,544	29.62	18,734	9,422	8.61	3,455	2,243	85.73	1,599	1,141
d=11	301	33		ratio =	**2.23**		ratio =	**1.99**		ratio =	**1.54**		ratio =	**1.40**
bwt3cc	45	19	32.55	19,643	9,122	20.03	15,696	8,149	8.51	3,113	2,046	85.57	935	731
d=11	301	42		ratio =	**2.15**		ratio =	**1.93**		ratio =	**1.52**		ratio =	**1.28**
depot01ac	66	14	1.45	7,420	2,504	0.73	4,056	1,995	0.42	1,214	830	1.48	506	432
d=5	298	33		ratio =	**2.96**		ratio =	**2.03**		ratio =	**1.46**		ratio =	**1.17**
depot01bc	66	14	1.31	7,068	2,358	0.55	3,333	1,641	0.39	1,316	886	1.47	514	432
d=5	298	33		ratio =	**3.00**		ratio =	**2.03**		ratio =	**1.49**		ratio =	**1.19**
depot01cc	66	14	1.36	7,156	2,411	0.82	4,333	2,196	0.38	1,262	841	1.47	269	219
d=5	298	33		ratio =	**2.97**		ratio =	**1.97**		ratio =	**1.50**		ratio =	**1.23**
			i=2			i=4			i=6			i=8		
driverlog01ac	71	9	1.37	7,490	2,134	0.41	3,143	1,412	0.05	279	237	0.10	451	331
d=4	271	38		ratio =	**3.51**		ratio =	**2.23**		ratio =	**1.18**		ratio =	**1.36**
driverlog01bc	71	9	1.36	7,447	2,128	0.42	3,098	1,389	0.04	231	210	0.07	247	212
d=4	271	38		ratio =	**3.50**		ratio =	**2.23**		ratio =	**1.10**		ratio =	**1.17**
driverlog01cc	71	9	1.61	7,741	2,185	0.10	883	622	0.04	279	237	0.07	295	239
d=4	271	38		ratio =	**3.54**		ratio =	**1.42**		ratio =	**1.18**		ratio =	**1.23**
mprime03ac	49	9	2.12	7,172	1,562	0.66	3,343	863	0.11	595	386	0.16	111	94
d=10	185	23		ratio =	**4.59**		ratio =	**3.87**		ratio =	**1.54**		ratio =	**1.18**
mprime03bc	49	9	2.07	7,266	1,573	0.68	3,486	849	0.12	641	396	0.10	111	94
d=10	185	23		ratio =	**4.62**		ratio =	**4.11**		ratio =	**1.62**		ratio =	**1.18**
mprime03cc	49	9	1.47	5,469	1,391	0.45	2,336	721	0.12	534	366	0.10	111	94
d=10	185	23		ratio =	**3.93**		ratio =	**3.24**		ratio =	**1.46**		ratio =	**1.18**

Table 6.2: CPU time in seconds, the trace of AND/OR Branch-and-Bound search (#cm) and the AOMDD size (#aomdd) for the planning instances. Compilation time limit 1 hour.

stronger and they are able to prune the search space significantly. In consequence, the difference in size between the AOMDD and the AND/OR graph explored decreases. When looking at the OR versus the AND/OR compilation schemes, we notice that AOMDD is smaller than the OR MDD, for all reported i-bounds. On some of the harder instances, the OR compilation scheme did not finish within the 1 hour time limit (e.g., `408`, `505`).

ISCAS'89 Benchmark Circuits

Table 6.1 shows the results for experiments with 7 WCSPs derived from the ISCAS'89 circuits described in Chapter 3. The columns are indexed by the mini-bucket i-bound. We observe again that the difference in size between the resulting AOMDD and the AND/OR search graph explored by `AOBB-C+SMB(`i`)` is more prominent for relatively small i-bounds. For example, on the `c432` circuit and at $i = 10$, the AOMDD is about 9 times smaller than the corresponding AND/OR graph.

Planning Instances

We also experimented with problems from planning in temporal and metric domains[1]. These instances were converted into binary WCSPs as follows: each fluent of the planning graph is represented by a variable with domain values representing possible actions to produce this fluent. Hard binary constraints represent mutual exclusions between fluents and actions, and activity edges in the graph. Soft unary constraints represent action costs. The goal is to find a valid plan which minimizes the sum of the action costs.

Table 6.2 shows the results for experiments with 12 planning networks. On this domain we only observe minor differences between the size of the compiled AOMDD and the corresponding AND/OR search graph. This is due to very accurate mini-bucket heuristics which cause the AND/OR Branch-and-Bound to avoid expanding nodes that correspond to solutions whose cost is above the optimal one.

6.5.2 0-1 Integer Linear Programs

We consider the AND/OR Branch-and-Bound algorithm developed in Chapter 5 and denoted by `AOBB-C-ILP`, as the basis for our AND/OR compilation scheme. The heuristic evaluation function used by `AOBB-C-ILP` is computed by solving the linear relaxation of the current subproblem with the *simplex* method [23] (our code used the implementation from the open source library `lp_solve 5.5`[2]).

MIPLIB Instances

MIPLIB is a library of Mixed Integer Linear Programming instances that is commonly used for benchmarking integer programming algorithms. For our purpose we selected four 0-1 ILP instances of increasing difficulty. Table 6.3 reports a summary of the experiment. We observe that the AOMDD is much smaller than the corresponding AND/OR search

[1] http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/BenchmarkS
[2] Available at http://lpsolve.sourceforge.net/5.5/)

miplib	(n, c)	(w*, h)	time	#cm	#aomdd	ratio
p0033	(33, 15)	(19, 21)	0.52	441	164	**2.69**
p0040	(40, 23)	(19, 23)	0.36	129	77	**1.66**
p0201	(201, 133)	(120, 142)	89.44	12,683	5,499	**2.31**
lseu	(89, 28)	(57, 68)	454.79	109,126	21,491	**5.08**

Table 6.3: The trace of AND/OR Branch-and-Bound search (#cm) versus the AOMDD size (#aomdd) for the MIPLIB instances. Compilation time limit 1 hour.

graph, especially for harder problems where the heuristic function is less accurate. For example, on the `lseu` instance, the compiled AOMDD has about 5 times fewer nodes than the AND/OR search graph explored by `AOBB-C-ILP`.

Combinatorial Auctions

Figure 6.6 shows results for experiments with combinatorial auctions drawn from the *regions-upv* and *regions-npv* distribution of the CATS 2.0 test suite [75] (see also Chapter 5 for additional details). The suffixes `npv` and `upv` indicate that the bid prices were drawn from either a normal or uniform distribution. These problem instances simulate the auction of radio spectrum in which a government sells the right to use specific segments of spectrum in different geographical areas. We looked at auctions with 100 goods and increasing number of bids. Each data point represents an average over 10 random instances. For comparison, we also included results obtained with the OR compilation scheme. On this domain, we observe that the compiled AOMDD improves only slightly over the size of the AND/OR search graph. This is because the context minimal AND/OR graph explored is already compact enough due to very accurate heuristic estimates.

MAX-SAT Instances

Table 6.4 shows the results for experiments with 8 `pret` instances (see also Chapter 5 for additional details). These are unsatisfiable instances of graph 2-coloring with parity constraints. The size of these problems is relatively small (60 variables with 160 clauses

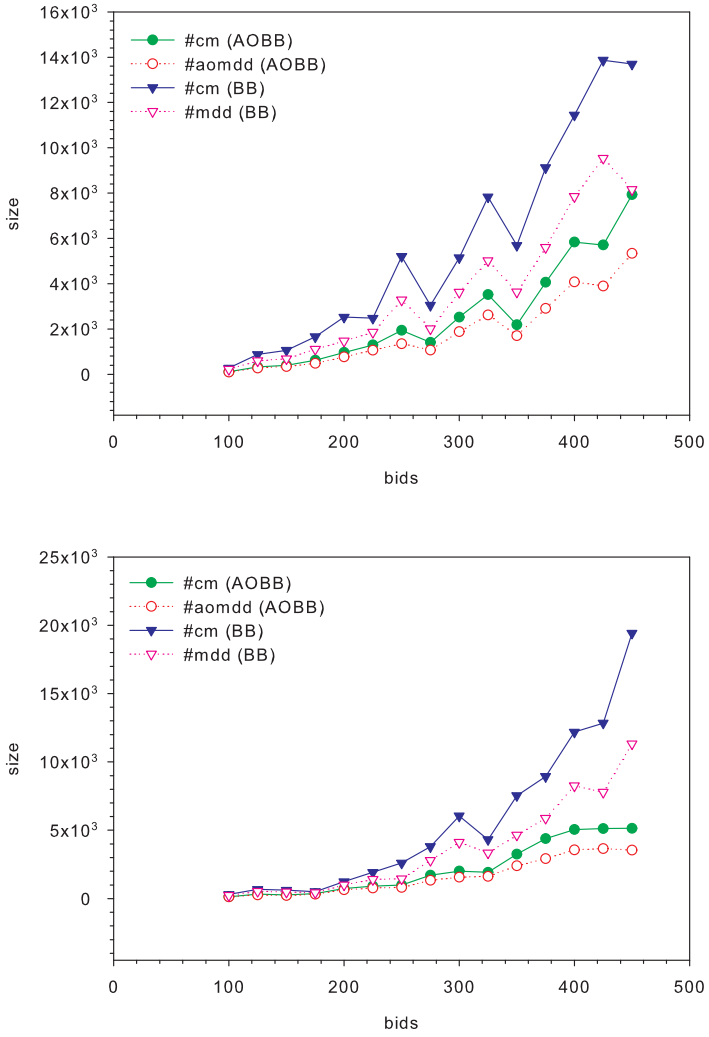


Figure 6.6: The trace of AND/OR Branch-and-Bound search versus the AOMDD size for the *regions-upv* (top) and *regions-npv* (bottom) combinatorial auctions.

pret	(w*, h)	time	#cm	#aomdd	ratio
pret60-25	(6, 13)	2.74	593	255	**2.33**
pret60-40	(6, 13)	3.39	698	256	**2.73**
pret60-60	(6, 13)	3.31	603	222	**2.72**
pret60-75	(6, 13)	2.70	565	253	**2.23**
pret150-25	(6, 15)	18.19	1,544	851	**1.81**
pret150-40	(6, 15)	29.09	2,042	922	**2.21**
pret150-60	(6, 15)	30.09	2,051	877	**2.34**
pret150-75	(6, 15)	29.08	2,033	890	**2.28**

Table 6.4: The trace of AND/OR Branch-and-Bound search (#cm) versus the AOMDD size (#aomdd) for MAX-SAT `pret` instances.

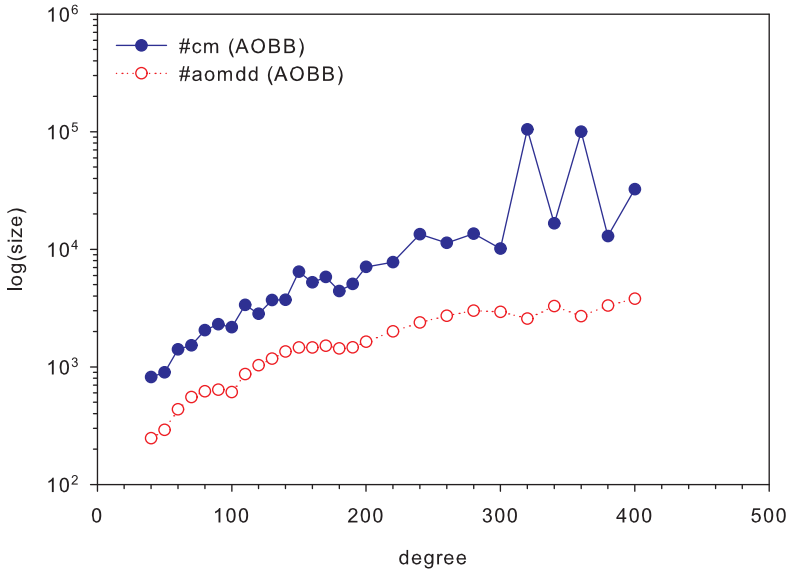


Figure 6.7: The trace of AND/OR Branch-and-Bound search (#cm) versus the AOMDD size (#aomdd) for MAX-SAT `dubois` instances.

for `pret60` and 150 variables with 400 clauses for `pret150`, respectively). However, they have a very small context with size 6 and a shallow pseudo tree with depth 13 and 15, respectively. For this problem class we observe that the AND/OR decision diagrams have about 2 times fewer nodes than the AND/OR search graphs explored by `AOBB-C-ILP`. This is because the respective search spaces are already small enough, and this does not leave much room for additional merging of isomorphic nodes in the diagram.

Figure 6.7 displays the results for experiments with random `dubois` instances with increasing number of variables (see also Chapter 5 for additional details). These are 3-SAT instances with $3 \times degree$ variables and $8 \times degree$ clauses, each of them having 3 literals. As in the previous test case, the `dubois` instances have very small contexts of size 6 and shallow pseudo trees with depths ranging from 10 to 20. The AND/OR decision diagrams compiled for these problem instances are far smaller than the corresponding AND/OR search graphs, especially for some of the larger instances. For example, at degree 320, the corresponding AOMDD is 40 times smaller than the trace of `AOBB-C-ILP`.

6.5.3 Summary of Empirical Results

In summary, the AOMDD offers a very compact representation of the search space explored by an AND/OR Branch-and-Bound algorithm to find the set of optimal solutions to a COP, especially on problem classes for which the heuristic generator produces relatively weak estimates. When the heuristic function is strong, the explored AND/OR search space is far tighter around the set of optimal solutions and does not leave room for additional reductions relative to isomorphic meta-nodes in the decision diagram. On the WCSP domain the size of the compiled AOMDD varies across different levels of the mini-bucket i-bound because we did not prune the non-optimal solutions contained in the AOMDD.

6.6 Conclusion to Chapter 6

We presented a new search based algorithm for compiling the optimal solutions of a constraint optimization problem into a weighted AND/OR Multi-Valued Decision Diagram (AOMDD). Our approach draws its efficiency from: (1) AND/OR search spaces for graphical models [38] that exploit problem structure, yielding memory intensive search algorithms exponential in the problem's *treewidth* rather than *pathwidth*. (2) Heuristic search algorithms exploring tight portions of the AND/OR search space. In particular, we use here a state-of-the-art AND/OR Branch-and-Bound search algorithm [79, 82], with very efficient heuristics (mini-bucket, or simplex-based), that in practice traverses only a small portion of the context minimal graph. (3) Reduction techniques similar to OBDDs further reduce the trace of the search algorithm.

We extended earlier work on AOMDDs [89] by considering weighted AOMDDs based on cost functions, rather than constraints. This can now easily be extended to any weighted graphical model, for example to probabilistic networks. Finally, using an extensive experimental evaluation we show the efficiency and compactness of the weighted AOMDD data-structure.

Chapter 7

Software

All the algorithms described in this dissertation have been implemented in two software packages, called REES and AOLIB, developed in C++ and currently available online, on the web page of the research group of Professor Rina Dechter, at the University of California, Irvine (*http://graphmod.ics.uci.edu/*). This chapter contains a short overview and description of the implementations.

7.1 REES: Reasoning Engine Evaluation Shell

In a typical application, a design is implemented that meets the set of requirements at the time of development. Often, after a program is delivered, the user will want added functionality, or different users will require custom functionality based on their specific needs. In order to accommodate these situations without a complete re-write, or causing a develop/compile/test/ship scenario, a framework that allows for future additions of modules without breaking the existing code base needs to be implemented. A Plug-In architecture will meet these needs.

To put it simply, a system using this architecture would be capable of looking for various Plug-In modules when starting up. Once all the Plug-Ins have been located they are loaded by the main application one by one, or selectively so as to use their built-in features. These Plug-Ins are normally DLLs (Dynamic Linked Library) in disguise and many commercial applications, even the Windows operating system, currently use similar technologies to

allow third-party developers to integrate with their existing application to add functionality or robustness, otherwise missing from the application.

The REES system was purposely designed in this manner. The main reason behind this is that different research groups in the community usually develop their own libraries of algorithms and in most cases they are incompatible with each other, thus making a joint comparison and evaluation practically impossible. REES provides a common interface that promotes reuse of already existing components and allows for comparison and evaluation of alternative technologies, while using a common workbench.

7.1.1 REES Architecture

The architecture of REES system is described in Figure 7.1. Constraint based or probabilistic reasoning problems are locally defined and/or loaded into the main workspace and transferred to the available Plug-Ins for processing. The results produced by the inference algorithms residing in various Plug-Ins are passed back to the REES main workspace for further refining and appropriate display. The existence of a pre-determined interface, implemented by each Plug-In, facilitates easy and complete communication between them and REES. We will now discuss the main components of the proposed architecture: *Workspace, Model, Plug-In modules*.

Workspace and Models

The **Workspace** is the main component of the system. It encapsulates all the problem models defined by the user and available for evaluation, as well as the list of currently loaded Plug-Ins. Using the graphical interface, one has the possibility of defining new problem models, modifying existing ones or selectively loading/unloading Plug-In modules for additional functionality.

A **Model** is an abstract representation of a reasoning problem. Within the framework, such a problem instance may be represented either in parametric form (e.g. we use the well

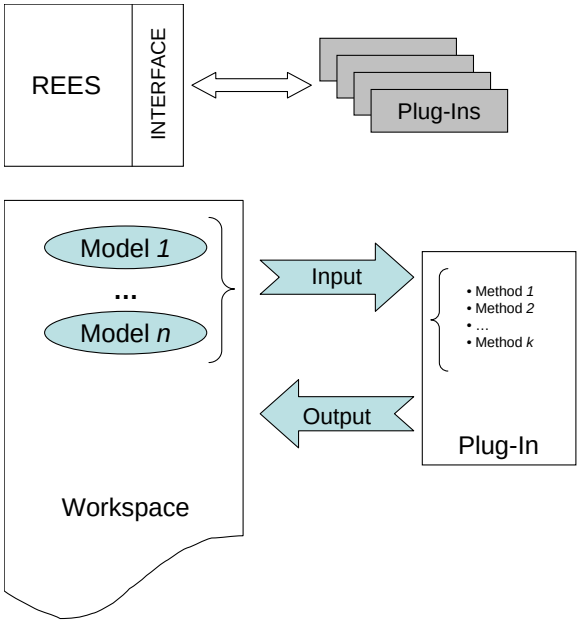


Figure 7.1: REES Plug-In Architecture

known **(N, K, C, T)** parametric model representation) or as a completely defined instance in terms of variables, domain sizes and relationships between variables (i.e. functions). Depending on the chosen model representation (parametric or complete), the graphical interface assists the user in further refining the model. Options such as modifying the values of some parameters (parametric model) or altering the graph structure of the network (complete model) are also available.

Together with the problem structure (i.e. constraint/belief network) a list of processing algorithms must also be defined. These inference algorithms may all reside in a single Plug-In library, but in the common case they may be part of different Plug-Ins. The list of selected algorithms together with their control parameters form the *experiment* associated with the problem model. In this way, reasoning algorithms developed within different research groups can be executed and evaluated altogether on the very same problem instances or benchmarks.

Once a problem model has been completely defined in the current workspace, the common interface takes care of creating an object that is *understood* and can be transferred to

any attached Plug-In. This sub-process is called *random problem generation* and in both cases it creates a complete problem instance. In case of a parametric model representation, the parameters completely define the graph structure and the functions of the problem. In the other case, there is no need for a problem instance generation and the already existing object can be passed along, as is.

Plug-Ins

A **Plug-In** is an external module (a DLL in our framework) that implements some functionality. Once installed, it can be loaded at runtime by the main application (REES) to use the functionality provided using exported functions/classes within the DLL. All the Plug-In modules must conform to a pre-defined interface (see Figure 7.1). The reason for that is determined by the fact that a call to a function residing inside the Plug-In can be issued only after knowing the function name.

As it is defined in this framework, a Plug-In library implements a collection of deterministic and/or non-deterministic reasoning algorithms. A pre-defined header structure ensures the compatibility with the main application (REES). In our implementation, a Plug-In must export the list of implemented algorithms together with their input/output control parameters as well as the list of functions that form the common interface.

7.1.2 A Closer Look

This section describes in more detail the main features of the REES environment and shows the basic steps of the entire process, from model creation to experimentation to viewing and interpreting the results. REES provides an easy to use graphical interface that allows intuitive creation/editing of the problem model, direct adjustment of the control parameters for all algorithms involved in some experiment as well as user friendly display of the results produced by the experiments. REES also provides support for saving either the entire workspace or individual models to a file for later use.

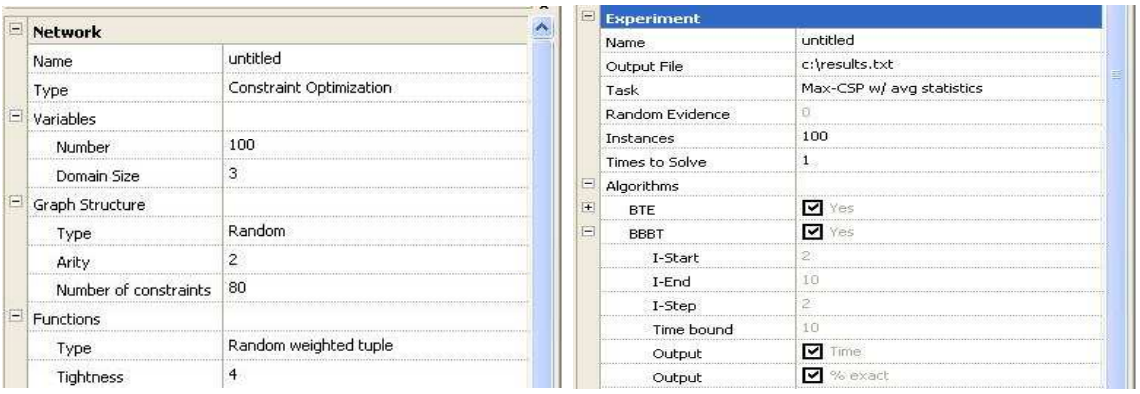


Figure 7.2: REES Graphical Interface. (a) Model. (b) Experiment

Model Definition

The first step in any deterministic/probabilistic reasoning problem is defining the problem model. To create a new model, simply select *Add Model* from the main menu or select the appropriate icon from the toolbar. REES will then begin the process of helping the user to create the model. The underlying network of the model (either a constraint network for deterministic reasoning problems or a belief network for probabilistic reasoning problems) can be specified in two ways:

1. *Parametric Form*. In this case, a parametric model of the problem is created. The user has the options of specifying the size of the network as number of variables and domain size, the type of graph structure to be generated (e.g. *random*, *grid*, *etc.*) as well as the type of functions to be defined over subsets of variables (e.g. *random weighted tuple*). Figure 7.2(a) displays an example of model definition using REES Graphical Interface. At any time, the system allows the user to modify the values of all these parameters. Later, the *random problem generator* will use all these user defined parameters to build a complete model, as described in Section 7.1.1.

2. *Complete Form*. In this case, the entire model, as represented by the graph structure and functions defined over subsets of variables, resides in a text file that will be loaded into the workspace. At this moment, REES system is able to parse several file

formats (e.g. DIMACS), as well as a proprietary format that may contain additional information for a graphical display of the network[1]. REES Net Editor provides easy graphical editing of constraint/belief networks including cut/paste/duplicate nodes and edges. All these options and many others are available from the main menu and toolbar of the application. In this way, the user is offered the possibility of creating its own model either from scratch or modifying an existing one.

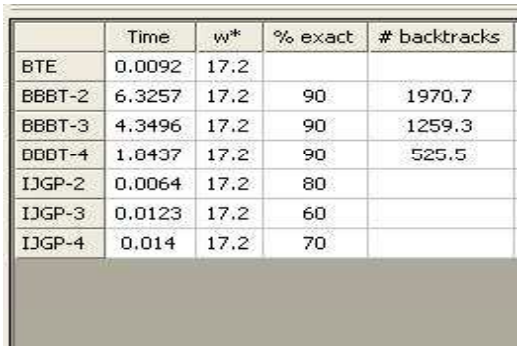
The model definition is completed once an experiment is defined and attached to it. Details on how to do it can be followed in the next section.

Running Experiments

Once a problem model is created, the knowledge it contains can be transferred to the available Plug-Ins, each one of them implementing a set of inference algorithms as described in Section 7.1.1. To create a new experiment, using the current problem model, simply select *New Experiment* from the main menu or select the appropriate icon from the toolbar. A REES wizard will then assist the user in the process of creating the experiment. A typical experiment must specify the *task* it will perform, the number of problem *instances* to be generated as well as the set of *algorithms* together with their control parameters to be executed. Figure 7.2(b) shows an example of an experiment defined on a constraint-based model.

1. *Task*: Depending on the problem model on which the experiment is defined, several tasks may be available (e.g. *Max-CSP, Solution Counting, etc.* for constraint-based models, *Belief, Most Probable Explanation, etc.* for probabilistic models). Each algorithm exported by a Plug-In must have its header information containing the task type it is able to perform.

[1]REES Net Editor is currently available only for belief networks.



	Time	w*	% exact	# backtracks
BTE	0.0092	17.2		
BBBT-2	6.3257	17.2	90	1970.7
BBBT-3	4.3496	17.2	90	1259.3
BBBT-4	1.0437	17.2	90	525.5
IJGP-2	0.0064	17.2	80	
IJGP-3	0.0123	17.2	60	
IJGP-4	0.014	17.2	70	

Figure 7.3: REES Results Display Window.

2. *Instances*: If there is no random behavior specified for this particular model (i.e. *complete model*), then there can only be one instance of the underlying network. If there is either a random structure definition and/or a random function definition (i.e. *parametric model*) then REES can create as many problem instances as indicated by the parameter value.

3. *Algorithms*: Each algorithm exported by some Plug-In library has a set of control parameters associated with. The user must set values for all *input* parameters (if there are any) and may select one or more *output* parameters for visualization. After the execution of the experiment has successfully completed, the average values of the output parameters will be displayed for further analysis.

Once the experiment is created, REES can be instructed to execute it. A detailed log of the execution can also be recorded so as the user to be able to abort the experiment once an error is signaled. The results produced by an experiment that completed successfully are displayed in a spreadsheet, each column representing one of the selected output parameters. This should make comparison between algorithms quite simple and intuitive, where such a comparison is appropriate.

In Figure 7.3 we provide an example of results produced by an *MPE* experiment, that is finding the *Most Probable Explanation* in Bayesian models. The problem was represented as a parametric model that generated 10 random instances of a binary belief network with

100 variables and 90 conditional probability tables. Three algorithms were chosen for evaluation: **BTE** (i.e. *Bucket Tree Elimination*), an exact inference algorithms based on the well known variable elimination mechanism, **BBBT(i)** (i.e. *Branch and Bound with mini-Bucket Tree* heuristics) a complete Branch and Bound search algorithm that uses dynamic heuristics generated by a Mini-Bucket Tree Elimination algorithm to guide the search and **IJGP(i)** (i.e. *Iterative Join Graph Propagation*) an iterative version of graph propagation algorithms. The latter two algorithms are controlled by a parameter called i-bound. For each algorithm, REES displays the average values of the selected output parameters, columnwise. They are: average running time (*Time*), average induced width of the problem (*w**), average accuracy as percent of exactly solved instances (*% exact*) as well as the average number of backtracks for the branch and bound search algorithm (*# backtracks*).

7.2 AND/OR Search for Optimization

The depth-first and best-first AND/OR search algorithms have been implemented from scratch by the author, in a package called AOLIB. The system uses its own input format file (*.simple), but can load any other usual format files (*.erg for Bayesian networks, *.wcsp for Weighted CSPs, as well as *.mps for 0-1 Integer Linear Programs). The package supports the following optimization tasks: finding the Most Probable Explanation of a Bayesian network (AOLIB-MPE), finding the minimal cost solution of a Weighted CSP (AOLIB-WCSP), as well as solving 0-1 Integer Linear Programs (AOLIB-ILP).

The AOLIB-MPE system participated in the UAI'06 (Uncertainty in Artificial Intelligence) Evaluation of Probabilistic Inference Systems, for the MPE task. Results are available at *http://ssli.ee.washington.edu/bilmes/uai06InferenceEvaluation/*. We describe next the components of the AOLIB package.

7.2.1 AOLIB-MPE

AOLIB-MPE contains the implementations of the depth-first AND/OR Branch-and-Bound with caching (`AOBB-C`) as well as the best-first AND/OR (`AOBF-C`) search algorithms for solving the MPE task in Bayesian networks. Both algorithms traverse the context minimal AND/OR search graph associated with the input Bayesian network and use static or dynamic mini-bucket heuristics.

AOLIB-MPE is invoked with three (if no evidence present) or four (if evidence present) arguments, as follows: **aolibMPE networkFile [evidenceFile] parameterFile outputFile**, with the following meaning:

- `<networkFile>` specifies the path to the network specification in Ergo[2] file format.
- `<evidenceFile>` (optional) specifies the path to the evidence specification according to the Ergo file format.
- `<parameterFile>` specifies the path to the file containing custom parameters for the algorithm.
- `<outputFile>` specifies the path to the file to which (the logarithm of) the probability of the most probable explanation is written.

The parameters can be specified within the parameter file using the syntax: **parameter = value**. The following parameters are defined for AOLIB-MPE:

- **h**: (string) the heuristic to use for finding an variable elimination order by which to construct the AND/OR search space. The following values can be used:
  - *minfill*: to indicate the min-fill heuristic

[2] A detailed description of the Ergo (*.erg) file format for Bayesian networks is available online at http://graphmod.ics.uci.edu/group/Ergo_file_format

  - *hypergraph*: to indicate the hypergraph partitioning heuristic

- **a**: (integer) specifies which algorithm to run. The following values can be used:
  - 3: AND/OR Branch-and-Bound with static mini-bucket heuristics
  - 300: AND/OR Branch-and-Bound with static mini-bucket heuristics and constraint propagation via unit resolution
  - 4: AND/OR Branch-and-Bound with dynamic mini-bucket heuristics
  - 400: AND/OR Branch-and-Bound with dynamic mini-bucket heuristics and constraint propagation via unit resolution
  - 9: Best-First AND/OR search with static mini-bucket heuristics
  - 10: Best-First AND/OR search with dynamic mini-bucket heuristics
- **ib**: (integer) specifies the i-bound of the guiding mini-bucket heuristic.
- **cb**: (integer) specifies the cache bound used by AND/OR Branch-and-Bound algorithms.
- **cs**: (string) specifies the caching scheme used by AND/OR Branch-and-Bound algorithms. The following values can be used:
  - *classic*: to indicate the naive caching scheme
  - *adaptive*: to indicate the adaptive caching scheme
- **l** (integer) specifies the time limit in seconds. Default value is -1, which indicates no time limit.

7.2.2 AOLIB-WCSP

AOLIB-WCSP contains the implementations of the depth-first AND/OR Branch-and-Bound as well as the Best-First AND/OR search algorithms for solving WCSPs. In addition to the

mini-bucket heuristics, we also provide an implementation of the AND/OR Branch-and-Bound guided by a form of local soft consistency propagation, called Existential Directional Arc Consistency (EDAC). The algorithm can also accommodate dynamic variable ordering heuristics. The input WCSP instance must be specified in the WCSP[3] file format.

AOLIB-WCSP is invoked with three arguments, as follows: **aolibWCSP networkFile parameterFile outputFile**. The arguments have the same meaning as for AOLIB-MPE. The following parameters are defined for AOLIB-WCSP:

- **h**: (string) the heuristic to use for finding a variable elimination order by which to construct the AND/OR search space:
  - *minfill*: indicates the min-fill heuristic
  - *hypergraph*: indicates the hypergraph partitioning heuristic
- **a**: (integer) specifies which algorithm to run. The following values can be used:
  - 3: AND/OR Branch-and-Bound with static mini-bucket heuristics
  - 4: AND/OR Branch-and-Bound with dynamic mini-bucket heuristics
  - 7: AND/OR Branch-and-Bound with EDAC heuristics
  - 9: Best-First AND/OR search with static mini-bucket heuristics
  - 10: Best-First AND/OR search with dynamic mini-bucket heuristics
- **ib**: (integer) specifies the i-bound of the guiding mini-bucket heuristic.
- **cb**: (integer) specifies the cache bound used by AND/OR Branch-and-Bound algorithms.
- **cs**: (string) specifies the caching scheme used by AND/OR Branch-and-Bound algorithms. The following values can be used:

[3] A detailed description of the WCSP (*.wcsp) file format for Weighted CSPs is available online at http://graphmod.ics.uci.edu/group/WCSP_file_format

  - *classic*: to indicate the naive caching scheme
  - *adaptive*: to indicate the adaptive caching scheme

- **l** (integer) specifies the time limit in seconds. Default value is -1, which indicates no time limit.

- **vo**: (string) specifies the variable ordering used. The following values can be used:
  - *svo*: stands for Static Variable Ordering (algorithms: 3, 4, 7, 9, 10)
  - *pvo*: stands for Partial Variable Ordering (algorithm 7 only)
  - *dvo*: stands for Full Dynamic Variable Ordering (algorithm 7 only)
  - *dso1*: stands for Dynamic Separator Ordering (algorithm 7 only)

7.2.3 AOLIB-ILP

AOLIB-ILP contains the implementations of the depth-first AND/OR Branch-and-Bound as well as the Best-First AND/OR search algorithms for solving 0-1 Integer Linear Programs. The input 0-1 ILP instance must be specified in the MPS[4] file format. AOLIB-ILP is based on the open-source `lp_solve` library (see also Chapter 5 for more details).

The following parameters are defined for AOLIB-ILP:

- **h**: (string) the heuristic to use for finding a variable elimination order by which to construct the AND/OR search space:
  - *minfill*: indicates the min-fill heuristic
  - *hypergraph*: indicates the hypergraph partitioning heuristic

- **a**: (integer) specifies which algorithm to run. The following values can be used:
  - 1: OR Branch-and-Bound search

[4] A detailed description of the MPS (*.mps) file format for integer programs is available online at http://graphmod.ics.uci.edu/group/MPS_file_format

  - 2: AND/OR Branch-and-Bound search without caching (i.e., tree search)
  - 3: AND/OR Branch-and-Bound search with caching (i.e., graph search)
  - 4: Best-First AND/OR tree search
  - 5: Best-First AND/OR graph search

- **l** (integer) specifies the time limit in seconds. Default value is -1, which indicates no time limit.

- **vo**: (string) specifies the variable ordering used. The following values can be used:
  - *svo*: stands for Static Variable Ordering (algorithms: 2, 3, 4, 5)
  - *pvo*: stands for Partial Variable Ordering (algorithms: 2, 4)

Chapter 8

Conclusion

The research presented in this dissertation is focused on the application of the AND/OR search spaces perspective to solving general constraint optimization tasks over graphical models. In contrast to the traditional OR search, the new AND/OR search is sensitive to problem decomposition, resulting often in significantly reduced computational costs. In conjunction with the AND/OR search space, we investigated extensively a class of partition-based heuristic functions, based on the Mini-Bucket approximation.

We introduced a general Branch-and-Bound algorithm that traverses an AND/OR search tree in a depth-first manner and explored the impact of various dynamic variable ordering heuristics. We also investigated memory intensive search algorithms that traverse an AND/OR search graph using both depth-first and best-first control schemes. Subsequently, we extended the general principles of solving optimization problems using AND/OR search with context-based caching to the class of 0-1 Integer Linear Programs. Our extensive empirical evaluation demonstrated conclusively that the new AND/OR search algorithms improved dramatically over the traditional OR competitive approaches, in many cases by several orders of magnitude.

We also applied the AND/OR search perspective to decision diagrams. We introduced a new search-based algorithm for compiling AND/OR Multi-Valued Decision Diagrams (AOMDDs), as representations of the optimal solutions to an optimization problem. Using an extensive experimental evaluation we showed the efficiency and compactness of the weighted AOMDD data-structure compared with the initial trace of the search algorithm.

Finally, we explored empirically the power of two systematic Branch-and-Bound search algorithms that traverse the traditional OR search space and exploit the mini-bucket based heuristics in both static and dynamic settings. We compared them against a number of popular stochastic local search algorithms, as well as against a class of iterative belief propagation algorithms. We showed that, when viewed as approximation schemes, the Branch-and-Bound algorithms were overall superior to the local search algorithms, except when the domain size was small, in which case they were competitive.

Directions of Future Research

The AND/OR search perspective for optimization leaves room for additional improvements that can be pursued in the future.

AND/OR Branch-and-Bound. Our current approach for handling the deterministic information present in the graphical model within the AND/OR Branch-and-Bound framework is based on a restricted form of relational arc consistency, namely unit resolution. Therefore, it would be interesting to exploit more powerful constraint propagation schemes such as generalized arc or path consistency. Recent improvements of the Mini-Bucket algorithm ($e.g.$, *Depth-First Mini-Bucket Elimination* [109]) could also be explored further in the context of AND/OR search. Finally, we plan to extend our memory intensive algorithms to dynamic variable orderings.

Best-First AND/OR Search. Our best-first AND/OR search algorithm, `AOBF-C`, can also be improved. First, rather than recompute a new estimated best partial solution tree after every node expansion, it is possible instead to expand one or more leaf nodes and some number of their descendants all at once, and then recompute an estimated best partial solution tree. This strategy reduces the computational overhead of frequent bottom-up operations but incurs the risk that some node expansions may not be on the best solution

tree.

The space required by AOBF-C can be enormous, due to the fact that all nodes generated by the algorithm have to be saved prior to termination. Therefore, a memory bounding strategy may also be used for context minimal AND/OR graphs, as previously suggested in [97, 103, 16]. To employ it, the algorithm periodically reclaims needed storage space by discarding some portions of the explicated AND/OR search graph. For example, it is possible to determine a few of those partial solution trees within the entire search graph having the *largest* estimated costs. These can be discarded periodically, with the risk of discarding one that might turn out to be the top of an optimal solution tree.

AND/OR Search for 0-1 ILP. Our depth-first and best-first AND/OR search approach for 0-1 ILP leaves room for future improvements, which are likely to make it more efficient in practice. For instance, it can be modified to incorporate *cutting planes* to tighten the linear relaxation of the current subproblem. We can also incorporate good initial upper bound techniques (using incomplete schemes), which in some cases can allow a best-first performance using depth-first AND/OR Branch-and-Bound algorithms. Finally, we plan to accelerate our solvers as well as to incorporate other dynamic variable ordering heuristics ($e.g.$, strong branching) in order to improve our results.

Multi-Objective Optimization. Multi-objective constraint optimization is the process of simultaneously optimizing two or more conflicting objectives subject to certain constraints. Maximizing profit and minimizing the cost of a product, maximizing performance and minimizing fuel consumption of a vehicle and minimizing weight while maximizing the strength of a particular component are examples of multi-objective optimization problems. Advances in exact methods for multi-objective optimization are critical in many real world applications. Therefore, we could extend our recent results on AND/OR search for single objective optimization to the multi-objective case. These new algorithms would exploit efficiently the problem structure during search and use caching of partial results effectively.

Bibliography

[1] D. Allen and A. Darwiche. New advances in inference using recursive conditioning. In *Uncertainty in Artificial Intelligence (UAI-2003)*, pages 2–10, 2003.

[2] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Pbs: A backtrack search pseudo-boolean solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, 2002.

[3] S. A. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, 25:2–23, 1985.

[4] F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination: Bayesian inference via backtracking search. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI'03)*, pages 20–28, 2003.

[5] R. Bayardo and D. Miranker. A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In *National Conference on Artificial Intelligence (AAAI)*, pages 298–304, 1996.

[6] R. Bayardo and J. D. Pehoushek. Counting models using connected components. In *National Conference of Artificial Intelligence (AAAI-2000)*, pages 157–162, 2000.

[7] E. Bensana, M. Lemaitre, and G. Verfaillie. Earth observation satellite management. *Constraints*, 4(3):293–299, 1999.

[8] C. Bessiere and J.-C. Regin. Mac and combined heuristics: two reasons to forsake fc (and cbj) on hard problems. In *Principles and Practice of Constraint Programming (CP-1996)*, pages 61–75, 1996.

[9] S. Bistarelli, U. Montanari, and F. Rossi. Semiring based constraint solving and optimization. *Journal of ACM*, 44(2):309–315, 1997.

[10] H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In *The Twenty Second International Symposium on Mathematical Foundations of Computer Science (MFCS'97)*, pages 19–36, 1997.

[11] H. L. Bodlaender and J. R. Gilbert. Approximating treewidth, pathwidth and minimum elimination tree-height. Technical report, Utrecht University, 1991.

[12] D. Brelaz. New method to color the vertices of a graph. *Communications of the ACM*, 4(22):251–256, 1979.

[13] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35:677–691, 1986.

[14] F. B. C. Lecoutre and F. Hemery. Backjump-based techniques versus conflict directed heuristics. In *Proceedings of ICTAI-2004*, pages 549–557, 2004.

[15] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. Warners. Radio link frequency assignment. *Constraints*, 4:79–89, 1999.

[16] P. Chakrabati, S. Ghose, A. Acharya, and S. de Sarkar. Heuristic search in restricted memory. *In Artificial Intelligence*, 3(41):197–221, 1989.

[17] M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational bayesian networks for exact inference. *International Journal of Approximate Reasoning*, 42(1–2):4–20, 2006.

[18] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[19] Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of the Twelfth International Conference of Artificial Intelligence (IJCAI'91)*, pages 318–324, 1991.

[20] M. C. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2):199–227, 2004.

[21] V. C. D. Applegate, R. Bixby and W. Cook. Finding cuts in the tsp (a preliminary report). In *Technical Report 95-05, DIMACS, Rutgers University*, 1995.

[22] P. Dagum and M. Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. In *National Conference on Artificial Intelligence (AAAI-93)*, 1993.

[23] G. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity Analysis of Production and Allocation*, 1951.

[24] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 125(1-2):5–41, 2001.

[25] S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted csps. In *International Joint Conference in Artificial Intelligence (IJCAI-2005)*, 2005.

[26] S. de Givry, J. Larrosa, and T. Schiex. Solving max-sat as weighted csp. *In CP*, 2003.

[27] S. de Givry, I. Palhiere, Z. Vitezica, and T. Schiex. Mendelian error detection in complex pedigree using weighted constraint satisfaction techniques. In *ICLP Workshop on Constraint Based Methods for Bioinformatics*, 2005.

[28] S. de Givry, T. Schiex, and G. Verfaillie. Exploiting tree decomposition and soft local consistency in weighted csp. In *National Conference on Artificial Intelligence (AAAI-2006)*, 2006.

[29] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[30] R. Dechter. Mini-buckets: A general scheme of generating approximations in automated reasoning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1297–1302, 1997.

[31] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.

[32] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113/1-2:41–85, 1999.

[33] R. Dechter. A new perspective on algorithms for optimizing policies under uncertainty. In *International Conference on Artificial Intelligence Planning Systems (AIPS-2000)*, pages 72–81, 2000.

[34] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[35] R. Dechter. And/or search spaces for graphical models. *Submitted*, 2004.

[36] R. Dechter, K. Kask, and J. Larrosa. A general scheme for multiple lower bound computation in constraint optimization. *Principles and Practice of Constraint Programming (CP2000)*, 2001.

[37] R. Dechter and D. Larkin. Hybrid processing of beliefs and constraints. In *Uncertainty in Artificial Intelligence (UAI-2001)*, pages 112–119, 2001.

[38] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2-3):73–106, 2007.

[39] R. Dechter, R. Mateescu, and K. Kask. Iterative join-graph propagation. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence (UAI'02)*, pages 128–136, 2002.

[40] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of a*. *In Journal of ACM*, 32(3):505–536, 1985.

[41] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[42] R. Dechter and I. Rish. Mini-buckets: A general scheme of approximating inference. *Journal of ACM (JACM)*, 2003.

[43] H. Dixon and M. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *National Conference on Artificial Intelligence (AAAI-2002)*, pages 635–640, 2006.

[44] C. L. F. Boussemart, F. Hemery and L. Sais. Boosting systematic search by weighting constraints. In *European Conference on Artificial Intelligence (ECAI-2004)*, pages 146–150, 2004.

[45] H. Fargier and M. Vilarem. Compiling csps into tree-driven automata for interactive solving. *Constraints*, 9(4):263–287, 2004.

[46] M. Fishelson, N. Dovgolevsky, and D. Geiger. Maximum likelihood haplotyping for general pedigrees. *Human Heredity*, 2005.

[47] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 18(1):189–198, 2002.

[48] E. C. Freuder and M. J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1076–1078, 1985.

[49] E. C. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70, 1992.

[50] N. L. G. Gottlob and F. Scarcello. A comparison of structural csp decomposition methods. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 394–399, 1999.

[51] M. L. G. Verfaillie and T. Schiex. Russian doll search for solving constraint optimization problems. In *National Conference on Artificial Intelligence (AAAI)*, 1996.

[52] T. Hadzic and H. R. Andersen. A bdd-based polytime algorithm for cost-bounded interactive configuration. In *National Conference on Artificial Intelligence (AAAI)*, 2006.

[53] T. Hadzic and J. Hooker. Postoptimality analysis for integer programming using binary decision diagrams. *Technical Report, Carnegie Mellon.*, 2006.

[54] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[55] R. A. Howard and J. E. Matheson. *Influence diagrams. The principles and applications of Decision analyis*. Strategic decisions Group, Menlo Park, CA, USA, 1984.

[56] J. Huang and A. Darwiche. A structure-based variable ordering heuristic. In *International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 1167–1172, 2003.

[57] F. Hutter, H. Hoos, and T. Stutzle. Efficient stochastic local search for mpe solving. In *In International Joint Conference on Artificial Intelligence (IJCAI-2005)*, pages 169–174, 2005.

[58] P. M. J. Larrosa and T. Schiex. Maintaining reversible dac for max-csp. *Artificial Intellignce*, pages 149–163, 1999.

[59] P. Jegou and C. Terrioux. Decomposition and good recording for solving max-csps. In *European Conference on Artificial Intelligence (ECAI 2004)*, pages 196–200, 2004.

[60] F. Jensen, S. Lauritzen, and K. Olesen. Bayesian updating in causal probabilistic networks by local computation. *Computational Statistics Quarterly*, 4:269–282, 1990.

[61] S. Joy, J. Mitchell, and B. Borchers. A branch and cut algorithm for max-sat and weighted max-sat. *In Satisfiability Problem: Theory and Applications*, pages 519–536, 1997.

[62] L. Kanal and V. Kumar. *Search in artificial intelligence.* Springer-Verlag., 1988.

[63] R. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations, Plenum Press, NY*, pages 85–103, 1972.

[64] K. Kask and R. Dechter. Stochastic local search for bayesian networks. In *Workshop on AI and Statistics*, pages 113–122, 1999.

[65] K. Kask and R. Dechter. A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence*, 129(1-2):91–131, 2001.

[66] K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying cluster-tree decompositions for reasoning in graphical models. *Artificial Intelligence*, 166 (1-2):165–193, 2005.

[67] U. Kjæaerulff. Triangulation of graph-based algorithms giving small total space. *Technical Report, University of Aalborg, Denmark*, 1990.

[68] F. R. Kschischang and B. H. Frey. Iterative decoding of compound codes by probability propagation in graphical models. *In IEEE Journal of Selected Areas in Communication*, 16(2):219–230, 1998.

[69] D. Larkin and R. Dechter. Bayesian inference in the presence of determinism. In *The Ninth International Workshop on Artificial Intelligence and Statistics, AISTATS'03*, 2003.

[70] J. Larrosa, P. Meseguer, and M. Sanchez. Pseudo-tree search with soft constraints. In *In European Conference on Artificial Intelligence (ECAI-2002)*, pages 131–135, 2002.

[71] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted csp. In *International Joint Conference in Artificial Intelligence (IJCAI-2003)*, pages 631–637, 2003.

[72] J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159(1–2):1–26, 2004.

[73] S. Lauritzen and D. Spiegelhalter. Local computation with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, 1988.

[74] E. Lawler and D. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[75] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *ACM Electronic Commerce*, pages 66–76, 2000.

[76] W. Li and P. van Beek. Guiding real-world sat solving with dynamic hypergraph separator decomposition. In *International Conference on Tools with Artificial Intelligence (ICTAI'04)*, pages 542–548, 2004.

[77] Z. Li and B. DAmbrosio. An efficient approach for finding the mpe in belief networks. In *Uncertainty in Artificial Intelligence (UAI-1993)*, pages 342–349, 1993.

[78] D. MacKay and R. Neal. Near shannon limit performance of low density parity check codes. *In Electronic Letters*, 33(1):457–458, 1996.

[79] R. Marinescu and R. Dechter. And/or branch-and-bound for graphical models. In *International Joint Conference on Artificial Intelligence (IJCAI-2005)*, pages 224–229, 2005.

[80] R. Marinescu and R. Dechter. And/or branch-and-bound search for pure 0/1 integer linear programming problems. In *International Conference on Integration of AI and OR techniques for Combinatorial Optimization (CPAIOR)*, pages 152–166, 2006.

[81] R. Marinescu and R. Dechter. Dynamic orderings for and/or branch-and-bound search in graphical models. *In European Conference on Artificial Intelligence (ECAI-2006)*, pages 138–142, 2006.

[82] R. Marinescu and R. Dechter. Memory intensive branch-and-bound search for graphical models. *In National Conference on Artificial Intelligence (AAAI-2006)*, 2006.

[83] R. Marinescu and R. Dechter. Best-first and/or search for 0/1 integer programming. In *International Conference on Integration of AI and OR techniques for Combinatorial Optimization (CPAIOR-2007)*, 2007.

[84] R. Marinescu and R. Dechter. Best-first and/or search for graphical models. In *National Conference on Artificial Intelligence (AAAI-2007)*, pages 1171–1176, 2007.

[85] R. Marinescu and R. Dechter. Best-first and/or search for most probable explanations. In *Uncertainty in Artificial Intelligence (UAI-2007)*, 2007.

[86] R. Marinescu, K. Kask, and R. Dechter. Systematic vs non-systematic algorithms for solving the mpe task. In *Uncertainty in Artificial Intelligence (UAI-2003)*, pages 394–402, 2003.

[87] A. Martelli and U. Montanari. Additive and/or graphs. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1–11, 1973.

[88] R. Mateescu and R. Dechter. AND/OR cutset conditioning. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 230–235, 2005.

[89] R. Mateescu and R. Dechter. Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs). In *Principles and Practice of Constraint Programming (CP)*, pages 329–343, 2006.

[90] R. Mateescu, R. Dechter, and K. Kask. Tree approximation for belief updating. In *Proceedings of The Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 553–559, 2002.

[91] R. McEliece, D. MacKay, and J. Cheng. Turbo decoding as an instance of pearls belief propagation algorithm. *In IEEE Journal of Selected Areas in Communication*, 16(2):140–152, 1998.

[92] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[93] P. Mills and E. Tsang. Guided local search for solving sat and weighted max-sat problems. *Journal of Automated Reasoning (JAR)*, 2000.

[94] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. *Design and Automation Conference*, 2001.

[95] G. Nemhauser and L. Wolsey. *Integer and combinatorial optimization.* Wiley, 1988.

[96] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA, 1980.

[97] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga, 1980.

[98] J. Ott. *Analysis of Human Genetic Linkage*. The Johns Hopkins University Press, 1999.

[99] N. N. P. Hart and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 8(2):100–107, 1968.

[100] M. Padberg and G. Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch-and-cut. *Operations Research Letters*, 6:1–7, 1987.

[101] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[102] J. Park. Using weighted max-sat engines to solve mpe. In *National Conference on Artificial Intelligence (AAAI)*, 2002.

[103] J. Pearl. Heuristics: Intelligent search strategies. In *Addison-Wesley*, 1984.

[104] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[105] Y. Peng and J. Reggia. A connectionist model for diagnostic problem solving. *IEEE Transactions on Systems, Man and Cybernetics*, 1989.

[106] J. R. Bayardo and D. P. Miranker. On the space-time trade-off in solving constraint satisfaction problems. In *Fourteenth International Joint Conference on Artificial Intelligence(I95)*, pages 558–562, 1995.

[107] R. M. R. Mateescu and R. Dechter. And/or multi-valued decision diagrams for constraint optimization. In *International Conference on Principles and Practice of Constraint Programming (CP-2007)*, 2007.

[108] I. Rish and R. Dechter. Resolution vs. search: two strategies for sat. *Journal of Automated Reasoning*, 24(1-2):225–275, 2000.

[109] E. Rollon and J. Larrosa. Depth-first mini-bucket elimination. In *Principles and Practice of Constraint Programming (CP)*, pages 563–577, 2005.

[110] A. P. S. Minton, M.D. Johnston and P. Laired. Solving large scale constraint satisfaction and scheduling problems using heuristic repair methods. In *National Conference on Artificial Intelligence (AAAI)*, pages 17–24, Anaheim, CA, 1990.

[111] T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *In International Joint Conference on Artificial Intelligence (IJCAI-1999)*, pages 542–547, 1999.

[112] T. Sang, P. Beame, and H. Kautz. Solving Bayesian networks by weighted model counting. In *National Conference of Artificial Intelligence (AAAI-2005)*, pages 475–482, 2005.

[113] E. Santos. On the generation of alternative explanations with implications for belief revision. In *Uncertainty in Artificial Intelligence (UAI-1991)*, pages 339–347, 1991.

[114] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *National Conference on Artificial Intelligence*, pages 440–446, 1992.

[115] P. Shenoy and G. Shafer. Propagating belief functions with local computations. *In IEEE Expert*, 1(4):43–52, 1986.

[116] S. Shimony and E. Charniak. A new algorithm for finding map assignments to belief networks. In *Uncertainty in Artificial Intelligence (UAI-1991)*, pages 185–193, 1991.

[117] B. Smith. Phase transition and the mushy region in constraint satisfaction. In *In European Conference on Artificial Intelligence (ECAI-1994)*, pages 100–104, 1994.

[118] B. K. Sy. Reasoning mpe to multiply connected belief networks using message-passing. In *National Conference of Artificial Intelligence (AAAI-1992)*, pages 570–576, 1992.

[119] P. B. T. Sang and H. Kautz. A dynamic approach to mpe and weighted max-sat. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-2007)*, pages 549–557, 2007.

[120] P. Thbault, S. de Givry, T. Schiex, and C. Gaspin. Combining constraint processing and pattern matching to describe and locate structured motifs in genomic sequences. In *Fifth IJCAI-05 Workshop on Modelling and Solving Problems with Constraints*, 2005.

[121] C. Voudouris. Guided local search for combinatorial optimization problems. Technical report, PhD Thesis. University of Essex, 1997.

[122] B. Wah and Y. Shang. Discrete lagrangian-based search for solving max-sat problems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 378–383, 1997.

[123] R. Wallace. Analysis of heuristic methods for partial constraint satisfaction problems. In *In Principles and Practice of Constraint Programming (CP-1996)*, pages 482–496, 1996.

[124] J. Walser. *Integer Optimization Local Search*. Springer, 1999.

[125] L. A. Wolsey. *Integer Programming*. Wiley, 1998.

[126] Z. Xing and W. Zhang. Efficient strategies for (weighted) maximum satisfiability. In *Principles and Practice of Constraint Programming (CP-2004)*, pages 660–705, 2004.