Abstract of "Efficient Search Procedures for Solving Combinatorial Problems" by Serdar Kadıoğlu, Ph.D., Brown University, May, 2012.

Solving combinatorial problems is an interplay between search and inference. In this thesis, we focus on search and investigate its important aspects. We start with complete search procedures and consider binary search, which is frequently used to augment a feasibility solver to handle optimization problems. In this setting, we often observe that negative trials (i.e., showing that a certain solution quality cannot be achieved) are significantly harder than positive trials. We consider a simple cost model where negative trials cost a constant factor more than positive trials and show how binary search can be biased optimally to achieve optimal worst-case and average-case performance.

Next, as a complementary approach, we turn to incomplete search procedures. We propose Hegel and Fichte's dialectic as a local search meta-heuristic. Dialectic is an appealing mental concept for local search as it allows developing functions for search space exploration and exploitation independently. We illustrate dialectic search, its simplicity and great efficiency on problems from highly different problem domains.

We then study variable and value selection heuristics, and propose a simple modification to impact-based search strategy. We present computational results on constraint satisfaction problems that show improvements in the search performance.

Finally, we look at the interaction between search and inference. In particular, we investigate incrementality during tree search interleaved with constraint propagation. We first consider constraints based on context-free grammars for which we devise a time-and space-efficient filtering algorithm. We then look at constraints that enforce the same-relation for every pair of variables in binary constraint satisfaction problems. We show that achieving generalized arc-consistency in special graphs such as cliques, complete bipartite, and directed acyclic graphs is NP-hard. However, we can leverage the knowledge that sets of pairs of variables all share the same relation for both theoretical and practical gains.

Efficient Search Procedures for Solving Combinatorial Problems

by
Serdar Kadıoğlu
B. S., Sabanci University, 2007
Sc. M., Brown University, 2009

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May, 2012

© Copyright 2012 by Serdar Kadıoğlu

This dissertation by Serdar Kadıoğlu is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ________________ ____
Meinolf Sellmann, Director

Recommended to the Graduate Council

Date ________________ ____
Sorin Istrail, Reader

Date ________________ ____
Pierre Flener, Reader
(Uppsala University)

Approved by the Graduate Council

Date ________________ ____
Peter M. Weber
Dean of the Graduate School

Vitæ

Serdar Kadıoğlu was born in Kars, Turkey in 1985. He received his undegraduate degree in Computer Science at Sabanci University, Turkey in 2007. He was also an Erasmus exchange student in Uppsala University, Sweden in 2006. He was an intern at ASTRA Research group in Uppsala, Sweden in the summer of 2006. He started the doctoral program in the Computer Science Department at Brown University, USA in 2007. He received a master's degree in Computer Science from Brown University in 2009. In spring 2010, he went back to Uppsala University, this time as a visiting student researcher. He had a summer internship at Adobe Systems Incorporated in 2010. He was a Paris Kanellakis fellow in summer 2011. Serdar's research interests are in constraint programming, combinatorial optimization, operations research and meta-heuristics. His thesis investigates complete and incomplete search procedures for solving combinatorial problems and was awarded the 2010 IBM Ph.D. Scholarship. In the beginning of fall 2011, Serdar will be joining the Advanced Constraint Technology team at Oracle Corporation as a research and development scientist.

Acknowledgements

I had four wonderful years at Brown University as a Ph.D. student, and I would like to thank everyone who contributed to this amazing experience.

It all started back when I took a course on Constraint Programming (CP) taught by Pierre Flener at Uppsala University in Sweden. After this engaging introductory course, I was further fortunate that Pierre Flener gave me a chance to work with him as a summer intern. We continued working together during his sabbatical year at Sabanci University. Overall, it was one and a half year of exposure to fundamental topics in CP. I learned immensely from working directly with him. My words would never be enough to express my gratitude for the opportunities he provided me with —her sey icin cok tesekkurler Hocam!

It was then Meinolf Sellmann who agreed to be my Ph.D. advisor. He offered absolutely everything that a Ph.D. student could ever ask for. His always open door and our frequent meetings in my first year helped me to get into research quickly. He guided me through challenging problems. I will never forget how exciting our meetings were, that I had to take a good five minutes back in my office afterward to get back to breathing normal. He made it possible to experience both sides; taking part in a research project, and leading one. He let me to explore my own directions while always being there with his encouragement. I learned enormously from him, his style, and his work ethic. His continuous support to engage me in the research community was invaluable to meeting other researchers in the field. Outside work, I enjoyed our hiking trips and rides together, dinners and many Halloweens, especially the one in Salem, the witch town. I would like to wholeheartedly thank him —Meinolf, without you, I would never be where I am right now.

I am also grateful to my advisor, Meinolf Sellmann, and to my hosts Pierre Flener and Justin Pearson, for the time I spent as a visiting student researcher at Uppsala University.

I received a lot of support within our Department. I would especially like to thank Shriram Krishnamurthi. I enjoyed our discussions, and benefited a lot from his advices. Sorin Istrail was always encouraging. I also thank Sorin Istrail and Anna Lysyanskaya for being on my thesis committee. I took extraordinary courses at Brown. I especially enjoyed the Combinatorial Optimization courses taught by Meinolf Sellmann and Pascal Van Hentenryck and the Approximation Algorithms course given by Claire Mathieu.

I would like to thank Shriram Krishnamurthi and Ugur Cetintemel for supporting my application to IBM Ph.D. Fellowship. I also thank Andrew Davenport from IBM Research for his support.

I would like to thank my co-authors; Yuri Malitsky, Kevin Tierney, Christopher Jefferson, Karen E. Petrie, Standa Živný, Ashish Sabharwal, Horst Samulowitz, Eoin O'Mahony, Philippe Refalo, and Siddhartha Jain. It was a pleasure to work with you, and I hope to continue our collaborations in future.

The research performed in this thesis was supported by National Science Foundation through the Career–0644113: Cornflower Project, IBM Ph.D. Scholarship and Paris Kanellakis fellowship. It was an honor for me as a Turkish student to be a Paris Kanellakis fellow. I am grateful to the Association of Constraint Programming and to Roberto Tamassia for supporting my participation in the CP Summer School in 2011.

Karen and Alen Usas, Olga Sellmann, and Esra Bayoğlu Flener made foreign lands feel like home!

A big thank you to all of my friends in CIT. The Optimization Gang was a unique group at the time; Yuri Malitsky, Justin Yip, Olya Ohrimenko, Kevin Tierney, Carleton Coffrin. It is sad to see that the Optimization team at Brown has dissolved after everyone has moved on to take different positions. I hope to continue to work together, however. Yuri, you have been a great team member! It was great fun to work with you, even under most extreme conditions —you know what I mean. Other Turkish students helped me with their experience; Alptekin Küpçü, Aysun Basçetinçelik, Çağatay Demiralp and Mert Akdere —sağolun gençler! Joseph Faustino Ramos has been a great roommate during these years. His company made Providence a much more fun place —sağol adamım! Our departmental soccer team was a great source of fun too. I truly enjoyed playing for Bytesoccer. I hope we keep this tradition and have a team for many upcoming years with great success! Lastly I must say that life in CIT would not have been as easy if it had not been for Lauren, Dawn, Jane, Genie, Stephen and Amy!

My professors at Sabanci University provided support even after my graduation. I would like to thank Albert Levi, Berrin Yanıkoğlu, Esra Erdem, Hüsnü Yenigün and Uğur Sezerman. I would also like to thank İlker Birbil and Kerem Bülbül for giving me a chance to participate in Operations Research projects. Thanks also to my friends from college; Can Çecen, Ersoy Bayramoğlu, and Serdar Çakıcı.

A thank you deep from my heart to the special one; HCG, for being with me in my best, and even more so, in my worst times. I will always remember the things we have done together!

Back in Turkey, I would like to thank Hasan Hüseyin Balcı for introducing me to science in a broad sense. I also received endless support from my relatives. Many thanks to my uncles Namık Kadıoğlu, Tuncer and Özer Kalem, and my aunts Filiz Kalem and Nesrin Kadıoğlu. Also, my grandmothers; Billur Kalem and Nebahat Kadıoğlu —ellerinize sağlık for your delicious food!

Finally, my mother Gülten Kadıoğlu and my father Necdet Kadıoğlu: Thank you for your everlasting love and care! This thesis is dedicated to you. I have done the least I could do with everything you have given to me. And my sister, Sibel, I love you beyond words.

Contents

List of Tables **xii**

List of Figures **xv**

1 Introduction **2**

1.1 Solving Optimization and Satisfaction Problems 3
1.2 Motivation 3
1.3 Background 4
1.4 Contributions and Outline 5
1.4.1 Part I – Complete search 5
1.4.2 Part II – Incomplete search 5
1.4.3 Part III – Variable and value selection 6
1.4.4 Part IV – Interplay between search and inference 6
1.5 Related Publications 7

2 Background **10**

2.1 Perturbative vs. Constructive Search 11
2.2 Systematic vs. Local Search 12
2.3 Systematic Search Algorithms 13
2.3.1 British Museum Procedure 13
2.3.2 Depth-First Search (DFS) 14
2.3.3 Breadth-First Search (BFS) 14
2.3.4 Uniform Cost Search 14
2.3.5 Best-First Search 14

2.3.6 A and A* Search . . . 15
2.3.7 Iterative Deepening Search (IDS) . . . 15
2.4 Local Search Meta-Heuristics . . . 15
2.4.1 Simulated Annealing (SA) . . . 16
2.4.2 Tabu Search (TS) . . . 17
2.4.3 Greedy Randomized Adaptive Search Procedure (GRASP) . . . 18
2.4.4 Evolutionary Algorithms . . . 18
2.4.5 Ant Colony Optimization . . . 19

I Complete Search 20

3 Dichotomic Search Protocols for Constrained Optimization 21
3.1 Introduction . . . 22
3.2 Skewed Binary Search . . . 24
3.3 Skewed Dichotomic Search for Constrained Optimization . . . 30
3.3.1 The Streeter-Smith Strategy . . . 30
3.3.2 Parameter Tuning based on Skewed Binary Search Protocols . . . 32
3.4 Numerical Results . . . 34
3.4.1 Experiments using IBM Ilog CP Optimizer . . . 37
3.5 Conclusion . . . 38

II Incomplete Search 39

4 Dialectic Search 40
4.1 Introduction . . . 41
4.2 Dialectic Search . . . 42
4.2.1 A Meta-Heuristic Inspired by Philosophy . . . 43
4.2.2 Dialectic Search . . . 43
4.3 Constraint Satisfaction . . . 46
4.3.1 Costas Arrays . . . 47
4.4 Continuous Optimization . . . 49
4.5 Constrained Optimization – Set Covering . . . 50
4.6 Boosting the Performance . . . 54
4.6.1 Instance Specific Algorithm Configuration . . . 54
4.7 Conclusion . . . 56

III Variable and Value Selection **58**

5 Incorporating Variance in Impact-Based Search **59**

5.1 Introduction . . . 60
5.2 Impact-based Search . . . 61
5.3 Impact Variance . . . 62
5.3.1 Variance . . . 62
5.3.2 Computing a Variance-Estimate . . . 64
5.4 Numerical Results . . . 64
5.4.1 Quasigroup Completion . . . 65
5.4.2 Magic Squares . . . 66
5.4.3 Costas Array . . . 67
5.5 Conclusion . . . 68

IV Interplay Between Search and Inference **70**

6 Efficient Context-Free Grammar Constraints **73**

6.1 Introduction . . . 74
6.2 Basic Concepts . . . 75
6.3 Context-Free Grammar Constraints . . . 77
6.3.1 Parsing Context-Free Grammars . . . 78
6.3.2 Example . . . 79
6.3.3 Context-Free Grammar Filtering . . . 79
6.3.4 Example . . . 82
6.3.5 Runtime Analysis . . . 82
6.4 Efficient Context-Free Grammar Filtering . . . 83
6.4.1 Space-Efficient Incremental Filtering . . . 84
6.5 Numerical Results . . . 92
6.6 Cost-Based Filtering for Context-Free Grammar Constraints . . . 95
6.7 Logic Combinations of Grammar Constraints . . . 97
6.8 Limits of the Expressiveness of Grammar Constraints . . . 100
6.9 Conclusion . . . 104

7 Same-Relation Constraints **105**

7.1 Introduction . . . 106

7.2 Theory Background . . . 106
7.3 Clique Same-Relation . . . 107
7.3.1 Complexity of Achieving GAC . . . 108
7.3.2 Restriction on the Size of Domain . . . 109
7.4 Bipartite Same-Relation . . . 110
7.4.1 Complexity of Achieving GAC . . . 111
7.5 DAG Same-Relation . . . 111
7.5.1 Complexity of Achieving GAC . . . 112
7.6 Grid Same-Relation . . . 112
7.7 Decomposing Same Relation Constraints . . . 114
7.7.1 Decomposing CSR . . . 114
7.7.2 Decomposing BSR . . . 117
7.8 Numerical Results . . . 119
7.9 Conclusion . . . 121

V Concluding Remarks 122

8 Related Work 123

9 Conclusions 129
9.1 Efficient Search Procedures for Solving Combinatorial Problems . . . 130
9.2 General Advice for Practitioners . . . 131
9.2.1 Dichotomic Search Protocols . . . 132
9.2.2 Dialectic Search . . . 133
9.2.3 Impact-Based Search . . . 133
9.2.4 Context-Free Grammar Constraints . . . 133
9.2.5 Clique-Same Relation Constraint . . . 134
9.3 Future Work . . . 134

Appendix 136
A Experimental Results – Set Covering Problem . . . 137
B Dialectic Search Procedure – Costas Array Problem . . . 145

Bibliography 152

List of Tables

4.1 Numerical Results for the Costas Array Problem. We compare tabu search and dialectic search in terms of minimum, maximum, standard deviation and average solution time in seconds over 100 runs. The tabu search algorithm is implemented in COMET platform, and the dialectic search algorithm is implemented in both COMET platform and C++. 48

4.2 Numerical Results for Continuous Optimization. We give average minimum value and average number of function evaluations over 250 runs for continuous function minimization with dimensions 20 and 50. SA cooling factors are set to 0.98 and 0.99. 50

4.3 Numerical Results for the Set Cover Problem. We present the average solution (standard deviation), best solution (standard deviation), average time to find the best solution, and the time limit used. The results are averaged for each benchmark class in the OR library. Hegel was run 50 times on each instance, ITEG data were taken from [132] who ran their algorithms 10 times on each instance. 52

4.4 Numerical Results for the Set Cover Problem. We present the average runtime (standard deviation) in seconds for finding the best solution in each run, as well as the average solution quality and the best solution quality. Results are averaged for all instances in each benchmark class in the OR library. Hegel was run 50 times on each instance and TS data were taken from [141] who ran their algorithms 10 times on each instance. 53

4.5 Comparison of the default, the instance-oblivious parameters provided by GGA, and the instance-aware parameters provided by SOP for Hegel and TS. We present the average run time in seconds, and the average degradation per instance when using the default or GGA parameters instead of ISAC. 56

5.1 The Quasigroup Completion Problem. We compare impact based search with and without incorporating standard deviation. We consider quasigroups with Order = 40 and Holes = 640 (the table on the top), and with Order = 50 and Holes = 1250 (the table on the bottom). We present the average running time in seconds, and the number of instances solved. We considered 100 instances for each order, and ran each instance with 10 different seeds. The time limit is set to 2,000 seconds. The numbers in bold denote the best in each row. 66

5.2 The Magic Square Problem. We compare impact based search with and without incorporating standard deviation. We present the average runtime in seconds, and the average number of successful trials to find magic squares orders between 5 and 16. We considered 50 instance for each order, and results are averaged over all orders. The time limit is set to 2,000 seconds. The numbers in bold denote the best in each row. 67

5.3 The Costas Array Problem. We compare impact based search with and without incorporating standard deviation. We present the average runtime in seconds, and the average number of successful trials to find Costas arrays of orders between 10 and 19. We considered 50 instance for each order, and results are averaged over all orders. The time limit is set to 2,000 seconds. The numbers in bold denote the best in each row. 68

5.4 Numerical Results using the Minimum Domain Size Heuristic. We present the average running time in seconds, and the number of instances solved when minimum domain size search strategy is used. In the table on the top, we show results on quasigroup completion problems with Order = 40 and Holes = 640 and with Order = 50 and Holes = 1250. We used 100 instances for each order and ran each instance with 10 different seeds, the same setting as in the experiments conducted for the impact-based search. In the table on the bottom, we show results on orders between 5 and 16 for the magic squares problem and orders between 10 and 19 for the Costas array problem. We considered 50 instance for each order, again, used the same settings as impact-based search experiments. Similarly, the time limit is set to 2,000 seconds. 69

6.1 Shift-scheduling: We report running times on an AMD Athlon 64 X2 Dual Core Processor 3800+ for benchmarks with one and two activity types. For each worker, the corresponding grammar in CNF has 30 non-terminals and 36 productions. Column #Propagations shows how often the propagation of grammar constraints is called for. Note that this value is different from the number of choice points as constraints are usually propagated more than just once per choice point. 93

6.2 Effects of Value Ordering: We compare the search trees generated by the default search heuristic and the the search guided by our context-free grammar constraint's propagator. The default search heuristic selects minimum domain size variable and assigns it the minimum value in its domain. The guided search heuristic again selects the variable with minimum domain size, but assigns it the value for which the corresponding production rule has the most number of supports, when both *in* and *out* supports are combined. The time limit is set to 1.000 seconds. We use a dash to indicate an instance that hits the time limit. 94

7.1 Average Speed-up for the Stable Marriage Problem. 120

7.2 Average Speed-up for the Table Planning Problem. 121

A.1 Numerical Results for the Set Cover Problem. We present the average solution (standard deviation), best solution (standard deviation), average time to find the best solution, and the time limit used. Hegel was run 50 times on each instance, ITEG data were taken from [132] who ran their algorithms 10 times on each instance. The authors of ITEG do not report average runtime for each instance which is denoted with a dash. 138

List of Figures

3.1 Dichotomic search for the optimum for a classic binary search and a skewed search when the cost of a negative trial is c and the cost for a positive trial is 1. 23

3.2 Dichotomic search for the optimum for a classic binary and a skewed search when the cost of trials follows a typical easy–hard–less-hard pattern 24

3.3 The Streeter-Smith strategy for constrained optimization on the interval [1,100]. 32

3.4 The progress of the skewing parameter, $a \in [0.5, 1)$, with respect to how costly the negative trials are compared to positive trials. 33

3.5 Comparison of SS, SS-lc, and SS-lc-skewed on weighted magic square problems. 35

3.6 Comparison of SS, SS-lc, and SS-lc-skewed on weighted quasigroup-completion problems. . 36

3.7 Comparison of SS-lc, and SS-lc-skewed on weighted quasigroup-completion problems. . . . 37

3.8 Comparison of SS-lc, and SS-lc-skewed on weighted magic square problems. 38

4.1 Costas Array example. 47

4.2 The global functions used for continuous optimization. 50

4.3 The function 'Merge' for the Set Covering Problem. 51

6.1 Context-Free Filtering: A rectangle with coordinates (i, j) contains nodes v_{ijA} for each non-terminal A in the set S_{ij}. All arcs are considered to be directed from top to bottom. The left picture shows the situation after step (2). S_0 is in S_{14}, therefore the constraint is satisfiable. The right picture illustrates the shrunken graph with sets S'_{ij} after all parts have been removed that cannot be reached from node v_{14S_0}. We see that the value ']' will be removed from D_1 and '[' from D_4. 81

6.2 We show how the algorithm works when the initial domain of X_3 is $D_3 = \{[\}$. The left picture shows sets S_{ij} and the right the sets S'_{ij}. We see that the constraint filtering algorithm determines the only word in $L_G \cap D_1 \dots D_4$ is "[][]". 82
6.3 Regular grammar filtering for $\{a^n b^n\}$. The left figure shows a linear-size automaton, the right an automaton that accepts a reordering of the language. 101

To my parents.

Anneme ve babama,
Gülten ve Necdet Kadıoğlu.

CHAPTER ONE

Introduction

1.1 Solving Optimization and Satisfaction Problems

The problem of finding a satisfactory or the best solution has always been of great interest and practical importance. A variety of constraint satisfaction and constrained optimization problems arise in diverse fields such as artificial intelligence, operations research,and bioinformatics. Some prominent examples include vehicle routing, production planning, finding satisfying assignments for propositional formulae, and predicting the 3D-structure of proteins.

Computational approaches for solving constraint satisfaction and constrained optimization problems consist of mainly two fundamental principles: exploring a vast solution space toward a desired solution while trying to eliminate sub-parts of the solution space which are guaranteed not to have a (better) solution. The former procedure is known as $search$, and the latter procedure is known as $inference$.

Since the time of Aristotle, who was the first to develop a systematic treatment of the principles governing inference and to investigate the formal deductive reasoning, a tremendous amount of research, conducted within very diverse fields, has matured our understanding of inference and reasoning. In Artificial Intelligence, predicate logic has emerged as the lingua franca, and automated logical inference systems that employ forward and backward chaining algorithms (e.g. Prolog [31]) are designed. In Constraint Programming, sophisticated constraint propagation techniques based on graph properties or formal languages, so-called global constraints, are developed. In Operations Research, cutting plane algorithms, a logical implication of a set of inequalities, and certain duality theorems, such as Lagrangean duality, are excelled [142]. In Boolean Satisfiability, based on unit propagation, a well-known method called the Davis-Putnam-Logemann-Loveland procedure [37] formed the basis for efficient solvers. Furthermore, there has been significant interest in developing hybrid methods to bring together the complementary strengths of these disciplines [93, 147, 176, 189]. Overall, decades of accumulated knowledge on inference is now embodied in today's high-performance solvers and decision support systems.

1.2 Motivation

All the above-mentioned paradigms for solving constraint satisfaction and constrained optimization problems share one common point. They use deterministic inference methods, in one way or another, to accelerate the *search* to find a satisfying or provably optimal solution. Advanced inference techniques are applied to reduce the search space, while a combination of variable and value selection heuristics are used to guide the exploration of that search space, yet still, the search generally involves making uncertain decisions.

From a computational complexity point of view, most of these problems are very difficult to solve due to their intrinsic complexity. Assuming that $P \neq NP$, on one hand, we have provably polynomial inference algorithms, and on the other hand, we attempt to solve NP-$hard$ problems. This means that we rely heavily on (exponential) search for solving these problems. In fact, search is an integral part of solution approaches for NP-hard combinatorial optimization and decision problems. Once the ability to reason deterministically is exhausted, state-of-the art solvers try out different alternatives which may lead to an improved (in case of optimization) or feasible (in case of satisfaction) solution. This consideration of alternatives may take place highly opportunistically as in local search approaches, or systematically as in backtracking-based methods.

Efficiency could be much improved if we could effectively favor alternatives that lead to optimal or feasible solutions and a search space partition that allows short proofs of optimality or infeasibility. After all, the existence of an ”oracle” is what distinguishes a non-deterministic from a deterministic Turing machine. This of course means that perfect choices are impossible to guarantee. The important insight is to realize that this is a worst-case statement. In practice, we may still hope to be able to make very good choices on average. As a consequence, we believe that there is an immense potential for improvement by boosting average-case search performance, and this is the very aspect we study in this thesis.

In this thesis, we develop efficient search procedures that can be used in a tree search approach, design a dichotomic search protocol for constrained optimization, and introduce a novel local search meta-heuristic.

The existing search methods can be classified into two main classes: *complete* and *incomplete* search methods. Complete search algorithms are guaranteed to find the optimal solution and to prove its optimality. If optimal solutions cannot be computed efficiently in practice, the only possibility is to trade optimality for efficiency. That is, the guarantee of finding optimal solutions can be sacrificed for the sake of getting very good solutions quickly. In this thesis, we consider both complete and incomplete search as they are complementary to each other. We also look at variable and value selection heuristics as they play a key role in the performance of search algorithms. Moreover, we investigate the interplay between search and inference.

1.3 Background

We provide general information about search algorithms in Chapter 2. While each chapter in this thesis is self-contained to a large extent and includes necessary background information, we assume that readers have some familiarity with the basic concepts of algorithms, complexity theory, and constraint programming. For

in-depth introductions, we refer the reader to:

- **Search and Algorithms**
  - Cormen, Leiserson, Rivest and Stein: *Introduction to Algorithms* [33].
  - Russell and Norvig: *Artificial Intelligence - A Modern Approach* [163].
  - Hoos and Stüzle: *Stochastic Local Search* [95].
- **Constraint Programming**
  - Marriott and Stuckey: *Programming with Constraints: An Introduction* [133].
  - Kumar: *Algorithms for Constraint-Satisfaction Problems: A Survey* [123].
  - Apt: *The Rough Guide to Constraint Propagation* [3].

1.4 Contributions and Outline

We start with a gentle introduction to complete and incomplete search algorithms in Chapter 2. The rest of the thesis can be categorized under four main themes, and our contributions in each one can be summarized as follows.

1.4.1 Part I – Complete search

We first study binary search, a very basic but widely used method for constrained optimization problems, and devise a theoretical model for skewed binary search. We show that a certain way of choosing the breaking point minimizes both expected as well as worst case performance. Furthermore, we show that, under the investigated cost model, our protocol is optimal in the expected and worst case [167]. We demonstrate performance gains when skewed binary search is used within the search strategy by Steeter and Smith [175].

1.4.2 Part II – Incomplete search

We introduce Hegel and Fichte's dialectic as a new local search meta-heuristic and provide empirical evidence that it strikes an appealing balance between exploration and exploitation. We illustrate the simplicity

and the efficiency of dialectic search on different problems from highly different problem domains: constraint satisfaction, continuous optimization, and discrete optimization. We show that the dialectic search algorithm devised for the set covering problem, one of the most studied combinatorial optimization problems, outperforms previous works in both solution quality, and running time, and discovers previously unknown solutions. Further, we boost the performance of our set covering solver based on dialectic search using instance-oblivious and instance-specific algorithm configuration.

1.4.3 Part III – Variable and value selection

Search heuristics used for variable and value selection have enormous effect on the performance of search algorithms. We consider a general method used in Constraint Programming; namely impact-based search. Impact-based search measures the average reduction in search space due to propagation after a variable-value assignment has been committed, and favors the assignment with the highest reduction in the search space. However, this estimate on the reduction of the search depicts a variance. Rather than considering the mean reduction only, we consider the idea of incorporating the variance in reduction when choosing a branching variable during search. Experimental results on three combinatorial design problems show that using variance can result in improved search performance.

1.4.4 Part IV – Interplay between search and inference

Solving combinatorial problems is an interplay between search and inference. While we use the search to advance toward a proof (solution), inference plays a dual role by detecting infeasible search directions and, when possible, preventing the search from even trying directions which can be proved wrong a priori. This continuous interplay is the key behind many successful complete search solvers. In other words, inference methods in separation do not yield to a solution, instead, they are embedded in a tree search approach that interleaves branching decisions with inference. As such, inference algorithms are executed many times during search. In the last part of our thesis, we turn our focus to the fact that significant performance gains are possible if inference algorithms can be maintained incrementally during search.

Constraints based on context-free grammars provide a perfect example for such a case where the complexity of existing propagation algorithms [154, 155, 165] is prohibitive for tree search. Our contribution in this line of work is to devise a time-and space-efficient propagation algorithm that can be maintained incrementally during search. In particular, we improve the space requirements of the context-free grammar propagator by a linear factor. Moreover, we show how this propagator can be used to guide value selection during search.

Boosting search performance is also possible through leveraging the knowledge about problem structure. We consider binary constraint satisfaction problems where sets of pairs of variables all share the same relation. In particular, we investigate problems with special associated constraint graphs like cliques, complete bipartite graphs, and directed acyclic graphs whereby we always assume that the *same* constraint is enforced on all edges in the graph. Our theoretical contribution is to show that most of these constraints pose NP-hard filtering problems. On the practical side, we provide substantial improvements in both asymptotic and run time performance using new, generic algorithms that use the knowledge that all pairs of variables share the relation.

1.5 Related Publications

The work presented here appeared in a number of publications previously. While the content of this thesis is based on these publications to a great extend, it is not a mere collection of an array of papers, but rather makes significant additions to the published versions. Our incremental changes in each paper can be summarized as follows.

– In Chapter 3, we extend the work presented in [167] with additional experiments that were carried out using a more recent commercial solver.

– The content of [110] is updated in Chapter 4. The changes include new experimental results that compare the performance dialectic search algorithm and the tabu search algorithm implemented in the same framework. Also, the details of dialectic search algorithm as well as the experimental results on individual set covering instances, which were missing in [110], are now presented in the Appendix. Moreover, the work done in [107] is unified into this chapter as an addition to the numerical results section.

– The experimental results from [108] on variable and value selection heuristics is updated to include the performance of the minimum domain search heuristic on the same problems. We also make a connection between [110] and [108] in this chapter, regarding the behavior of the problem classes that we considered.

– In Chapter 6, we present a new algorithm that exploits the structure behind the filtering algorithm of the context-free grammar constraints in order to guide the search. This algorithm connects the inference mechanism with the search toward a feasible solution. New experiments reveal improvements in solving efficiency when value selection is directed using the information provided by the constraint's propagator. Also, the details of the incremental filtering algorithm which was omitted in [109] are now included in this section.

– The filtering algorithm for same-relation constraints based on AC-4 schema was not presented in [104]. We now include the details of this propagator in Chapter 7.

I was fortunate to have outstanding co-authors and I am indebted to them for their many contributions to the work described in this thesis.

In our joint work on impact-based search (Chapter 5), Philippe Refalo has provided important insights about how the impact-based search strategy works, and Eoin O'Mahony provided me with the instances we used for the Quasigroup completion problem. I implemented the impact-based search strategy and its variant that incorporates the variance information. I also implemented the online variance calculation which is due to [121]. I conducted the experiments and all authors contributed to writing the paper [108].

The idea of same-relation constraints (Chapter 7) has started on constraint networks where there exists a constraint between every pair of variables. Our co-authors Christopher Jefferson, Karen E. Petrie and Standa Živný helped us extending this notion to other constraint networks such as complete bipartite graphs, DAGs and grids. I provided an implementation of the clique based same-relation constraint in IBM Ilog Solver based on AC-6 and AC-4 algorithms while Christopher Jefferson provided implementations using Minion Solver. The experiments presented in the paper are conducted by Christoper Jefferson. All authors contributed to writing the paper [104].

My contribution in [107] is the section regarding the set covering problem for which the ISAC framework is used to boost the performance of both dialectic search and tabu search algorithms.

Journal paper

- S. Kadioglu and M. Sellmann. Grammar Constraints. *Constraints Journal*, 15(1):117–144, 2010.

International conference papers

- M. Sellmann and S. Kadioglu. Dichotomic Search Protocols for Constrained Optimization. *Proceedings of the Fourteenth International Conference on the Principles and Practice of Constraint Programming (CP)*, LNCS, 5202:251–265, Springer, 2008.
  Nominated for the Best Paper award.
- S. Kadioglu and M. Sellmann. Dialectic Search. *Proceedings of the Fifteenth International Conference on the Principles and Practice of Constraint Programming (CP)*, LNCS, 5732:486–500, Springer, 2009.
- S. Kadioglu, Y.Malitsky, K. Tierney and M. Sellmann. ISAC – Instance Specific Algorithm Configuration. *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI-2010)*, 751–756, 2010.

- S. Kadioglu, E. O'Mahony, P. Refalo and M. Sellmann. Incorporating Variance in Impact-Based Search. *Proceedings of the Seventeenth International Conference on the Principles and Practice of Constraint Programming (CP)*, LNCS, 470–477, Springer, 2011.

- S. Kadioglu and M. Sellmann. Efficient Context-Free Grammar Constraints. *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI)*, 310–316, 2008.

- C. Jefferson, S. Kadioglu, K. E. Petrie, M. Sellmann, S. Živný. Same-Relation Constraints *Proceedings of the Fifteenth International Conference on the Principles and Practice of Constraint Programming (CP)*, LNCS, 5732:470–485, Springer, 2009.

Reviewed workshop paper

- S. Kadioglu and M. Sellmann. A Local Search Meta-Heuristic for Non-Specialists. In *Proceedings of CP-2009 Satellite Workshop on Constraint Reasoning and Optimization for Computational Sustainability (CROCS-09)*, 7–8, 2009.

CHAPTER TWO

Background

The task that a symbol system is faced with, then, when it is presented with a problem and a problem space, is to use its limited processing resources to generate possible solutions, one after another, until it finds one that satisfies the problem defining test. If the symbol system had some control over the order in which potential solutions were generated, then it would be desirable to arrange this order of generation so that actual solutions would have a highly likelihood of appearing early. A symbol system would exhibit intelligence to the extent that it succeeded in doing this. Intelligence for a system with limited processing resources consists in making wise choices of what to do next....

— Newell and Simon, *Turing Award Lecture* (1976)

The fundamental idea behind the search approach is to iteratively generate and evaluate candidate solutions. In the case of decision problems, evaluating a candidate solution means checking whether it is an actual solution, and in the case of optimization problems it means calculating the respective value of a given objective function. In either case, the evaluation of candidate solutions is problem dependent. The fundamental differences between search algorithms lies in the way they generate candidate solutions, which can have a significant impact on the algorithm's theoretical properties and practical performance.

In this chapter, we begin with two different classifications of search algorithms based on their underlying concepts to generate candidate solutions. Then, we review systematic search algorithms with respect to various node selection strategies. Finally, we present an overview of prominent local search metaheuristics.

2.1 Perturbative vs. Constructive Search

For this and the following classification of search algorithms, we borrow the notion of *solution components* from [95]. Given an instance of a combinatorial problem, solution components can be seen as the building blocks of candidate solutions. For example, in Travelling Salesman Problem, the solution components are cities in the order in which they are traversed. By swapping a city with another city in a given permutation, we can easily generate other candidate solutions. This procedure is known as perturbing a given candidate solution. Search algorithms that rely on this mechanism for generating the candidate solutions are classified as *perturbative search methods*.

Candidate solutions in which one or more solution components are missing are referred to as partial candidate solutions. Given a partial candidate solution, we can build a candidate solution by iteratively extending the partial solution components. This procedure is called *constructive search*. Value orderings

can be seen as partial solutions. It can even be used to learn favorable value-selection heuristics over the course of several restarts during search [166].

2.2 Systematic vs. Local Search

Another classification of search approaches, as we stated in the introduction of this thesis, is based on the distinction between complete and incomplete search.*Systematic search algorithms* partition the space of potential solutions searches into different parts systematically. The main aspects of systematic solvers are how the solution space is partitioned, and in what order the different parts are to be considered. Both choices have an enormous impact on solution efficiency. Systematic search algorithms have the property that they can prove the optimality of a solution or the inexistence of solutions. This property of systematic search is called *completeness*.

This notion is in contrast to *local search algorithms*. Local search algorithms are initiated with a candidate solution from the potential search space. Then, they move from the current solution to neighboring solutions in the search space. Each move is based on the local knowledge only. Local search algorithms cannot be used to prove infeasibility as they are *incomplete*, and there is no guarantee that a solution will eventually be found. Since the search is conducted in a non-systematic manner, it is possible that local search methods can generate the same candidate solution more than once. This is the main disadvantage of local search algorithms, that they can get stuck in some part of the search space. In order to overcome this limitation, special mechanisms are introduced such as restarting the search process from a new (randomly) generated candidate solution or applying some type of exploration steps to escape locally optimum solutions.

Local search methods are often based on perturbative search, but it can also be used for constructive search processes. A common way to bring the two methods together is to use constructive search in order to generate the starting point for perturbative search. High-performance local search algorithms make use of randomized choices in generating or selecting candidate solutions. These algorithms are also called *stochastic local search algorithms* [95].

Systematic search algorithms can be decomposed into a constructive search method and backtracking. Backtracking is the process of reverting the search process to the most recent choice point where there remains unexplored alternatives. Once the search process 'backtracks', an alternative option is selected, and the constructive search is resumed from this point where the alternative option is applied. This procedure generates a structure which is referred to as *search tree*. Generating all solutions using backtracking search can quickly become intractable even for small problem instances. It is however possible to prune large

parts of tree search which can be shown to not contain any (better) solutions. In the beginning of this thesis, we referred to this line of reasoning as *inference*. For example, in operations research the *branch & bound* algorithm is aimed at exactly for this. By recursively partitioning the problem into sub-problems ("branching"), the algorithm systematically covers all parts of the search space. When the objective is to minimize, a relaxation of the problem is used to compute an under-estimate of the best solution for a given sub-problem ("bounding"). By comparing this bound with the best previously found solution, it possible to deduct that a given sub-problem cannot contain improving solutions, which allows to discard (or "prune") the sub-problem from further consideration. There exist a variety of relaxations which can be computed efficiently, the most commonly used one is linear relaxation. In Boolean satisfiability, the search tree can be pruned considerably by using *unit propagation*. When a particular variable assignment is committed, the logical consequence of that assignment is propagated which in turn helps discarding subtrees of the search tree that cannot contain a satisfying assignment for the given formula.

The constructive methods are often deterministic, but they can as well be randomized in order to obtain stochastic systematic search algorithms (see e.g. [76]). Another reason for introducing randomization in systematic search algorithms is to tailor them for restarted search procedures. These type of solution approaches was shown to be effective especially for problems that depict heavy-tailed run time distributions [77].

2.3 Systematic Search Algorithms

Various exploration strategies have been defined in the literature for guiding the search, all of them including a procedure to select the next node from the set of candidate nodes. The selection strategy attempts to guide the resolution process toward an objective node, by means of evaluating the candidate nodes and selecting the "most promising". It is common to use an evaluation function for the selection process. The information that a function of this type can incorporate varies widely: measures or bounds of the distance to an objective node, the probability that a node is on the path which leads to an objective state, additive cost measures etc. In the following, we give descriptions of some particular evaluation and selection functions. While some procedures are oriented to find optimal solutions, others merely attempt to find feasible solutions.

2.3.1 British Museum Procedure

This technique consists of finding all feasible solutions, beginning with the smallest, and selecting the one with the best objective function. To generate all feasible solutions any procedure that generates all the states of the search tree can be applied, and in particular the *depth-first search* and *breadth-first search* are commonly used due to their simplicity, with one modification: search does not stop when the first feasible

solution is found. Conceptually, this procedure finds the best solution, but in practice it tends to take an unacceptable amount of time [143].

2.3.2 Depth-First Search (DFS)

The *depth-first search* consists of always exploring a child node of the node most recently branched. If this node does not have successors, a process of backtracking is performed by restarting the search at the nearest ancestor node that possess child nodes not yet explored. This strategy is equivalent to considering the evaluation function of a node as the number of steps from the initial node to the one in question, and selecting the node with the greatest distance to be explored [127]. It is also called *linear search* or *single branch search* [99], *LIFO search strategy* [100, 148] or *search toward the bottom* [1].

2.3.3 Breadth-First Search (BFS)

Dijkstra [43] and Moore [139] both proposed breadth-first search, which is also called *FIFO search strategy* [148]. The breadth-first search consists in always exploring a node at the depth d before any one at the depth $d + 1$; as a result, the nodes are branched in the same order in which they were generated. This strategy is equivalent to considering the evaluation function of a node as the number of steps from the initial node, and selecting the node with the shortest distance to be explored.

2.3.4 Uniform Cost Search

Uniform cost search or smaller cost first search is based on selecting the nodes to branch in a tree, where the arcs are associated with non-negative costs, by the least total cost from the root [5]. If all the arcs have the same associated cost, then uniform cost search is identical to breadth-first search.

2.3.5 Best-First Search

Best-first search, also called *ordered state space search* [5] or *heuristic search* [100], uses an evaluation function, $f(n)$, and the node that seems most promising according to that function is selected to be explored [148]. This procedure can be considered as a general strategy of heuristic search. For instance, depth-first search is a special case of the best first search for $f(n)$ is made equal to the negative value of the depth of the node n, breadth-first search is a special case when $f(n)$ is taken as the depth of the node, and uniform cost search is a special case when $f(n)$ is chosen as the cost of the path of the root to the node n.

2.3.6 A and A* Search

A and A* search are both variants of best-first search. The procedure A selects the node n to be branched by considering the minimal cost associated with reaching the node from the root, $g(n)$, plus a heuristic estimate, $h(n)$, of the minimal cost of reaching an objective node from the node n, $h*(n)$. The node with the smallest value of $f(n) = g(n) + h(n)$ is selected. The procedure $A*$ can be defined as procedure A in which the heuristic estimate $h(n)$ of $h*(n)$ is bounded, and therefore fulfills the admissibility condition $(h(n) \leq h*(n) \forall n)$. Again, the node with the smallest value of $f(n) = g(n) + h(n)$, with $(h(n) \leq h*(n))$ is selected [1].

If there is no heuristic information, i.e., the function $h(n) = 0$, and $g(n)$ is defined as the depth of the node n, the selection strategy of procedure A and $A*$ search is equivalent to that of the breadth-first search. If the function $h(n) = 0$ and $g(n)$ is defined as the negative value of the depth of the node n, then the selection strategy of the next node to be explored pf procedure $A*$ is equivalent to depth-first search. Finally, when the function $h(n) = 0$, the selection strategy of the procedure $A*$ is that of uniform cost search.

2.3.7 Iterative Deepening Search (IDS)

This procedure consists in executing depth-first search iteratively, whereby the depth level of the search is increased at each step. On each iteration, IDS visits the nodes in the search tree in the same order as depth-first search, but overall, order in which nodes are first visited, assuming no pruning, is breadth-first [162]. The result is a search procedure that is effectively breadth-first with the low memory requirements of depth-first search.

2.4 Local Search Meta-Heuristics

George Polya defines *heuristic* as ”the study of methods and rules of discovery and invention” [150]. This meaning can be traced to the term’s Greek root, the verb *eurisco*, which means “I discover“. When Archimedes emerged from his famous bath clutching the golden crown, he shouted “Eureka¡‘ meaning “I have found it¡‘. In search, heuristics are formalized as rules for choosing those branches in a search space that are most likely to lead an acceptable solution.

Unfortunately, heuristics can be fallible. A heuristic is only an informed guess of the next step to be taken in solving a problem. It is often based on experience or intuition. Because heuristics use limited information, they are seldom able to predict the exact behavior of the search space farther along in the search is limited. A heuristic can lead a search algorithm to a suboptimal solution or fail to find any solution at all. This is an

inherent limitation of heuristic search. It cannot be eliminated by ”better” heuristics or more efficient search algorithms [64].

In the early years, specialized heuristics were typically developed to solve complex problems. With the emergence of more general solution schemes the picture drastically changed. Glover used the term *meta-heuristics* for such methods [67]. Basically, a meta-heuristic is a top-level strategy that guides an underlying heuristic to solve a given problem. According to Glover, “it refers to a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality“ [70]. We may also consider the following definition by Osman and Kelly: “A meta-heuristic is an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space using learning strategies to structure information in order to find efficiently near-optimal solutions [146]. Now, the challenge is to adapt a meta-heuristic to a particular problem or problem class, which requires much less work than developing a specialized heuristic from scratch.

In the following, we summarize the basic concepts of the most prevalent meta-heuristics. It is interesting to see that adaptive processes originating from different settings such as pyschology(“learning”), biology(“evolution”), and physics(“annealing”) have served as a starting point for local search meta-heuristics.

2.4.1 Simulated Annealing (SA)

Simulated annealing is a randomized local search procedure. In Simulated Annealing algorithm the acceptance rate for a modification to the current solution leading to an increase in solution cost is based on some probability [26, 118]. This algorithm is motivated from an analogy with the physical annealing process used to find low-energy states of solids. A solution corresponds to a state of the physical system and the solution cost corresponds to the energy of the system. At each iteration, the current solution is modified by randomly selecting a move from a particular definition of a neighborhood solution. If the new solution provides an improvement, it is automatically accepted and becomes the new current solution. Otherwise, the new solution is accepted according to the Metropolis criterion. The probability of acceptance is related to the magnitude of the cost increase and a parameter called *temperature*. Basically, a move is more likely to be accepted if the temperature is high and the cost increase is low. The temperature parameter is progressively lowered, according to some predefined cooling scheme, and a certain number of iterations are performed at each temperature level. When the temperature is sufficiently low, only improving moves are accepted and the method stops at a local optimum. As opposed to most heuristics, this method provably converges to a global optimum, assuming that a sufficient number of iterations will be carried out with the correct cooling parameter.

2.4.2 Tabu Search (TS)

Tabu search is basically a deterministic local search strategy where, at each iteration, the solution in the neighborhood of the current solution is selected as the new current solution, even if it leads to an degradation in solution cost. The method will thus escape from local optimum as opposed to a pure local descent. A short-term memory, known as the tabu list, stores recently visited solutions to avoid short-term cycling. Typically, the search stops after a fixed number of iterations or a maximum number of consecutive iterations without any improvement to the best known solution [67–69].

Starting from the simple search scheme described above, a number of developments and refinements have been proposed over the years.

Frequency memories are used to record how often certain solution attributes are found in previously visited solutions. Neighborhood solutions which contain elements with high frequency counts can then be penalized to allow search to visit other regions of the search space. This mechanism provides a form of diversification by introducing a bias in the evaluation of neighborhood solution at each iteration [70].

Adaptive memories contain a pool of previously generated elite solutions. These solutions are then used to restart the search. This is usually done by taking different fragments of elite solutions and by combining them to generate a new starting solution, similarly to many population based meta-heuristics [161]. Intensification or diversification is obtained depending if the fragments are taken from solutions that lie in a common region of the search or not.

Path relinking [70, 71] generates new solutions by exploring trajectories between elite solutions. Starting from one of these solutions, it generates a path in the neighborhood space leading to another solution. This solution is called the guiding solution. This can be done by selecting modifications that introduce attributes found in the guiding solutions. This mechanism can be used to diversify or intensify the search, depending on the path generation mechanism and the choice of the initiating and guiding solutions.

In strategic oscillation [70], an oscillation boundary (usually a feasibility boundary) is defined. Then, the search is allowed to go for a specified depth beyond the boundary before turning around. When the boundary is crossed again from the opposite direction, the search goes beyond it for a specified depth before turning around again. By repeating this procedure an oscillatory search pattern is produced. It is possible to vary the amplitude of the oscillation to explore a particular region of the search space.

The reactive tabu search [6] provides a mechanism for dynamically adjusting the search parameters, based on the search history. In particular, the size of the tabu list is automatically increased when some configurations are repeated too often to avoid short-term cycles (and conversely).

2.4.3 Greedy Randomized Adaptive Search Procedure (GRASP)

Multi-start greedy search methods repeatedly apply a local search from different initial solutions. If the greedy solutions are different enough to allow for a good sampling of local optima, then the use of a quick greedy heuristic to generate starting solutions looks attractive in this regard. Semi-greedy or randomized greedy heuristics have been proposed to add variability to greedy heuristics [53, 85], and led to the search scheme known as GRASP. GRASP is a multi-start procedure where each restart applies a randomized greedy construction heuristic to generate an initial solution, which is then improved through local search [54]. This is repeated for a given number of restarts and the best overall solution is returned at the end. At each step of the construction heuristic, the elements not yet incorporated into partial solutions are evaluated with a greedy function, and the best elements are kept in a so-called *restricted candidate list* (RCL). One element is then randomly chosen from this list and incorporated into the solution. Through randomization, the best current element is not necessarily chosen, thus leading to a diversity of solutions.

One drawback of GRASP comes from the fact that each restart is independent of the previous ones, thus preventing the exploitation of previously obtained solutions to guide the search. Some recent developments aimed at providing this capability. One example is the reactive GRASP [152] where the size of the RCL is dynamically adjusted, depending on the quality of recently generated solutions. Another example is the use of memories to guide the search. In [57], a pool of elite solutions is maintained to bias the probability distribution associated with the elements in the RCL. Intensification or diversification can be obtained by either rewarding or penalizing elements that are often found in the pool of elite solutions. Such a pool can be used to implement *path relinking* [71], by generating a search trajectory between a randomly chosen elite solution and the current local optimum.

2.4.4 Evolutionary Algorithms

Evolutionary algorithms represent a large class of problem-solving methodologies, with genetic algorithms (GA) [92] being the most widely known. These algorithms are motivated by the way species evolve and adapt to their environment, based on the Darwinian principle of natural selection. Under this paradigm, a population of solutions (often encoded as a bit or integer string, referred to as a *chromosome*) evolves from one generation to the next through the application of operators that mimic those found in nature, namely, selection of the fittest, crossover and mutation. Through the selection process, which is probabilistically biased toward the best elements in the population, only the best solutions are allowed to become *parents* and to generate *offspring*. The mating process, called crossover, then takes two selected parent solutions and combine their most desirable features to create one or two offspring solutions. This is repeated until a new population of offspring solution is created. Before replacing the old population, each member of the new

population is subjected (with a small probability) to small random perturbations via the mutation operator. Starting from a randomly or heuristically generated initial population, this renewal cycle is repeated for a number of iterations, and the best found is returned at the end [8, 73, 151].

The distinctive feature of evolutionary algorithms is the exploitation of population of solutions and the creation of new solution through the recombination of good attributes of parent solutions. Many meta-heuristics integrate this feature (e.g., via adaptive memories). There is a clear connection between recombination, where an intermediate solution is generated from parent solutions and path relinking [157].

2.4.5 Ant Colony Optimization

Another meta-heuristic that works with a population of solutions rather than a single solution is Ant Colony Optimization (ACO). The metaphor behind the technique is as follows. Ants use a chemical compound known as phermone for means of communication among each other. When an ant detects a phermone trail it will follow it, and phermone trail will become strengthened by the ant's own phermone following it. This process results in the increased probability for ants to follow a trail that is previously followed by a number of other ants. In that way, ants can find shortest paths from their nest to food sources, as phermone tends to accumulate faster on shorter paths. Originally the first ant system framework was described by Dorigo in [44] for the Travelling Salesman Problem. A number of refinements have been integrated into this general scheme later on.

In ACO algorithm runs for a fixed number of iterations or until search stagnation occurs. At each iteration, a number of ants a number of artificial ants sequentially construct solutions in a randomized and greedy way. When choosing the next element to be incorporated into a partial solution, the ants consider the amount phermone associated with that element. This is a heuristic evaluation which is based on the previously constructed solutions. In order to allow the construction of a variety of different solutions, a probability distribution is defined over all elements, where the best elements have a higher probability of selection. Each time an element is selected by an ant, its phermone level is updated first removing a fraction of it, which represents the phermone evaporation, and then by adding some new phermone. The search is restarted when all ants construct a complete solution, and the process is iterated again. Detailed descriptions of the algorithm are given for example in [45–47].

A number of mechanism have been integrated into this approach to either intensify or diversify the search. For example, more phermone are associated with elements found the incumbent solution. This way, a more intense search around that solution is conducted. Conversely, the phermone levels may be reduced for some elements in order to construct more diverse solutions. A successful variant of this algorithm is known as Max-Min Ant System where phermone levels are bounded [178, 179].

Part I

Complete Search

Chapter Three

Dichotomic Search Protocols for Constrained Optimization

In this chapter, we focus on complete search procedures and devise a theoretical model for dichotomic search algorithms for constrained optimization. We show that, within our model, a certain way of choosing the breaking point minimizes both expected as well as worst case performance in a skewed binary search. Furthermore, we show that our protocol is optimal in the expected and in the worst case. Experimental results illustrate performance gains when our protocols are used within the search strategy by Streeter and Smith. While the content in this chapter is based on our paper [167], the numerical results section is extended with a new set of experiments that uses a latest commercial solver.

3.1 Introduction

In Constrained Optimization, there are two fundamental strategies being used to find and prove optimal feasible solutions. By far the most common strategy is branch-and-bound. By recursively partitioning the problem into sub-problems ("branching"), we systematically cover all parts of the search space. When our objective is to minimize costs, we use a relaxation of the problem to compute an under-estimate of the best solution for a given sub-problem ("bounding"). By comparing this bound with the best previously found solution, we may find that a given sub-problem cannot contain improving solutions, which allows us to discard (or "prune") the sub-problem from further consideration. There exist a variety of relaxations which can be computed efficiently, the most commonly used is linear relaxation.

Obviously, the efficiency of a branch-and-bound approach depends heavily on the quality of the bounds. For many problems, standard relaxation techniques are reasonably accurate or they can be improved to be reasonably accurate, for example by automatically adding valid inequalities to a linear programming formulation. However, for some problems we have grave difficulty in providing lower bounds that can effectively prune the search. In particular, by exploiting constraint filtering techniques, in Constraint Programming (with few exception such as optimization constraints [58]) the primary focus is on feasibility and not on optimality considerations.

In order to augment a black-box feasibility solver to handle discrete objective functions, there exists a second strategy known as "dichotomic" or "binary search." Given an initial interval $[l, u]$ in which the optimal objective value must lie, we can compute the optimum by testing whether a cost lower or equal $l + \lfloor (u-l)/2 \rfloor$ can still be achieved. If so, we continue searching recursively in $[l, l + \lfloor (u-l)/2 \rfloor - 1]$. If not, we know the optimum must lie in $[l + \lfloor (u-l)/2 \rfloor + 1, u]$. When a query to the feasibility solver incurs a cost of T, using classic binary search we can compute the optimum in time $O(T \log(u-l))$.

An implicit assumption in dichotomic search is that positive trials incur the same costs as negative trials. However, based on our empirical knowledge from phase transition experiments [35, 91, 119, 138] we expect

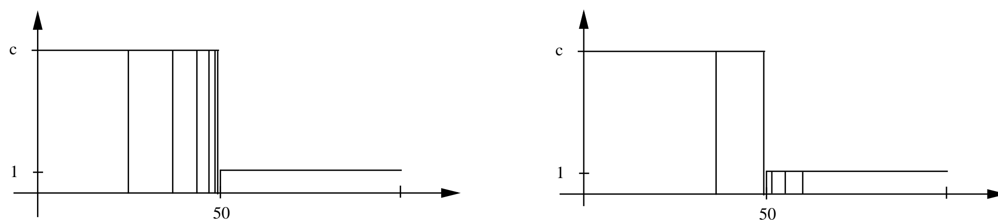


Figure 3.1: Dichotomic search for the optimum 50 in the interval [0,100] when the cost of a negative trial is c and the cost for a positive trial is 1. The left picture illustrates the costs of a classic binary search, the right the costs of a skewed search.

that negative trials, where we prove that no better solution exists, are generally more costly than positive trials, where we only need to find one improving solution.

Assume that we are trying to minimize costs within the interval $[0, 100]$, and the true minimum is (seemingly conveniently) 50. A classic binary search hits the optimum immediately, and then attempts to find solutions with objective lower or equal 24, 37, 43, 46, 48, and 49. While we need to consider the bound 49 in any case to prove optimality of 50, given that a proof of unsatisfiability may be costly, it is unfortunate that binary search considers a rather large number of almost satisfiable instances before 49.

To avoid this situation, we could of course start with an upper bound of 100, and whenever we find a solution with value v only require that from then on we are only interested in solutions with objective value $v - 1$ or lower (see for example the minimization goal in Ilog CP Solver). The downside of this strategy is that we may end up making very slow progress in finding improving solutions.

Our objective is therefore to devise a strategy that allows fast upper bound improvement while avoiding as much as possible costly proofs of unsatisfiability. In particular, we consider *skewed binary searches* [20] where we do not split the remaining objective interval in half but according to a given ratio a. In our example above, assume we use $a = 0.6$ to organize our dichotomic search. Then, we consider 60, 35, 49, 55, 52, 50. Compared to classic binary search, we see that this skewed search considers a number of almost infeasible problems instead of almost feasible problems. In Figure 3.1 we illustrate the costs of classic binary search and the skewed binary search in a model where a negative trial costs of a factor $c \geq 1$ more than a positive trial.

Based on the community's empirical experience on typical runtime over constrainedness, we expect that finding near-optimal solutions is often significantly easier than proving optimality/infeasibility. In Figure 3.2 we sketch the two dichotomic searches when assuming a typical curve describing the cost of finding a feasible solution or proving infeasibility for a given upper bound on the objective with the typical easy–hard–less-hard regions (when considering subproblems with increasing constrainedness in the sketch from right to left).

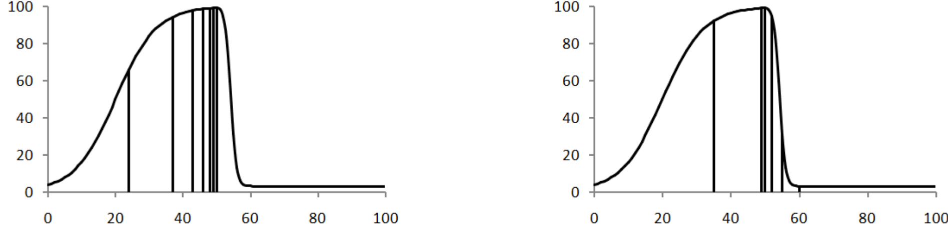


Figure 3.2: Dichotomic search for the optimum 50 in the interval [0,100] when the cost of trials follows a typical easy–hard–less-hard pattern. The left picture illustrates the costs of a classic binary search, the right the costs of a skewed search.

In this section, we provide dichotomic search protocols for such skewed search problems. In particular, we consider the theoretical model where failures incur costs a factor $c \geq 1$ more than positive trials. For this model, we devise a provably optimal dichotomic search protocol. We then exploit this protocol in a heuristic algorithm which integrates dichotomic search and restarted branch-and-bound. Experimental results on weighted quasi-group and weighted magic square problems illustrate the performance improvements achieved by the new algorithm.

3.2 Skewed Binary Search

We consider the following theoretical model.

Definition Given a search-interval $\{l, \dots, u\}$ and a function $f : \{l, \dots, u\} \rightarrow \{0, 1\}$ such that $f(x) = 1 \Rightarrow f(x+1) = 1 \; \forall \; l \leq x < u$, we call the problem of finding $y = \min\{x \in \{l, \dots, u\} \mid f(x) = 1\}$ a *dichotomic* or *binary search problem*. We call the test whether $f(x) = 1$ for some $x \in \{l, \dots, u\}$ a *trial at* x. A trial at x is called *positive* when $f(x) = 1$, otherwise its called *negative* or a *failure*. If the cost of a negative trial is c times the cost of a positive trial for some $c \geq 1$, we call c the *bias*. A binary search problem is called *skewed* when $c > 1$. An algorithm that makes trials at x to continue its search in $\{x+1, \dots, u\}$ in case of a failure and in $\{l, \dots, x-1\}$ in case of a positive trial is called a *(skewed) dichotomic search* or a *(skewed) binary search*. In the case that the search considers trials $x = l + \lfloor a(u-l) \rfloor$ for some constant $a \in [0, 1]$, we call a the *balance* of the search.

Theorem 3.2.1. *When we assume a uniform distribution of optima in the given interval, the expected effort for a skewed binary search with bias $c \geq 1$ is minimized when setting the balance $a \in [0.5, 1)$ such that*

$$a^c + a = 1.$$

Proof. Let us assume our search interval has length $n \in \mathbb{N}$. According to [20,145,191], the expected search cost in a skewed binary tree with balance a is in $\Theta(f(a))$ with[1]

$$f(a) := \frac{a + (1-a)c}{-a \log a - (1-a)\log(1-a)} \log(n) + c.$$

Let us denote with $H(a) := -a\log(a) - (1-a)\log(1-a) \in (0,1]$ the entropy of $a \in (0,1)$. Then, for the first derivative of f, it holds

$$f'(a) = \left(\frac{(a + (1-a)c)(\log a - \log(1-a))}{H^2(a)} - \frac{(c-1)}{H(a)} \right) \log(n) \tag{3.1}$$

$$= (a(\log(a) - \log(1-a)) + c\log(a) + cH(a) - cH(a) + H(a)) \frac{\log(n)}{H^2(a)} \tag{3.2}$$

$$= \frac{(c\log(a) - \log(1-a))}{H^2(a)} \log(n) \tag{3.3}$$

We note that the sign of the first derivative depends solely on the sign of $c\log(a) - \log(1-a)$. When a satisfies $a^c + a = 1$ then $f'(a) = 0$. For all lower values for $a \in [0.5, 1)$ the derivative is negative, for all larger values it is positive. Consequently, a with $a^c + a = 1$ marks a global minimum of f in the interval $[0.5, 1)$. □

When our objective is to minimize expected costs under the uniform distribution, the previous theorem tells us how to choose the balance a. The question arises how we should choose a when our goal is to minimize the worst-case performance. Interestingly, we find:

Theorem 3.2.2. *The worst-case effort for a skewed binary search with bias $c \geq 1$ is minimized when setting the balance $a \in [0.5, 1)$ such that $a^c + a = 1$.*

Proof. When searching an interval of length $n \in \mathbb{N}$, the worst-case effort of a skewed binary search with balance a is given by the value of the following optimization problem: Maximize $x + cy + c$ such that $a^x(1-a)^y \geq 1/n$, $x, y \geq 0$. We linearize this optimization problem and get

[1] The additional summand c is caused by the fact that we incur costs at nodes and not on branches.

$$
\begin{aligned}
\max \quad & x + cy + c \\
\text{such that} \quad & \log\left(\frac{1}{a}\right) x + \log\left(\frac{1}{1-a}\right) y \leq \log(n) \\
& x, y \geq 0
\end{aligned}
$$

From linear programming theory we know that the maximum is achieved in a corner of this 2-dimensional polytope. The maximum value is thus in

$$
\Theta\left(\max\left\{\frac{1}{\log\left(\frac{1}{a}\right)}, \frac{c}{\log\left(\frac{1}{1-a}\right)}\right\}\log(n) + c\right).
$$

Since $-log(a)$ is strictly monotonically decreasing and $-log(1-a)$ is strictly monotonically increasing over $[0.5, 1)$, this cost is minimized when choosing the balance $a \in [0.5, 1)$ such that $\frac{1}{\log(\frac{1}{a})} = \frac{c}{\log(\frac{1}{1-a})}$, which is the same as $\log(1-a) = c\log(a)$, or $1 = a^c + a$.

□

Consequently, we conveniently minimize both expected and worst-case time when setting $a \in [0.5, 1)$ such that $a^c + a = 1$. Then, for the runtime it holds:

Lemma 3.2.3. *The expected and worst-case costs of a skewed binary search with bias $c \geq 1$ and balance $a \in [0.5, 1)$ such that $a^c + a = 1$ are in $\Theta\left(c\left(\frac{\log(n)}{\log(\frac{1}{1-a})} + 1\right)\right)$.*

Proof. First, note that $a^c + a = 1$ iff $c = \frac{log(1-a)}{\log(a)}$. Recall from the proof of Theorem 3.2.1 that the expected runtime is in $\Theta(f(a))$ with

$$
f(a) = \frac{a + (1-a)c}{-a\log a - (1-a)\log(1-a)}\log(n) + c.
$$

Then,

$$
\begin{aligned}
f(a) &= \frac{a\log(a) + (1-a)\log(1-a)}{(-a\log a - (1-a)\log(1-a))\log(a)}\log(n) + c && (3.4) \\
&= \frac{\log(n)}{-\log(a)} + c && (3.5) \\
&= \frac{c}{\log(\frac{1}{1-a})}\log(n) + c && (3.6)
\end{aligned}
$$

Regarding the worst-case runtime, recall from the proof of Theorem 3.2.2 that $a^c + a = 1$ implies $\frac{1}{\log\left(\frac{1}{a}\right)} = \frac{c}{\log\left(\frac{1}{1-a}\right)}$. Then,

$$\Theta\left(\max\left\{\frac{1}{\log\left(\frac{1}{a}\right)}, \frac{c}{\log\left(\frac{1}{1-a}\right)}\right\}\log(n) + c\right) = \Theta\left(\frac{c}{\log\left(\frac{1}{1-a}\right)}\log(n) + c\right).$$

□

So we essentially gain a factor of $\log(\frac{1}{1-a})$ by skewing our search. The question arises how big this factor is in terms of the given bias c.

Lemma 3.2.4. *Given $c \geq 1$ and $a \in [0.5, 1)$ such that $a^c + a = 1$, we have that*

$$\log\left(\frac{1}{1-a}\right) \geq \frac{\log(c)}{2}.$$

Proof. Since $a^c + a = 1$ is equivalent with $c = \frac{\log(1-a)}{\log(a)}$, it is sufficient to show that

$$\left(\frac{1}{1-a}\right)^2 \geq \frac{\log(1-a)}{\log(a)},$$

or equivalently that $(1-a)^{(1-a)^2} - a \geq 0$. Let us define $b := 1-a \in (0, 0.5]$, $x := 1/b \geq 2$, and $g(b) := b^{b^2} + b - 1$. Our claim is then equivalent to showing that

$$b^{b^2} + b - 1 = g(b) \geq 0$$

for all $b \in (0, 0.5]$. Consider the first derivative of g:

$$g'(b) = b^{b^2+1}(1 + 2\ln(b)) + 1.$$

To show that g is monotonically increasing over $(0, 0.5]$, we show that $g'(b) \geq 0$ for all $b \in (0, 0.5]$. Since $1 + 2\ln(b) < 0$ for all $b \in (0, 0.5]$, it is sufficient to show that $b(1 + 2\ln(b)) + 1 \geq 0$, or equivalently, that

$$h(x) := 1 + x - 2\ln(x) \geq 0 \qquad \forall\, x \geq 2.$$

A simple extremum analysis based on the first and second derivative of h shows that h is convex and takes its unique minimum for $x = 2$. Since $f(2) > 0$, we have shown that $g'(b) \geq 0$ over $(0, 0.5]$, and therefore

that g is monotonically increasing over $(0, 0.5]$. However, as g approaches 0 from above, we have

$$\lim_{b \to 0^+} g(x) = \lim_{b \to 0^+} b^{b^2} + b - 1 \tag{3.7}$$

$$= \lim_{b \to 0^+} e^{b^2 \ln(b)} - 1 \tag{3.8}$$

$$= e^{\lim_{b \to 0^+} b^2 \ln(b)} - 1 \tag{3.9}$$

$$= e^0 - 1 = 0. \tag{3.10}$$

Consequently, $g(b) \geq 0$ for all $b \in (0, 0.5]$.

□

With the help of Lemmas 3.2.3 and 3.2.4, we get immediately:

Theorem 3.2.5. *The expected and worst-case costs of a skewed binary search with bias $c \geq 1$ and balance $a \in [0.5, 1)$ such that $a^c + a = 1$ are in $O\left(c\left(\frac{\log(n)}{\log(c)} + 1\right)\right)$.*

To summarize our findings so far: Given a minimization problem where negative trials cost a factor $c \geq 1$ more than positive ones, we minimize the (expected and worst-case) costs of a skewed binary search by choosing the balance $a \in [0.5, 1)$ such that $a^c + a = 1$. With this setting, we essentially gain an asymptotic factor in $\Omega(\log(c))$.

The question arises whether there are other protocols that could minimize the costs further. For example, one may consider a protocol where the balance is not chosen as a constant for the entire search, but that $a \in [0, 1]$ is set in each iteration according to some function over c and also n, the remaining interval length. The following theorem proves that all other dichotomic search protocols cannot perform asymptotically better.

Theorem 3.2.6. *Given an interval with length n, considering the breaking point $a \cdot n$ with $a^c + a = 1$ in a skewed binary search with bias $c \geq 1$ is expected optimal in the O-calculus when we assume a uniform distribution of optima in the given interval.*

Proof. Consider a dichotomic search protocol that selects the next trial according to some function $s(c, n)$. For any given interval length $n \in \mathbb{N}$ and bias $c \geq 1$ we show that the expected time that a skewed search using function s takes is greater or equal $\frac{c}{\log\left(\frac{1}{1-a}\right)} \log(n) + \frac{c}{2}$, where $a^c + a = 1$. We induce over n. For $n = 1$ the claim is trivially true. Now assume $n > 1$ and that the claim holds for all $m < n$. Denote with $p = s(c, n)$ the current trial point. Given that the chance for a positive trial at p is p/n (and $(n-p)/n$ for a

negative trial), and by induction hypothesis, for the expected costs it holds that

$$\mathrm{cost}(s,c,n) \geq \frac{n-p}{n}\left(c+\frac{c\log(n-p)}{\log\left(\frac{1}{1-a}\right)}+\frac{c}{2}\right)+\frac{p}{n}\left(1+\frac{c\log(p)}{\log\left(\frac{1}{1-a}\right)}+\frac{c}{2}\right).$$

With $c = \frac{\log\left(\frac{1}{1-a}\right)}{\log\left(\frac{1}{a}\right)}$, it follows

$$\mathrm{cost}(s,c,n) \geq c+\frac{\log(n-p)}{\log\left(\frac{1}{a}\right)}+\frac{p}{n}\left(1+\frac{\log(p)-\log(n-p)}{\log\left(\frac{1}{a}\right)}-c\right)+\frac{c}{2} \tag{3.11}$$

$$= \frac{1}{\log\left(\frac{1}{a}\right)}\left(\log\left(\frac{n-p}{1-a}\right)+\frac{p}{n}\log\left(\frac{(1-a)p}{a(n-p)}\right)\right)+\frac{c}{2}. \tag{3.12}$$

Since we wish to show $\mathrm{cost}(s,c,n) \geq \frac{c\log(n)}{\log\left(\frac{1}{1-a}\right)}+\frac{c}{2} = \frac{\log(n)}{\log\left(\frac{1}{a}\right)}+\frac{c}{2}$, it is therefore sufficient to show that

$$\log\left(\frac{n-p}{n(1-a)}\right)+\frac{p}{n}\log\left(\frac{(1-a)p}{a(n-p)}\right) \geq 0,$$

or equivalently that

$$\left(\frac{(1-a)p}{a(n-p)}\right)^{\frac{p}{n}} \geq \frac{n(1-a)}{n-p},$$

or that

$$t(a) := \left(\frac{p}{a}\right)^{\frac{p}{n}} - n\left(\frac{1-a}{n-p}\right)^{\frac{n-p}{n}} \geq 0.$$

For the first and second derivation of t we have

$$t'(a) = \left(\frac{1-a}{n-p}\right)^{\left(\frac{n-p}{n}\right)-1} - \frac{1}{n}\left(\frac{p}{a}\right)^{\frac{p}{n}+1} \qquad \text{and}$$

$$t''(a) = \frac{p}{a^2 n}\left(\frac{p}{a}\right)^{\frac{p}{n}}\left(\frac{p}{n}+1\right)+\frac{1}{n-p}\left(1-\frac{n-p}{n}\right)\left(\frac{1-a}{n-p}\right)^{\frac{n-p}{n}-2}.$$

Clearly, $t''(a) > 0$ for all $a \in [0.5, 1)$, and therefore t is convex on this interval. Furthermore, $t'(\frac{p}{n}) = 0$ and $t(\frac{p}{n}) = 0$, and therefore $t(a) \geq 0$ over $[0.5, 1)$.

□

As a consequence of the previous theorem and Lemma 3.2.3, which states that our skewed binary search protocol does not work worse in the worst-case than it does in the expected case, we finally get:

Corollary 3.2.7. *Given an interval with length n, considering the breaking point $a \cdot n$ with $a^c + a = 1$ in a skewed binary search with bias $c \geq 1$ is asymptotically optimal in the worst-case.*

3.3 Skewed Dichotomic Search for Constrained Optimization

The previous theoretical study, while applicable in realistic scenarios like the one considered in [20], cannot be exploited directly when considering constrained optimization. This is for various reasons. First of all, as we discussed earlier and illustrated in Figure 3.2, in optimization practice, failures do not generally incur costs that are a constant factor higher than those of positive trials. Consequently, there is a disconnect between the theoretical model and reality.

The second reason why our protocol is not directly applicable is because, in practice, we do not actually know the factor by which a negative trial is – say, on average – more expensive than a positive trial. We could of course try to estimate such a ratio based on our experience with past trials. However, when the skewed search actually works well we hope to avoid negative trials as best as we can, so the sampling is skewed and there will be very little statistical data to work with. Furthermore, in some cases the lower bounds on the objective that we can compute may be so bad that we may not even strive to find and prove an optimal solution. Instead, our objective may be to compute high quality solutions as quickly as possible.

Finally, in real applications, we may expect that, when a backtracking algorithm finds a new upper bound, there may be other solutions that further improve the objective and can be found quickly when investing only a little more search. Classic branch-and-bound algorithms (to which we will refer as *B+B*), where the current upper bound on the objective is based on the best solution found so far, benefit from such a clustering of good solutions. Note that branch-and-bound can also be parametrized to improve on upper bounds more aggressively. For example, when only an approximately optimal solution is sought, we can set the new upper bound to $(1 - \varepsilon)u$ where $\varepsilon > 0$ and u is the value of the best solution found. Or, following an idea presented in [190], one could set the upper bound for pruning more aggressively based on empirical evidence where the optimal objective may be expected.

3.3.1 The Streeter-Smith Strategy

To address some of these issues, we follow the work from Streeter and Smith [175] who propose a dichotomic search strategy which considers (potentially incomplete) trials with a given fail-limit. They show that their parametrized strategy given in Algorithm 1 achieves an optimal competitive ratio for any fixed set of parameters $0 < \beta \leq 0.5$, $0 < \gamma < 1$, and $0 < \rho < 1$.

Query strategy $S_3(\beta, \gamma, \rho)$**:**

1. Initialize $T \leftarrow \frac{1}{\gamma}$, $l \leftarrow 1$, $u \leftarrow U$, $t_l \leftarrow \infty$, and $t_u \leftarrow -\infty$.
2. While $l < u$:
   (a) If $[l, u-1] \subseteq [t_l, t_u]$ then set $T \leftarrow \frac{T}{\gamma}$, set $t_l \leftarrow \infty$, and set $t_u \leftarrow -\infty$.
   (b) Let $u' = u - 1$. If $[l, u']$ and $[t_l, t_u]$ are disjoint (or $t_l = \infty$) then define

$$k = \begin{cases} \lfloor (1-\beta)l + \beta u' \rfloor & \text{if } (1-\rho)l > \rho(U - u') \\ \lfloor \beta l + (1-\beta)u' \rfloor & \text{otherwise;} \end{cases}$$

   else define

$$k = \begin{cases} \lfloor (1-\beta)l + \beta(t_l - 1) \rfloor & \text{if } (1-\rho)(t_l - l) \\ & > \rho(u' - t_u) \\ \lfloor (1-\beta)u' + \beta(t_u + 1) \rfloor & \text{otherwise.} \end{cases}$$

   (c) Execute the query $\langle k, T \rangle$. If the result is "yes" set $u \leftarrow k$; if the result is "no" set $l \leftarrow k + 1$; and if the result is "timeout" set $t_l \leftarrow \min\{t_l, k\}$ and set $t_u \leftarrow \max\{t_u, k\}$.

Algorithm 1: The Streeter-Smith Strategy

Assume we set $\beta = \rho = \gamma = 0.5$. Strategy S_3 then proceeds as follows: It first tries the midpoint of the given interval under some fail-limit. When the trial is inconclusive, the next trial is at $\frac{3}{4}$ of the interval and $\frac{1}{4}$ if the first is also inconclusive. This way, the search points are driven to the borders of the search interval where we expect cheaper trials. If no improved upper and lower bounds are found even for trials at the very border of the interval, the fail-limit is multiplied with $\frac{1}{\gamma}$, and the entire process is repeated. As soon as an improved upper or lower bound is found, the search interval is shrunk accordingly. Note how parameter β shifts the trial point towards the upper bound for lower values of β. Parameter ρ determines the balance how much effort we put on upper-bound rather than lower-bound improvement. In our experiments, negative trials were so costly that the best performance was always achieved by setting $\rho \leftarrow 1$. The parameter γ finally determines how quickly the fail-limit grows. In our experiments, we chose the initial fail-limit as 1000 and $\gamma \leftarrow \frac{2}{3}$. We will refer to this algorithm with the acronym *SS*.

The way how the algorithm proceeds is illustrated in Figure 3.3. The algorithm sets a fail-limit T and then maintains the current upper and lower bound as well as a time-out interval. The algorithm then performs two interleaved dichotomic searches with bias β, one in the interval $[t, t_l]$, the other in $[t_u, u]$, until the best upper and lower bounds for the given time-limit are achieved. Then, the fail-limit is increased geometrically, and two new dichotomic searches are initiated.

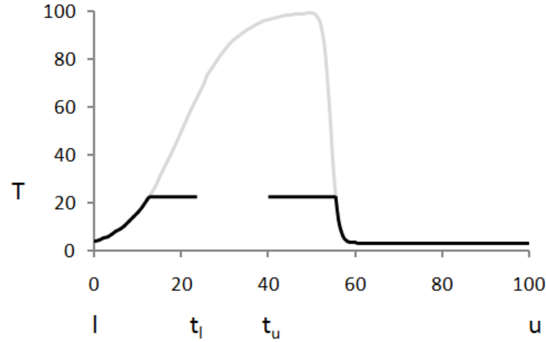


Figure 3.3: The Streeter-Smith strategy for constrained optimization on the interval [1,100].

3.3.2 Parameter Tuning based on Skewed Binary Search Protocols

While the Streeter-Smith strategy exploits a black-box feasibility solver, the specific solvers that we use for constraint satisfaction are known to benefit from randomization and restarts. Therefore, in a variant of algorithm SS, we choose to set the fail-limits in a more continuous fashion than in the Streeter-Smith strategy: After each inconclusive trial, we update the fail-limit linearly to $1000(t+1)$, where t is the number of the last trial that was inconclusive.

With respect to the fact that a backtrack-search may actually yield feasible and potentially improving solutions near a new solution that has been found, we also propose not to stop the search in case of a positive trial. Instead, we choose the next trial point and use this upper bound to prune the search from then on. When we find a new improving solution, we again set the new upper bound aggressively. If we prove unsatisfiability of the new trial or end the search at the initially given fail-limit, we continue in accordance to S_3. We will refer to this algorithm with the acronym *SS-lc*.

We observe that the interleaved searches for the best achievable upper and lower bound under some fail-limit depicted in Figure 3.3 resemble our cost-model from Figure 3.1. Based on our theoretical study of this cost-model, we are now in a good position to exploit our dichotomic search protocol to tune the parameter β which we propose to choose dynamically for every trial rather than treating it as fixed. Our modified Streeter-Smith strategy works as follows: Whenever we find an improving upper bound, we record how many fails it took within the current restart to produce the new upper bound. Based on these numbers, we keep track of the current average number of failures that it takes to compute a new upper bound. Then, we set the bias c to the ratio of the current fail-limit and this running average, as we expect the search for an improved upper bound to take the running average while a negative trial incurs at most the costs of the current fail-limit.

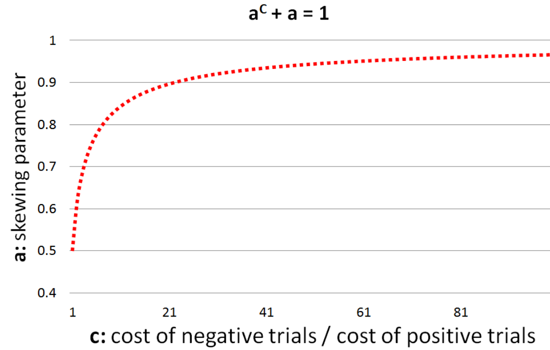


Figure 3.4: The progress of the skewing parameter, $a \in [0.5, 1)$, with respect to how costly the negative trials are compared to positive trials.

Of course, for our bias $c \geq 1$, we could compute $a \in [0.5, 1)$ online. The algorithm will be faster, however, when we pre-compute the corresponding a-values for realistic values of c, say for all natural numbers lower than 1000. In our implementation, we pre-computed values for a corresponding to c which grows exponentially starting at 1 by setting $c_{t+1} := c_t(1 + \varepsilon)$ for some small $\varepsilon > 0$. For a concrete c, we then interpolate the value for a. The parameter β is then dynamically set to $\beta \leftarrow 1 - a$. In Figure 3.4 we present the progress of the skewing parameter, a, as the negative trials become more and more costly. Notice that when cost of negative and positive trials are equal, the skewing parameter is set to 0.5.

Depending on whether we use a specific β or our skewed protocol $\beta = 1 - a$ we refer to this variation of the Streeter-Smith strategy with the acronym *SS-lc* or *SS-lc-skewed*, respectively. The latter is outlined in Algorithm 2: Given a search interval [l, u], as well as an increment unit δ to update the successive fail-limits T, the average number of failures to compute a new upper bound, avg, is initialized to 1 and the trial point k, is determined by the skewing parameter, $\beta \leftarrow 1 - a[T/avg]$, where $a[T/avg]$ gives the skewing parameter a for bias T/avg. The algorithm performs a search with fail-limit T', and returns the number of failures along with a new upper bound, $bestSol$, if a solution is ever found. If the search for an improving solution is successful, we decrease the upper bound u, increase the number of successful trials s as well as the total number of failures f, and reset the timeout flag. Then, the backtrack search is continued with the updated values of β, k and T'. If the search proves that no solution with costs lower or equal k exists, we increase the lower bound l, and the fail-limit and reset the time-out flag. If the query result is "timeout", the timeout flag is set to $true$, a temporary lower bound l' is set to k, and the fail-limit is increased. During the search, if the temporary lower bound meets the upper bound, we reset the timeout flag and restart the search from the lower bound l with a linearly increased fail-limit T. This entire process is repeated until the search interval is consumed. To facilitate the presentation, we only show the modified upper bound improvement here. Just as in the Streeter-Smith strategy, we can of course interleave the while-loop in step (2) with another skewed

Query strategy $SS - lc - skewed(l, u, \delta)$

1. Initialize $f \leftarrow 0$, $s \leftarrow 0$, $avg \leftarrow 1, T \leftarrow \delta, T' \leftarrow T, timeout \leftarrow false, l' \leftarrow l.$
2. While $l < u$:
   (a) Let $u' \leftarrow u - 1$
   (b) If $timeout = true$ and $l' \geq u'$, then set $timeout \leftarrow false$ and $l' \leftarrow l$, $T \leftarrow T + \delta, T' \leftarrow T$
   (c) Let $\beta \leftarrow 1 - a[T/avg]$
   If $timeout = true$, then set $k \leftarrow l' + (u' - l') * \beta$ else set $k \leftarrow l + (u' - l) * \beta$
   (d) Execute a limited randomized backtrack search with parameters $\langle in : k, in : T', out : failures, out : bestSol \rangle$.
   (e) i. If the result is "yes"
   Set $u \leftarrow bestSol$, $s \leftarrow s + 1$, $f \leftarrow f + failures, T' \leftarrow T - failures,$ $avg \leftarrow f/s$, $\beta \leftarrow 1 - a[T/avg]$, $k \leftarrow l + (u - l) * \beta, timeout \leftarrow false$ and $l' \leftarrow l$. Continue the latest backtrack search with parameters $\langle in : k, in : T', out : failures, out : bestSol \rangle$. Go back to (e)
   ii. If the result is "no"
   Set $l \leftarrow k + 1, T \leftarrow T + \delta, T' \leftarrow T, timeout \leftarrow false$ and $l' \leftarrow l$.
   iii. If the result is "timeout"
   Set $l' \leftarrow k, T \leftarrow T + \delta$ and $T' \leftarrow T, timeout \leftarrow true$.

Algorithm 2: Skewed Restarted Search

search that aims at increasing the lower bound quickly.

3.4 Numerical Results

In [20], skewed dichotomic search has been thoroughly investigated in the context of sorting. Here, branch-prediction and cache-misses can cause a skewed search to work more efficiently than classic binary search. Experimental results show that skewing the search leads to gains in the order of around 15%. To assess the effect of skewing dichotomic search for constrained optimization, in this section we compare the three algorithms outlined above on two benchmark problems, the weighted quasigroup-completion problem and the weighted magic square problem.

Definition [Weighted Quasigroup Completion] Given a natural number $n \in \mathbb{N}$, a quasigroup Q on symbols $1, \cdots, n$ is an $n \times n$ matrix in which each of the numbers from 1 to n occurs exactly once in each row and in each column. We denote each element of Q by q_{ij}, $i, j \in \{1, 2, \cdots, n\}$. n is called the order of the quasigroup. Given profit values $p_{ij} \in \mathbb{N}$, $i, j \in \{1, 2, \cdots, n\}$, and a set of tuples $F = \{(k, i, j) \mid 1 \leq i, j, k \leq n\}$, the Weighted Quasigroup Completion problem consists in computing a quasigroup Q such that $q_{ij} = k$ for all $(k, i, j) \in F$ and the value $\min_i \sum_j p_{ij} q_{ij}$ is minimized.

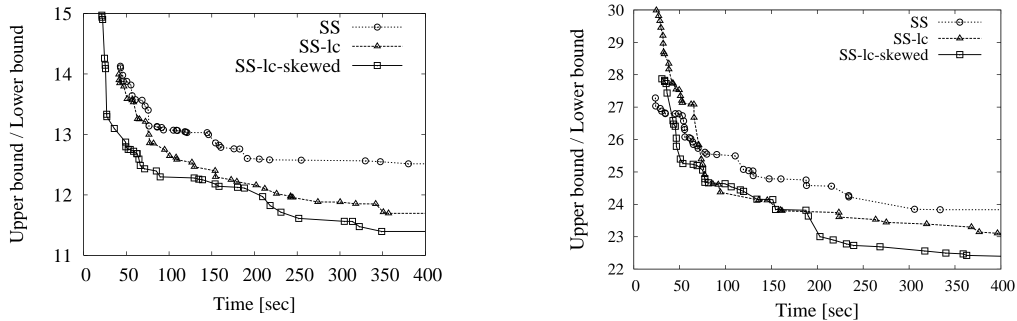


Figure 3.5: Comparison of SS, SS-lc, and SS-lc-skewed on weighted magic square problems. We show the average ratio of upper to lower bound for 20 instances with 36 (left) and 64 cells (right) and objective weights p_{ij} for each cell (i, j) chosen uniformly in [1,36] and [1,64], respectively.

Definition [Weighted Magic Square Problem] Given a natural number $n \in \mathbb{N}$, a magic square M of order n is an $n \times n$ matrix in which each of the numbers from 1 to n^2 occurs exactly once and such that the sum of all values in each row, column, and main diagonal are identical. We denote each element of M by m_{ij}, $i, j \in \{1, 2, \cdots, n\}$. Given profit values $p_{ij} \in \mathbb{N}$, $i, j \in \{1, 2, \cdots, n\}$, the Weighted Magic Square Problem consists in computing a magic square M such that the value $\min_i \sum_j p_{ij} m_{ij}$ is minimized.

From the perspective of the Constraint Programming (CP), Artificial Intelligence (AI), and Operations Research (OR) communities, combinatorial design problems as the ones given above are interesting as they are easy to state but possess rich structural properties that are also observed in real-world applications such as scheduling, timetabling, and error correcting codes. Thus, the area of combinatorial designs has been a good source of challenge problems for these research communities. In fact, the study of combinatorial design problem instances has pushed the development of new search methods both in terms of systematic and stochastic procedures. For example, the question of the existence and non-existence of certain quasigroups with intricate mathematical properties gives rise to some of the most challenging search problems in the context of automated theorem proving [193]. So-called general purpose model generation programs, used to prove theorems in finite domains, or to produce counterexamples to false conjectures, have been used to solve numerous previously open problems about the existence of quasigroups with specific mathematical properties. Considerable progress has also been made in the understanding of symmetry breaking procedures using benchmark problems based on combinatorial designs [50, 59, 86, 173]. The study of search procedures on benchmarks based on quasigroups has led to the discovery of the non-standard probability distributions that characterize complete (randomized) backtrack search methods, so-called heavy-tailed distributions [77].

For the purpose of testing dichotomic search protocols, the chosen benchmarks are interesting since even finding feasible solutions only is already hard. Moreover, it is a challenge to provide tight bounds on the

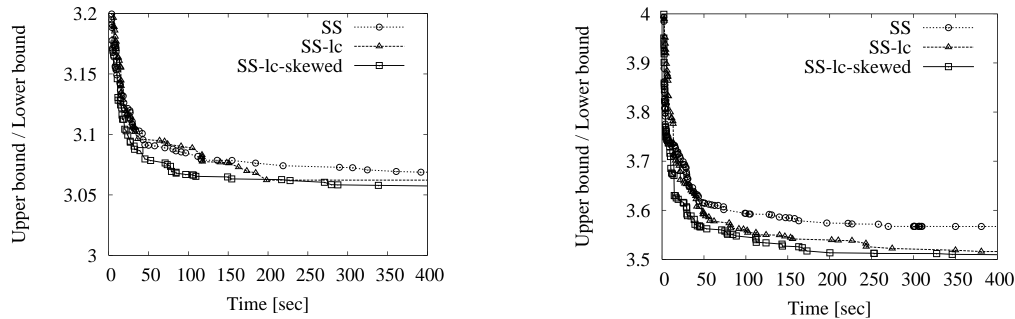


Figure 3.6: Comparison of SS, SS-lc, and SS-lc-skewed on weighted quasigroup-completion problems. We show the average ratio of upper to lower bound for 20 instances with 100 (left) and 144 cells (right) and objective weights p_{ij} for each cell (i, j) chosen uniformly in [1,100] and [1,144], respectively.

objective, which is exactly when experts usually revert to a dichotomic search to solve a problem.

Our results are illustrated in Figures 3.5 and 3.6. Experiments were run on an AMD Athlon 64 X2 Dual Core Processor 3800+ using Ilog Solver 6.5. For both problems, the CP-models used to solve particular queries are based on the obvious AllDifferent constraints. We fill the squares row by row, whereby the row to be filled next is determined by the row that currently marks the lower bound on the objective. Within a row, we pick a random variable with minimal domain and assign the lowest value in its domain first. All dichotomic algorithms perform an initial improvement phase where we try to quickly tighten the initial search interval as best as possible. Because of the difficulty to find even feasible solutions only, we did not use local search for this purpose, but a number of short, restarted tree-searches with a tight fail-limit.

The pure B+B approach without restarts often fails to provide feasible solutions within the given time-frame. Consequently, we do not show the results for this method in the figures. We believe that the inferior performance of B+B is due to the fact that it conducts one continuous search that is not restarted. Thus, it gets easily stuck in an area of the search space which does not contain feasible and improving solutions. This trap is particularly big as the CP domain-based lower bounds available to our algorithms are not of very high quality. All other algorithms avoid this problem by exploiting the benefits of a somewhat randomized branching variable selection with frequent restarts.

With respect to the remaining algorithms, we observe that SS-lc works better than the pure Streeter-Smith strategy SS. That is, we find that continuously updating the fail-limit and continuing the search with an improved upper bound after a new solution has been found is beneficial for constrained optimization. In the B+B approach, when it does find a solution, we often find that more improving solutions are found shortly afterwards. We believe that this clustering of solutions in some small subtree is caused by the algorithm having found a desirable partial assignments. Such a clustering is exploited by SS-lc by continuing the search rather than restarting directly.

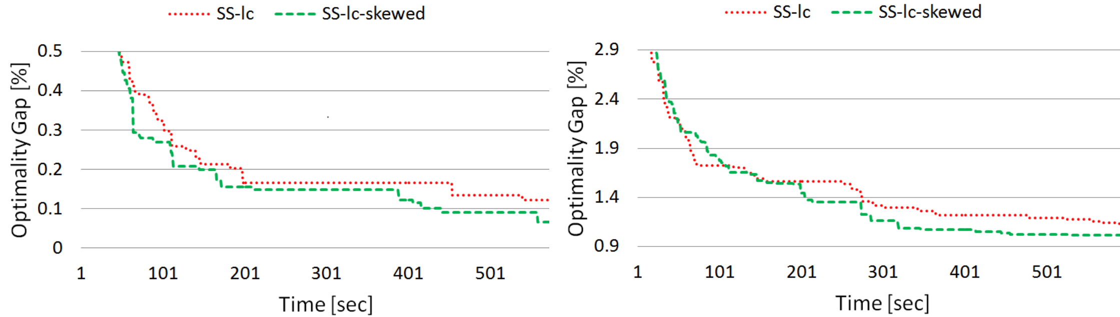


Figure 3.7: Comparison of SS-lc, and SS-lc-skewed on weighted quasigroup-completion problems. We show the development of the average percent optimality gap for 30 instances with 100 (left) and 144 cells (right) and objective weights p_{ij} for each cell (i, j) chosen uniformly in [1,100] and [1,144], respectively.

Finally, we see that SS-lc-skewed leads to an additional improvement. In this method, the fact that the Streeter-Smith strategy considers strict fail-limits allows us to get a good estimate on the search-bias c. As we had hoped, using an optimistic but not overly aggressive way to set new upper bounds based on this estimate of the bias and our theoretically optimal setting allows us to find improving solutions faster and thereby close the gap between upper and lower bound more rapidly.

3.4.1 Experiments using IBM Ilog CP Optimizer

The previous computational results were conducted in 2008. While we used a recent version of Ilog Solver at the time, we were also curious how our algorithm compares when embedded in latest commercial solvers. To this end, we reimplemented the Latin and magic square models, and the skewed dichotomic search protocol using CP Optimizer component of the IBM ILOG CPLEX Optimization Studio 12.2. This solver uses proprietary search algorithm which is based on a restarted search strategy [102]. This new set of experiments were carried out on a dual processor dual core Intel Xeon 2.8 GHz computer with 8GB of RAM. We compare the binary search with our protocol which uses the skewing technique described in this chapter. Both strategies are again embedded in the Streeter-Smith strategy.

In Figure 3.7 we plot the development of the average percent optimality gap of the current solution and the best known solution for each of 30 randomly generated weighted quasigroup completion problems of order 10 (left figure) and 12 (right figure) with and without skewing the search. Since there is a significant gap between the upper and lower bounds and we cannot find the optimum solution, we found a best known solution for each of the instances using a longer time limit.

Similar to our previous experimental results, skewing the search again helps. As the search progresses

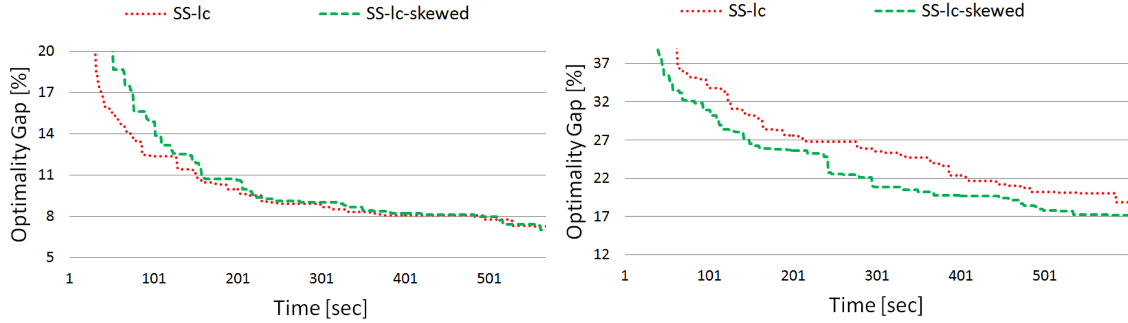


Figure 3.8: Comparison of SS-lc, and SS-lc-skewed on weighted magic square problems. We show the development of the average percent optimality gap for 30 instances with 36 (left) and 64 cells (right) and objective weights p_{ij} for each cell (i, j) chosen uniformly in [1,36] and [1,64], respectively.

and finding improving solutions becomes more and more difficult, skewing pays off and helps in finding better solutions faster.

We repeat the same experiment for the weighted magic square problem. In Figure 3.8 we plot the development of the average percent optimality gap of the current solution and the best known solution for each of 30 randomly generated weighted magic square problems of order 6 (left figure) and 8 (right figure) with and without skewing the search.

As in [167] we observe that these problems are harder to solve than weighted quasigroup completion problems. In this problem, there is more than an order of magnitude difference on how much we can close the optimality gap in this problem. That aspect aside, the results on this benchmark confirm the findings made earlier: skewing the search gives a boost in closing the optimality gap faster.

3.5 Conclusion

We studied a theoretical model for dichotomic search algorithms and devised a protocol which minimizes both expected as well as worst case performance in a skewed binary search. Furthermore, we showed that our protocol is optimal in the expected and in the worst case. Earlier experiments in the sorting domain by Brodal and Moruz had already shown practical gains from skewing binary search algorithms. In the context of constrained optimization, by exploiting the strategy proposed by Streeter and Smith, dichotomic search can be exploited in practice while skewing the search leads to faster improvements of the upper bound in constrained minimization.

In Part–II, we move on to a complementary approach and consider incomplete search algorithms. Our next goal is to devise simple yet efficient local search algorithms for an array of different problems.

Part II

Incomplete Search

Chapter Four

Dialectic Search

In this chapter, we introduce Hegel and Fichte's dialectic as a search meta-heuristic for constraint satisfaction and optimization. Dialectic is an appealing mental concept for local search as it tightly integrates and yet clearly marks off of one another the two most important aspects of local search algorithms, search space exploration and exploitation. We believe that this makes dialectic search easy to use for general computer scientists and non-experts in optimization. We illustrate dialectic search, its simplicity and great efficiency on problems from three different problem domains: constraint satisfaction, continuous optimization, and combinatorial optimization.

4.1 Introduction

Local search (LS) is a powerful algorithmic concept which is frequently used to tackle combinatorial problems. While originally developed for constrained optimization, beginning with the seminal work of Selman et al. [169] in the early 90ies local search algorithms have become extremely popular to solve also constraint satisfaction problems. Today, many highly efficient SAT solvers are based on local search. Recently there have also been developed general purpose constraint solvers that are based on local search [185].

The general idea of local search is easy to understand and often used by non-experts in optimization to tackle their combinatorial problems. There exists a wealth of modern hybrid LS paradigms like iterated local search (ILS) [177], very large scale neighborhood search [2] [181], or variable neighborhood search [83]. By far the most prevalent LS methods used by non-experts are simulated annealing [26, 118, 134] and tabu search [68, 69]. We provided descriptions of some prominent local search algorithms in the Background section.

Simulated annealing (SA) is inspired by the physical annealing process in metallurgy. The method starts out by performing a random walk as almost all randomly generated neighbors are accepted in the beginning. It then smoothly transitions more and more into a hill-climbing heuristic when neighbors are more and more unlikely to be accepted the more they degrade the solution quality. In tabu search (TS) we move to the best solution in the neighborhood of the current solution, no matter whether that neighbor improves the current solution or not. To avoid cycling, a tabu list is maintained that dynamically excludes neighbors which we may have visited already in the near past. Typically, the latter is achieved by excluding neighbors that have certain problem-specific properties which were observed recently in the search.

Both concepts are very popular with non-experts because they are easy to understand and to implement. However, to achieve a good heuristic performance for a given problem, the vanilla methods rarely work well without significant tuning and experimentation. In particular, it has often been observed that SA is able to find high-quality solutions only when the temperature is lowered very slowly or more sophisticated

neighborhoods and techniques like reheats are used. TS, on the other hand, often finds good solutions much earlier in the search than SA. However, the vague definition of the tabu-concept is difficult to handle for non-experts. If the criteria that define which neighbors are currently tabu are too broad, then many neighbors which have actually not been visited earlier are tabu. Then, so-called aspiration criteria need to be introduced to override the tabu list. Moreover, the tabu tenure is of great practical importance and difficult to tune. There exist sophisticated methods to handle this problem like reactive TS [6] which dynamically adapts the length of the tabu list and other techniques such as strategic oscillation or ejection chaining.

We argue that these techniques outside the core methods are too involved for non-experts and that there is a need for a simple method that is easy to handle for anyone with a general background in constraints. The objective of this work is to provide such a meta-heuristic which, by design, draws the user's attention to the most important aspects of any efficient local search procedure. To this end, in the next section we introduce *dialectic search*. In the sections thereafter, we provide empirical evidence that demonstrates the effectiveness of the general approach on different problems from highly different problem domains: constraint satisfaction, continuous optimization, and discrete optimization.

4.2 Dialectic Search

Without being able to make any assumptions about the search landscape, there is no way to extrapolate search experience and any unexplored search point is as good as any other. Only when we observe statistical features of the landscape which are common to many problem instances we may be able to use our search experience as leverage to predict where we may find improving solutions. The most commonly observed and exploited statistical feature is the correlation of fitness and distance [105]. It gives us a justification for intensifying the search around previously observed high quality solutions.

While the introduction of a search bias based on predictions where improving solutions may be found is the basis of any improvement over random search, it raises the problem that we need to introduce a second force which prevents us from investigating only a very small portion of the search space. This is an inherent problem of local search as the method does not allow us to memorize, in a compact way, all previously visited parts of the search space. In artificial intelligence, the dilemma of having to balance the wish for improving solutions with the need to diversify the search is known as the exploitation-exploration trade-off (EET). It has been the subject of many practical experiments as well as theoretical studies, for example on bandit problems [130].

SA and TS address the EET in very different ways. SA explores a lot in the beginning and then shifts more and more towards exploitation by lowering the acceptance rate of worsening neighbors. TS, on the

other hand, mixes exploitation and exploration in every step by moving to the best neighbor which is not tabu. The often extremely good performance of TS indicates that binding exploration and exploitation steps more tightly together is beneficial. However, the idea to mix exploration and exploitation in the same local search step is arguably what makes TS so opaque to the non-expert and what causes the practical problems with defining the tabu criteria, tabu tenure, aspiration criteria, etc.

4.2.1 A Meta-Heuristic Inspired by Philosophy

We find an LS paradigm where exploration and exploitation are tightly connected yet clearly separated from each other in philosophy: Hegel and Fichte's Dialectic [56, 87]. Their concept of intellectual discovery works as follows: The current model is called the *thesis*. Based on it, we formulate an *antithesis* which negates (parts of) the thesis. Finally, we merge thesis and antithesis to achieve the *synthesis*. The merge is guided by the principle of *Aufhebung*. The latter is German and has a threefold meaning: First, that parts of the thesis and the antithesis are preserved ("aufheben" in the sense of "bewahren"). Second, that certain parts of thesis and antithesis are annihilated ("aufheben" in the sense of "ausloeschen"). And third, that the synthesis is better than thesis and antithesis ("aufheben" in the sense of "aufwerten"). The synthesis then becomes the new thesis and the process is iterated.

Analyzing Hegel and Fichte's dialectic, we find that it strikes an appealing balance between exploration and exploitation. In essence, the formulation of an antithesis enforces search space exploration, while the optimization of thesis and antithesis allows us to exploit and improve. Furthermore, while in each step both exploration and exploitation play their part, they are clearly marked off of one another and can be addressed separately. We argue that this last aspect makes dialectic search very easy to handle.

4.2.2 Dialectic Search

We outline the dialectic search meta-heuristic in Algorithm 3. After initializing the search with a first solution, we first improve it by running a greedy improvement heuristic. We initialize a global counter which we use to terminate the search after a fixed number 'GLOBALLIMIT' of global iterations.

In each global iteration, we perform local dialectic steps, whereby the quality of the resulting solution of each such local step is guaranteed not to degrade. Again, a counter 'local' is initialized which counts the number of steps in which we did not improve the objective.

In each dialectic step, we first derive an antithesis from the thesis, which is immediately improved greedily. We assume that the way how the antithesis is generated is randomized. The synthesis is then generated by merging thesis and antithesis in a profitable way, very much like a cross-over operator in genetic algorithms. Here we assume that 'Merge' returns a solution which is different from the thesis, but may coincide

```
Dialectic Search
thesis ← InitSolution()
thesis ← GreedyImprovement(thesis)
global ← 0
bestSolution ← thesis
bestValue ← Objective(bestSolution)
while global++<GLOBALLIMIT do
  local ← 0
  while local++<LOCALLIMIT do
    antithesis ← GreedyImprovement(Modify(thesis))
    synthesis ← Merge(thesis,antithesis)
    synthesis ← GreedyImprovement(synthesis)
    thesisValue ← Objective(thesis)
    synthesisValue ← Objective(synthesis)
    if thesisValue<synthesisValue then
      goto Line 9
    end if
    if synthesisValue<bestValue then
      bestSolution ← synthesis
      bestValue ← synthesisValue
    end if
    if synthesisValue<thesisValue then
      local ← 0
    end if
    thesis ← synthesis
  end while
  thesis ← antithesis
end while
return bestSolution
```

Algorithm 3: Dialectic Search.

```
Merge (thesis, antithesis)
bestValue ← INFINITY
S ← {i | thesis[i] ≠ antithesis[i]}
while S ≠ ∅ do
  bestMoveValue ← INFINITY
  for all i ∈ S do
    margin ← SwitchMargin(thesis,antithesis,i)
    if margin < bestMoveValue then
      bestMoveValue ← margin, bestMove ← i
    end if
  end for
  S ← S \ {bestMove}
  thesis[bestMove] ← antithesis[bestMove]
  thesisValue ← thesisValue−bestMoveValue
  if thesisValue ≤ bestValue then
    synthesis ← thesis
    bestValue ← thesisValue
  end if
end while
return synthesis
```

Algorithm 4: A Procedure to Compute the Synthesis.

with the antithesis. In case that the greedily improved synthesis is actually worse than the thesis, we return to the beginning of the loop and try improving the (old) thesis again by trying a new antithesis. In case that the synthesis improves the best solution (bestSolution) ever seen, we update the latter. If the synthesis at least improves the thesis, the no-improvement counter 'local' is reset to zero. Then, the synthesis becomes the new thesis.

Finally, when the number of non-improving local improvement steps is exceeded, we make the last antithesis the new thesis and start over with the next global step.

In short, the dialectic search algorithm can be summarized as follows: For a given assignment (the thesis), it greedily improves it. Then it tries to improve the solution further by generating randomized modifications (an antithesis) of the current assignment, greedily improving it, and then combining the two assignments to form a new assignment, which is also greedily improved (the synthesis). If this new assignment is at least as good, it is considered the new current assignment. If this process does not result in improvements for a while, then the search moves to the modified assignment and continues searching from there.

As any meta-heuristic, the general outline of dialectic search that we gave above leaves certain steps open. In genetic algorithms, for example, we need to define mutation and cross-over operators. In dialectic search, we need to specify how the thesis is transformed into an antithesis, how an assignment is greedily

improved, and how thesis and antithesis are combined to form the synthesis. These functions must be defined for each problem individually. The contribution of dialectic search is that it manages the balance between exploitation and exploration, which is arguably the hardest part when devising a new local search procedure. With dialectic search, the user can focus on both tasks separately. When defining how antitheses are formed (function 'Modify'), the task is pure search space exploration. When improving a solution greedily (function 'GreedyImprovement'), the task is pure exploitation. The rule of thumb is that the antithesis is a randomized perturbation of parts of the thesis and the greedy improvement consists in moving to the best neighbor until a local minimum is reached.

Only when the synthesis is computed ('Merge'), both exploration and exploitation play a role as we would obviously like to find a very good combination of thesis and antithesis. In Algorithm 4 we give a function for computing the synthesis from two given assignments, the thesis and the antithesis. The procedure works iteratively. In each step we consider the variables on which thesis and antithesis differ and by what margin the objective changes when a variable in the thesis is re-assigned to the corresponding value in the antithesis. We perform the best change and iterate until we reach the antithesis. Like this, we generate a path from thesis to antithesis, and we return as synthesis the best solution on the path.

The idea to merge thesis and antithesis is well-founded by the empirical finding that optimization problems often exhibit a correlation between the fitness of local optima and their average distance to each other, i.e., a "big valley" structure [17]. The particular Algorithm 4 is inspired by the path relinking technique [71] and represents, of course, only one possible way of merging thesis and antithesis. Depending on the background of the reader, the function presented may also be viewed as a kind of tabu search as variables which have already been assigned their target value are no longer allowed to change their value. Another way to look at the problem of generating the synthesis is to view it as an optimization problem itself, where the task is to find the best combination of thesis and antithesis. Thus, dialectic search is implicitly related to iterated local search [177], variable neighborhood search [83], and very large scale neighborhood search [2] [181].

4.3 Constraint Satisfaction

We first test dialectic search on problems from the constraint satisfaction domain, the Costas arrays problem (CAP) and the magic squares problem (MSP).

4.3.1 Costas Arrays

A Costas array [74] is a pattern of n marks on an $n \times n$ grid, one mark per row and one per column, in which the $n(n-1)/2$ vectors between the marks are all different. Such patterns are important as they provide a template for generating radar and sonar signals with ideal ambiguity functions [34, 60]. A model for CAP is to define an array of variables $X_1, \ldots, X_n$ which form a permutation. For each length $l \in \{1, \ldots, n-1\}$, we add $n - l$ more variables $X_1^l, \ldots, X_{n-l}^l$, whereby each of these variables is assigned the difference of $X_i - X_{i+l}$ for $i \in \{1, \ldots, n-l\}$. These additional variables form a difference triangle as shown in Figure 4.1. Each line of this difference triangle must not contain any value twice. That is, the CAP is simply a collection of AllDifferent constraints on $X_1, \ldots, X_n$ and $X_1^l \ldots, X_{n-l}^l$ for all $l \in \{1, \ldots, n-1\}$.

Costas arrays can be constructed using the generation methods based on the theory of finite fields for infinitely many n. However, there is no construction method for all n and it is, e.g., unknown whether there exists a costas array of order 32. We devise a simple dialectic search for the problem and compare with tabu search.

Objective, Initialization and Greedy Improvement

As objective, we use the sum of the square of the violations of all AllDifferent constraints in the difference triangle. Our initial costas array is a random permutation of the numbers from 1 to n. As greedy improvement heuristic, we consider pairs of variables X_i and X_j and compute the cost-delta that would result from flipping the values of the two cells. We commit the pair that would decrease the violations the most and iterate until no possible flip results in a cost improvement anymore, i.e, when we are stuck in a local minimum.

Antithesis and Synthesis

Hegel defined the antithesis as the negation of the thesis. For non-binary variables it is not uniquely defined what the negation of a variable assignment is. We interpret the negation of an assignment to mean that the

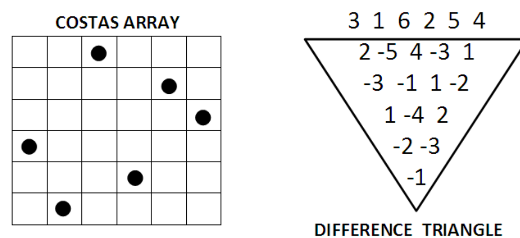


Figure 4.1: 6x6 Costas Array 316254.

	Minimum			Maximum			Std.Deviation			Average		
	TS	Dialectic		TS	Dialectic		TS	Dialectic		TS	Dialectic	
Order	Comet	Comet	C++	Comet	Comet	C++	Comet	Comet	C++	Comet	Comet	C++
13	0.03	0.01	**0**	1.24	0.64	**0.27**	0.24	0.12	**0.05**	0.25	0.13	**0.05**
14	0.03	0.01	**0**	4.9	4.25	**2.07**	0.82	0.63	**0.31**	0.96	0.55	**0.26**
15	0.04	0.05	**0.04**	22.9	15.67	**6.84**	3.45	2.81	**1.33**	3.59	3.17	**1.31**
16	0.13	0.3	**0.1**	95.8	89.84	**32.6**	19.5	17.64	**7.11**	21.8	14.6	**7.74**
17	1.03	0.72	**0.65**	741	418.93	**250**	126	93	**49.4**	114	95.3	**53.4**
18	5.49	**1.17**	4.43	2568	2539.1	**1936**	613	559	**370**	696	568	**370**

Table 4.1: Numerical Results for the Costas Array Problem. We compare tabu search and dialectic search in terms of minimum, maximum, standard deviation and average solution time in seconds over 100 runs. The tabu search algorithm is implemented in COMET platform, and the dialectic search algorithm is implemented in both COMET platform and C++.

variable is assigned a different value. For the CAP, we define an antithesis as follows. First, we determine randomly the fraction of variables that must change their value. Then, we compute an antithesis by iteratively switching the values of two cells, whereby in each step we choose the pair of cells which yields the best solution. Note that this procedure is closely related to the greedy improvement heuristic. The difference is that, in the antithesis computation, cells which have already switched values are not allowed to change their values anymore. As synthesis, we return the best solution found while moving from thesis to antithesis in this iterative way. The details of this algorithm can be found be found in the Appendix section.

Numerical Results

In Table 4.1 we compare this simple approach with the tabu search algorithm using the quadratic neighborhood which is implemented in COMET. This algorithm was shown to be highly competitive compared to specialized procedures for constraint satisfaction in [186]. All tests in this section were run on a Pentium III 733MHz machine with 512Mb RAM. Our algorithms are implemented in C++ and in COMET. C++ models were compiled using GCC 4.3, with the -O3 flag. COMET models are run using the just-in-time (-j2) compiler flag.

Even though the tabu search approach incorporates sophisticated techniques like an adaptive tabu tenure procedure, we see that the simple dialectic search algorithm is superior and outperforms TS in terms of average solution time and the minimal and maximal time needed in 100 trials for both type of implementations. Moreover, the standard deviation shows that dialectic search performs far more robustly and predictably than TS. Finally, we note that an adaptive search algorithm was proposed for generating Costas arrays

in [41] This algorithm improves over our results presented in this section, and the ones that were published in 2008 [110].

4.4 Continuous Optimization

We next apply our dialectic search algorithm to continuous optimization, the problem of finding the minimum of an n-dimensional, real-valued function over a box polytope (i.e., the only constraints are lower and upper bounds on the continuous variables). Continuous optimization problems arise in many practical application areas, like VLSI design, chemical engineering, and trajectory planning. The problem is relatively simple for functions that are differentiable and for which zero points of the derivatives can be computed. However, for higher-dimensional functions with many local minima, continuous optimization can become a challenging task. We present a simple dialectic search algorithm for the problem and compare it with simulated annealing.

Initial Solution

An initial solution is obtained by assigning to each variable a value chosen uniformly at random from the variable's domain interval.

Antithesis and Synthesis

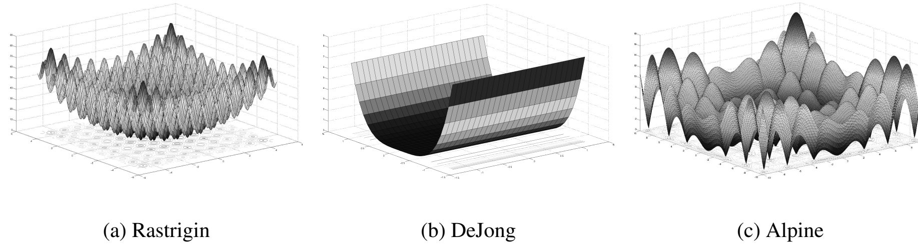
The antithesis is determined by selecting a random variable with value x^0 from the current solution and changing it to a new random value x^1. To compute the synthesis, we conduct an equi-distant walk from thesis to antithesis. At each step of the walk, we move $|x^0 - x^1|/K$ towards the antithesis. The best solution encountered during this walk is returned as the synthesis.

Numerical Results

The performance of dialectic search is examined on three well-known functions; Rastrigin, De Jong's noiseless function #4, and Alpine, with dimensions 20 and 50. DeJong's function is convex and unimodal whereas Rastrigin and Alpine functions are highly multimodal and exhibit many local minima. In Figure 4.2 we give the definition, boundary values and visualization of each function. The minimum objective value in all cases is zero. We compare our results with the SA implementation from [170] which is known to be robust, easy to

	Dialectic		SA-0.98		SA-0.99	
Function	Value	Eval.	Value	Eval.	Value	Eval.
Rastr.-20	$< \mathbf{10^{-3}}$	**208K**	24.4	3.4M	22.4	6.8M
Rastr.-50	$< \mathbf{10^{-3}}$	**818K**	87.3	8.3M	86.8	9.9M
DeJong-20	$< 10^{-3}$	**848**	$< 10^{-3}$	946	$< 10^{-3}$	946
DeJong-50	$< 10^{-3}$	3.7K	$< 10^{-3}$	**2.5K**	$< 10^{-3}$	**2.5K**
Alpine-20	$< 10^{-3}$	**86K**	$< 10^{-3}$	1M	$< 10^{-3}$	2M
Alpine-50	$< 10^{-3}$	**458K**	$< 10^{-3}$	2.9M	$< 10^{-3}$	5.8M

Table 4.2: Numerical Results for Continuous Optimization. We give average minimum value and average number of function evaluations over 250 runs for continuous function minimization with dimensions 20 and 50. SA cooling factors are set to 0.98 and 0.99.



(a) Rastrigin (b) DeJong (c) Alpine

Figure 4.2: $Rastrigin(x) = 10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i))$ where $-5.12 \leq x_i \leq 5.12$, $DeJong(x) = \sum_{i=1}^{n} ix_i^4$ where $-1.28 \leq x_i \leq 1.28$ and $Alpine(x) = \sum_{i=1}^{n} |x_i sin(x_i) + 0.1x_i|$ where $-10 \leq x_i \leq 10$.

use and applicable to complex continuous problems. Table 4.2 shows that dialectic search robustly provides very good solutions at little cost also on this problem domain.

4.5 Constrained Optimization – Set Covering

Our final evaluation of the dialectic search paradigm is on one of the most studied NP-hard combinatorial optimization problems, the set cover problem (SCP): Given a finite set $S := \{1, \ldots, m\}$ of items, and a family $F := \{S_1, \ldots, S_n \subseteq S\}$ of subsets of S, and a cost function $c : F \rightarrow \mathbb{R}^+$, the objective is to find a subset $C \subseteq F$ such that $S \subseteq \bigcup_{S_i \in C} S_i$ and $\sum_{S_i \in C} c(S_i)$ is minimized. The SCP has numerous practical applications such as crew scheduling for airlines or railway companies [24, 90, 97], location of emergency facilities [182], and production planning in various industries [188].

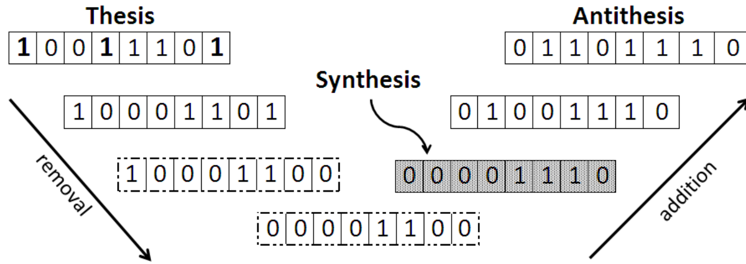


Figure 4.3: Function 'Merge' for the SCP. The decision to select a bag or not is represented as a binary variable. The bold variables in the thesis correspond to a randomized subset T of the current selection C. Dashed boxes are used to indicate solutions that do not form a cover. The synthesis is the lowest-cost cover found on the walk from thesis to antithesis.

Initial Solution and Greedy Improvement

A simple greedy construction for SCP is to pick sets one by one until a cover is found. [187] compare 7 different criteria how the next set is chosen (like the set which covers the most uncovered items, the set with least costs, the set with best cost over newly covered items ratio, and several variations of the latter). It was suggested to choose one of the criteria at random in each step of the greedy construction. Run around 30 times, this randomized approach was reported to yield good solutions, and we use this method to initialize our search. As greedy improvement heuristic, we simply remove redundant sets, if any. If there are several, we first pick a set which leaves the fewest items uncovered.

Antithesis and Synthesis

As antithesis, we pick a randomized subset T of the current selection C, whereby we choose the size of this subset randomly between one half, one third, and one quarter of the cardinality of C. T is empty first and then augmented iteratively by selecting two sets whose removal would leave the fewest items uncovered which are still covered by $C \setminus T$. One of the two sets is chosen uniformly at random and added to T. We repeat this until T has the desired size. If $A \leftarrow F \setminus T$ does not cover all items, we greedily add sets in T to A until it is a cover. A becomes our antithesis.

To obtain a synthesis, we conduct a greedy walk from the thesis to the antithesis. This walk consists of two phases. In the first phase, we remove all sets in C that are not part of A. In the second phase, we greedily select a set in A which minimizes the cost over newly covered covers items and repeat until we obtain a cover which is returned as the synthesis. Figure 4.3 illustrates such a greedy walk from thesis to antithesis.

	AvgSol		BestSol		AvgTime		TimeLimit		
Class	ITEG	Dialectic	ITEG	Dialectic	ITEG	Dialectic	ITEG	Dialectic	Speedup
a	38.78	**38.77** (0.16)	38.6	38.6	-	1.59 (1.38)	7.5	7.5	1
b	22.04	**22.00** (0.04)	22.0	22	-	0.47 (0.23)	15	2.5	6
c	43.44	43.44 (0.42)	43.0	43	-	3.00 (2.48)	10	10	1
d	25.00	**24.86** (0.15)	25.0	**24.4**	-	0.74 (0.49)	27.5	5	5.5
e	5.00	5.00 (0.00)	5.0	5.0	-	0.00 (0.00)	2.5	0.1	25
4	**38.07**	38.43 (0.28)	37.8	37.8	-	0.56 (0.49)	2.5	2.5	1
5	**34.47**	34.51 (0.35)	34.1	34.1	-	0.76 (0.56)	2.5	2.5	1
6	20.86	**20.76** (0.11)	20.8	**20.6**	-	0.24 (0.24)	15	2.5	6
nre	17.04	**17.00** (0.00)	17.0	17.0	-	0.42 (0.09)	8.5	1	8.5
nrf	10.50	**10.44** (0.49)	10.0	10	-	0.58 (0.21)	16.5	1	16.5
nrg	62.82	**62.56** (0.47)	62.0	**61.6**	-	2.85 (0.98)	6.5	5	1.3
nrh	34.78	**34.49** (0.5)	34.0	34.0	-	1.62 (0.54)	15	2.5	6

Table 4.3: Numerical Results for the Set Cover Problem. We present the average solution (standard deviation), best solution (standard deviation), average time to find the best solution, and the time limit used. The results are averaged for each benchmark class in the OR library. Hegel was run 50 times on each instance, ITEG data were taken from [132] who ran their algorithms 10 times on each instance.

Numerical Results

We compare this simple dialectic search with the iterative greedy algorithm, ITEG, from [132] and the tabu search, TS, from [141]. We consider 70 well-known benchmark instances that are available from the OR library [7]. These instances involve up to 400 items and 4000 sets. In order to compare with ITEG and TS which were developed for the uni-cost SCP, the costs of all sets are set to one. ITEG was run on a multi-user Silicon Graphics IRIX Release 6.2 IP25, 194MHz MIPS R10000 processor and TS was run on a Pentium 4 with 2.4GHz. When comparing with ITEG, we use again our Pentium III 733MHz machine and we divide the cutoff times reported for ITEG by a factor of 4 which corresponds to the SPECint95 ratio of the two machines used. For the comparison with TS, we use an AMD Athlon 64 X2 Dual Core Processor 3800 2.0 GHz machine which is slightly slower than the machine used in [141].

Tables 4.3 and 4.4 summarize the results. We report aggregate results for each of the different benchmark classes. Detailed comparison for each individual problem instance can be found in the Appendix section. It should be noted that the developers of the TS approach tuned the tabu tenure on and for each of these sets individually. Similarly, the developers of ITEG set the algorithm parameters to a suitable value for each benchmark class. In contrast, Hegel was run with one set of parameters on all instances in all classes. As we

	AvgSol		BestSol		AvgTime		
Class	TS	Dialectic	TS	Dialectic	TS	Dialectic	Speedup
a	**38.66** (0.24)	38.74 (0.16)	**38.4**	38.6	4.3 (3.78)	**1.78** (1.63)	2.4
b	22.02 (0.06)	**22.00** (0)	22	22	7.02 (6.98)	**0.49** (0.25)	14
c	43.5 (0.44)	**43.45** (0.41)	43	43	7.86 (7.16)	**2.97** (2.45)	2.6
d	25 (5.04)	**24.81** (0.12)	24.8	**24.4**	14.4 (14.4)	**1.07** (0.77)	13.4
e	5 (0)	5 (0)	5	5	0 (0)	0 (0)	0
4	**37.92** 0.27)	38.20 (0.30)	**37.7**	37.8	**0.67** (0.83)	1.63 (1.80)	0.4
5	34.36 (0.35)	**34.28** (0.15)	34.1	34.1	1.87 (2.35)	**1.85** (1.77)	1
6	20.78 (0.06)	**20.66** (0.09)	20.6	20.6	**0.26** (0.54)	0.72 (0.69)	0.3
nre	17.14 (0.3)	**16.98** (0.06)	17	**16.6**	5.94 (11.3)	**0.50** (0.46)	11.7
nrf	10.62 (0.5)	**10** (0)	10	10	31.4 (61.96)	**1.31** (0.90)	23.8
nrg	62.7 (0.6)	**62.25** (0.47)	61.8	**61.2**	32.0 (32.3)	**4.33** (2.28)	7.3
nrh	34.88 (0.44)	**34.03** (0.19)	34	**33.8**	22.4 (57.5)	**3.49** (2.20)	6.4

Table 4.4: Numerical Results for the Set Cover Problem. We present the average runtime (standard deviation) in seconds for finding the best solution in each run, as well as the average solution quality and the best solution quality. Results are averaged for all instances in each benchmark class in the OR library. Hegel was run 50 times on each instance and TS data were taken from [141] who ran their algorithms 10 times on each instance.

can see, Hegel provides high quality solutions very quickly. With the exception of classes '4' and '5' where it performs slightly worse on average, Hegel produces equally good or better results than ITEG in sometimes substantially less time. In terms of the best solutions found over the different runs, when computing the average for each class, Hegel always performs as good or better than ITEG.

Comparing with TS, Hegel is performing slightly worse on classes '4' and 'a' and outperforms TS in terms of solution quality otherwise, at times quite substantially (see classes 'nrf' and 'nrh'). Moreover, Hegel always finds the best solution earlier, leading to speed-ups of up to 23.

Finally, in terms of individual instances, Hegel found formerly unknown improving solutions on four instances (d4(24), nre1(16), nrg3(61), nrg5(61)), that is over 5% of all instances in one of the best studied benchmark sets in OR.

4.6 Boosting the Performance

When developing a new heuristic algorithm, it is often the case that we face the problem of choice. There may be multiple exploration strategies, different types of exploitation mechanisms, or a multitude of neighborhoods to choose from. The task of making these design choices is known as algorithm configuration. This is true for dialectic search presented in this chapter. In numerical results presented in the previous section, for each of the problem classes, we configured a dialectic search algorithm whereby we used the same principles of how to define an antithesis and how to merge the thesis and antithesis in order to obtain a synthesis. That is, our algorithms performed well across different benchmarks without undergoing any sophisticated tuning or parameterization. While this was a positive result, we are also curious how much we could boost the performance of dialectic search when special mechanisms were used.

In [107] we presented a method for instance specific algorithm configuration (ISAC). It is based on the integration of the automatic algorithm configuration system GGA [66] and the stochastic off-line programming paradigm [131]. We will not go into the details of this framework here but instead provide a high-level overview.

4.6.1 Instance Specific Algorithm Configuration

Briefly, the ISAC configurator is provided with a solver with categorical, ordinal, and/or continuous parameters, a training benchmark set of input instances for that solver, and an algorithm that computes a feature vector that characterizes any given input instance. The framework then works in two phases. In the learning phase, it clusters the instances in the training set based on their feature vectors and then, for each cluster, finds the best potential configuration of the solver at hand. Next, in the testing phase, it provides high quality parameter settings for any new input instance.

In the following, we describe the benchmarks instances used, the solvers that are provided to the ISAC framework, the features that were used to cluster the instances and to assign a cluster to a new instance, and finally, the performance gains achieved over the default solvers.

Benchmark: We now evaluate the performance gains that can be achieved using the ISAC framework. For this experiment, we again use the Set Covering problem. There exists up to around 70 instances in the OR library instances for this problem, which is not large enough to be used for the purpose of clustering and tuning. We instead use a highly diverse set of randomly generated set covering instances that was introduced in [131]. These instances involve up to 100 items and 10.000 sets. We pre-compute the optimal values of these instances. The final data set comprises 200 training instances and 200 test instances.

Solvers: We consider the two set covering solvers from the previous section; our dialectic search algorithm;

Hegel, and TS from [141]. We made some modifications in the TS solver in order to expose it to our parameter tuner, and we introduced new choices in this solver that were was previously switched off during the development phase as a design choice.[1] When tuning TS which is designed to solve unicost SCP, we set the cost of each set uniformly to 1 to achieve unicost instances.

Instance Features: The generation of a feature vector for each SCP instance was done according to the process outlined in [131]. This process first computes the following statistical information about an instance:

- The normalized cost vector $c' \in [1, 100]^m$.
- The vector of bag densities $(|S_i|/n)_{i=1\ldots m}$.
- The vector of item costs $(\sum_{i,j \in S_i} c'_i)_{j=1\ldots n}$.
- The vector of item coverings $(|\{i \mid j \in S_i\}|/m)_{j=1\ldots n}$.
- The vector of costs over density $(c'_i/|S_i|)_{i=1\ldots m}$.
- The vector of costs over square density $(c'_i/|S_i|^2)_{i=1\ldots m}$.
- The vector of costs over $k \log k$-density $(\frac{c'_i}{(|S_i| \log |S_i|)})_{i=1\ldots m}$.
- The vector of root-costs over square density $(\sqrt{c'_i}/|S_i|^2)_{i=1\ldots m}$.

The final feature vector is then formed by computing the maxima, minima, averages, and standard deviations of all these vectors.

Numerical Results: We now evaluate the performance gains on two efficient local search SCP solvers; Hegel and TS. Experiments were run on dual core Intel Xeon 2.8 Ghz processors with 8GB of RAM, whereby we used a timeout for Hegel and TS of 10 seconds for training and testing. For both solvers we measure the time until they have found a set covering solution which is within 10% of optimal.

In Table 4.5, we compare the default configuration of the solvers, the instance-oblivious configuration obtained by GGA, and the instance-specifically tuned versions provided by ISAC. We present the average runtime in seconds, and the average slow down per instance when comparing each solver with the ISAC version.

We first observe that the default configuration of both solvers can be improved significantly by automatic parameter tuning. For solver TS, we measure an average time of 2.18 seconds for ISAC-TS, 3.33 seconds for GGA-TS, and 3.44 seconds for default TS. That is, instance-oblivious parameters run 50 % slower than instance-specific parameters. Notice however that, the instance-oblivious parameters do not results in any

[1] We would like to thank N. Musliu for providing us the source code of TS solver for this experiment!

Solver		Avg. Run Time		Avg. Slow Down	
		Train	Test	Train	Test
TS	Default	2.79	3.44	1.49	1.7
	GGA	2.58	3.33	1.35	1.62
	ISAC	1.99	2.18	1	1
Hegel	Default	3.04	3.17	2.2	2.1
	GGA	1.58	1.98	1.1	1.1
	ISAC	1.45	1.94	1	1

Table 4.5: Comparison of the default, the instance-oblivious parameters provided by GGA, and the instance-aware parameters provided by SOP for Hegel and TS. We present the average run time in seconds, and the average degradation per instance when using the default or GGA parameters instead of ISAC.

significant improvement over the default solvers, as TS is a solver that is already tuned. When we compare the default configurations Hegel is again faster than TS on this diverse set of instances as well. As Hegel did not go into any tuning previously, the default version runs more than 60 % slower than ISAC-Hegel.

It is worth noting that the variance of the runtimes for the various instances is relatively high, which is caused by the large diversity of our benchmark sets. Therefore, we also computed the average slow down of each solver when compared with the corresponding ISAC version. For this measure we find that, for an average test instance, default TS requires more than 1.7 times the time of ISAC-TS, and GGA-TS needs 1.62 times over ISAC-TS. For default Hegel an average test instance 2.1 times the time of ISAC-Hegel while GGA-Hegel only runs 10 % slower. It is interesting to notice that ISAC-Hegel only slightly improves over GGA-Hegel. This confirms our findings in the previous numerical results that Hegel runs robustly over different instance classes with one set of parameters.

Finally, we would like to mention that even highly sophisticated state-of-the-art solvers can greatly benefit from automatic parameter tuning. Depending on the solver, instance-specific parameter tuning works as well or significantly better than instance-oblivious tuning.

4.7 Conclusion

We proposed to use Hegel and Fichte's dialectic as a meta-heuristic search paradigm and demonstrated its power and effectiveness by solving four problems from three greatly different problem domains: constraint satisfaction, continuous optimization, and combinatorial optimization.

The dialectic search paradigm allowed us to devise a local search algorithm for the Costas arrays problem. Moreover, we devised a local search algorithm for the set covering problem, one of the most intensively

studied problems in the operations research literature which has been the subject of many research projects. Dialectic search outperforms the fastest algorithms from the very rich literature which were individually tuned on and for each class of benchmark problems. In contrast, our algorithm is the same for all problems from all classes, it did not undergo any sophisticated tuning, and it still provides solutions of the same or better quality in less time. We confirm this finding using the recently introduced ISAC framework. While Hegel was able to yield the best results when compared with on default setting on a set of diverse SCP instances, it was also possible to boost its performance by a factor of two when algorithm configuration was applied. More importantly, instance-specific algorithm configuration only slightly improved the performance when compared to its instance-oblivious counter part, which was another empirical indication confirmation that Hegel runs robustly over different instance classes with one set of parameters.

We conclude that Hegel and Fichte's dialectic provides an appealing framework for devising highly efficient local search algorithms for anyone working on constraints. We believe that the reason for the simplicity of use is primarily caused by the fact that dialectic search allows us to develop functions for exploitation and exploration in separation. We outlined a close relation with existing techniques, especially tabu search, iterated local search, variable neighborhood search, and very large scale neighborhood search. We showed that dialectic search represents a special case of all of these methods. Our objective was to devise a meta-heuristic which is, on one hand, general enough to be applied to a great variety of problems and which, on the other hand, is specific enough to guide the user to develop effective problem-specific methods for search space exploration and exploitation. We believe we found a search paradigm which strikes a good balance between being specific and being general in Hegel and Fichte's philosophy of dialectic.

Part III

Variable and Value Selection

Chapter Five

Incorporating Variance in Impact-Based Search

Constraint Programming (CP) is a powerful paradigm to solve combinatorial problems. It applies constraint propagation to reduce the search space and a combination of variable and value selection heuristics to guide the exploration of that search space. While it is often the case that applications of CP would adapt a problem specific search strategy, there has been significant efforts to design generic and robust search heuristics similar to those of general purpose strategies used in Mixed Integer Programming (MIP) and SAT solvers. One traditional method is based on the famous fail-first principle which favors variables with the minimum domain size [84]. In case there is more than one variable with the minimum domain size, ties can be broken based on the degree of variables [19]. Other variations include considering the ratio between the size of the domain and the degree of the variable [16], or looking at the neighborhood structure of variables [15, 172]. For value selection, minimizing the number of conflicts with neighbouring variables is a popular technique. More recently, impact-based search strategies have been studied. Different measures of impact have been proposed and they have been successfully applied for solving constraint satisfaction problems [23, 158, 192].

In this chapter, we present a simple modification to the idea of impact-based search proposed by Refalo in 2004 [158] which has been proven to be highly effective for several applications. Impacts measure the average reduction in search space due to propagation after a variable assignment has been committed. Rather than considering the mean reduction only, we consider the idea of incorporating the variance in reduction. Experimental results show that using variance can result in improved search performance.

5.1 Introduction

Impact-based search strategies give efficient variable and value ordering heuristics to solve decision problems in constraint programming [158]. This method learns information about the importance of variable and values choices. This is done by averaging the observed search space reduction due to constraint propagation after an assignment. These observed reductions are averaged and become more and more accurate as we explore the search space. It's a simple way to exploit parts of the search tree that are apparently not useful because they do not lead to a solution.

Other impact measures have been designed and subjected to experimental validation. They refine or take into account more information in order to obtain better strategies. In [192] the solution density of constraints and occurrences of values in constraints' feasible assignments are used to guide search. In [23] the measure of the impact of an assignment is based on explanations provided by the constraint programming solver. These approaches can be more effective than regular impacts on some problems.

We propose a new way to refine the classical averaging of impact observations by taking into account

the *variance* of the observations. In practice, when one needs to choose between two variables that have the same average impact, one can take into account the distribution of the observed impacts. Assuming that the two distributions have different variances, a risk-free choice will choose the variable with smaller variance, while an optimistic choice will choose the variable with larger variance.

Incorporating variance in impact based search is natural since impacts are based on taking the mean of observed domain reductions. Moreover, in practice, impact values are normally distributed [159]. Experimental validation was performed to determine the best way to use variance in practice. We present our results on quasi-group completion problems, magic squares and on the Costas array problem. Our results show that including variance can be rewarding in several cases and that it is an enhancement to be considered for impact-based search implementations.

5.2 Impact-based Search

In constraint programming we strive to find feasible solutions, and our main inference mechanism is constraint propagation. Namely, by considering the problem constraints, one at a time, we eliminate potential values for the variables involved in the constraint. We iterate this process until no one constraint alone can eliminate values from the domain of variables anymore.

If we want to avoid an explicit enumeration of all potential solutions, we must obviously rely on constraint propagation to discard most of these solution candidates *implicitly*. Therefore, the way how we partition the space needs to enable constraint propagation to function well.

There are several ways how search methods try to provide the underlying inference mechanism with the necessary "grip." To list only a few, solution-density guided search [192] finds a constraint where one variable clearly favors one value in the sense that in most assignments that obey this constraint the variable is overwhelmingly assigned to the respective value. The method branches over that variable in the hope that the constraint will fail quickly when one of the other values is assigned to it. [1]

In mathematical programming a well-known and successful technique is pseudo-cost branching. While solution-density guided search looks ahead, pseudo-cost branching keeps a running average of the change in relaxation objective value due to the branching on a variable. That is, pseudo-cost branching extrapolates the past search experience to make predictions which search partition is likely to affect the inference mechanism the most.

Impact-based search in constraint programming is following the same motivation. Lacking an objective function, [158] proposes to keep a running average of the reduction in search space that is observed after

[1] Note that this is our summary which does not align with the motivation given in [192].

committing a variable assignment $X = v$. Assume the Cartesian product of the variables' domains *before* committing the assignment has size $B \in \mathbb{N}$ and the product of all domain sizes *after* committing and propagating the assignment is $A \in \mathbb{N}$. Then, the impact of the assignment is defined as

$$I(X = v) = 1 - \frac{A}{B}.$$

The running average of these values is denoted with $\bar{I}(X = v)$.

From these values we can derive the expected search space reduction factor (ERF) for a variable. Namely, the sum of the Cartesian products of all domain sizes after committing in turn $X = v$ for all values v in the domain $D(X)$ is expected to be multiplied by

$$\mathrm{ERF}(X) = 1 - \sum_{v \in D(X)} \bar{I}(X = v).$$

In [158] it was proposed to branch over the variable with the lowest corresponding ERF: if we assume that we are searching an infeasible part of the search space, we expect that all alternative values for X must be explored. The lower the ERF, the smaller we expect the union of the remaining search spaces to be after committing assignments $X = v$ for all $v \in D(X)$. This method has since proven to work very well in various domains such as Latin square completion, magic square, and multi-knapsack problem.

5.3 Impact Variance

Obviously, when estimating the ERF by computing a running average of reductions in search space that we observe, our estimate will come with some uncertainty.

5.3.1 Variance

To assess the confidence that we have in our estimate, variance is the statistical quantity that comes to mind first. Between two variables that have the same low ERF, being risk averse clearly we would favor the one that has exhibited less variance in search space reduction. On the other hand, if we are optimistic we might want to choose a variable that offers the potential of significantly reducing the search space.

If we incorporate variance, we now have two quantities that we want to optimize. The natural question is what should be the right trade-off between both objectives. If we assumed that the ERFs of a variable are normally distributed, then

- with about **68%** probability the real reduction factor will be lower than the mean **plus the standard deviation** (i.e., the square root of the variance),
- with about **95%** probability the real reduction factor will be lower than the mean plus **two times** the standard deviation, and
- with about **99.7%** probability the real reduction factor will be lower than the mean plus **three times** the standard deviation.

Alternatively, if we take the optimistic viewpoint and value variables with larger potential more, for a normal distribution we can argue that

- with about **32%** probability the real reduction factor will be lower than the mean **minus the standard deviation**, and
- with about **5%** probability the real reduction factor will be even lower than the mean minus **two times** the standard deviation.

Even though we cannot assume that the real reduction factors will be exactly normally distributed, the trend will be the same for all distributions. We therefore propose to choose an $\alpha \in \mathbb{Q}$ (where $\alpha > 0$ means we are risk averse, and $\alpha < 0$ means "we are feeling lucky") and to compute the *adjusted reduction factor*

$$\mathrm{ARF}_\alpha(X) = \mathrm{ERF}(X) + \alpha\sqrt{\mathrm{VAR}(X)}.$$

Then, we choose as branching variable

$$X = \mathrm{argmin}_Y \mathrm{ARF}_\alpha(Y).$$

If we choose $\alpha > 0$, then we compare variables by their ability to shrink the search space which we can expect with some higher probability. On the other hand, if we choose $\alpha < 0$, we compare variables based on their potential to shrink the search space a lot.

Note that the idea to use a factor $\alpha < 0$ somewhat resembles the idea of *upper confidence trees (UCTs)* [125]. As a very high-level description, in the UCT method we probe a tree and achieve estimates of the quality of a subtree by the samples drawn from the probes over the different child nodes. The question arises which probes should be launched next. Based on a very nice theory it was proven that it pays off to optimistically consider subtrees first which combine a good current estimate and larger uncertainty.

Our situation is of course different, as each "probe" can incur a significant cost. Essentially, without nogood-learning, with each unfortunate choice of the branching variable we could multiply the minimally required search effort. Therefore, we compare risk-averse and optimistic impact-based search in an empirical

study.

5.3.2 Computing a Variance-Estimate

To implement the approach outlined above we obviously need to assess the quantity $\mathrm{VAR}(X)$. We can achieve this based on the variance that we observe for the variable assignment impacts $I(X = v)$. Since the random variable ERF is based on the sum of these random variables, if we assume that the $I(X = v)$ (for various v) are independent, then the variance of the ERF will be simply the sum of the variances of the $I(X = v)$. In other words, the variance can be estimated as

$$\mathrm{VAR}(X) = \sum_{v \in D(X)} \mathrm{VAR}(X = v).$$

All that is left to develop now is a way for estimating the variance of the $I(X = v)$. We could of course keep a history of these values and compute the variance from scratch. However, there is a much more elegant way, namely, after a new value for $I(X = v)$ or $\mathrm{ERF}(X)$ is observed, we can update the variance *online*.

This is trivial for the mean of a sequence. Given numbers $a_1, \ldots, a_{n-1}$ and their mean $\mu_{n-1} = \frac{\sum_i a_i}{n-1}$, and a new number a_n, for the new mean it obviously holds

$$\mu_n = \frac{(n-1)\mu_{n-1} + a_n}{n}.$$

A similar update rule holds for the sum of square differences $\mathrm{SD}_{n-1} = \sum_i (a_i - \mu_{n-1})^2$ [121]:

$$\mathrm{SD}_n = \mathrm{SD}_{n-1} + (a_n - \mu_n)(a_n - \mu_{n-1}).$$

Therefore, we maintain three numbers for each $I(X = v)$: The number of times n we have observed a value, the current mean μ_n, and the current sum of square differences SD_n. Then, to choose a branching variable we use the unbiased [121] variance estimate $\frac{\mathrm{SD}_n}{n-1}$.

5.4 Numerical Results

We now present empirical results demonstrating the benefits obtained by incorporating variance information as well as impacts when branching. We implemented the new heuristic in IBM Ilog Solver, and studied

a number of problems. The goal of our experiments is to compare the relative performance of these three different variable selection heuristics:

- Impact-based search (IBS).
- Impact-based search with addition of variable variance. That is, α is set to 1 or to 2, which means we favor variables with low variance.
- Impact-based search with subtraction of variable variance. That is, α is set to -1 or to -2, which means we favor variables with high variance.

To conduct a fair test of the different variable selection heuristics we use a randomized value selection strategy and perform multiple runs of the same instance with different random seeds. The initial mean and variance value are obtained by probing each value of the domain of the variable. This gives a first impact for each value. Then a second impact is computed by performing a few steps of a dichotomic search to approximate impacts as described in [158]. Thus we have enough values at the beginning to compute the impact mean and the unbiased variance. In the instances we consider here, the overhead of probing the whole variable domain is negligible compared to the solution time. All approaches ran on identical models with the DFS search implementation of IBM ILOG Solver. The experiments were run on dual processor dual core Intel Xeon 2.8 GHz computers with 8GB of RAM.

5.4.1 Quasigroup Completion

Problem Definition *A quasigroup completion problem [78] is tasked with completing an $n \times n$ partially filled matrix such that the numbers from 1 to n appear exactly once in each row and column*

Quasigroup completion problems are a well-known combinatorial problem. These problems, unlike Latin square or quasigroup with holes problems, are not necessarily satisfiable. We consider two sets of 100 instances, one set with order 40 and 640 unassigned cells ("holes"), and another one with order 50 and 1250 unassigned cells. They were generated randomly using a standard tool provided by the authors of [78]. We used depth-first search to solve the problems and four standard deviation factors $\alpha = \{-2, -1, 1, 2\}$. When setting α to 0, the strategy is simply the standard impact-based strategy. We perform 10 runs for each instance with as many different random seeds. The time limit is 2,000 seconds. We report the average running time in seconds and the number of instance solved (the maximum is 1,000 for each set).

Our evaluation is presented in Table 5.1. We can see that an optimistic approach outperforms both the classical and risk-averse impact-based search on the instances. The optimistic strategy solves considerably more instances in substantially less time. On the other hand, risk-averse strategy marks the worse performance: it is slower and solves the least number of the instances.

α value	-2	-1	0 (IBS)	1	2
Time	**302**	320	336	374	408
# Success	**596**	555	523	439	355

α value	-2	-1	0 (IBS)	1	2
Time	**166**	187	184	247	333
# Success	**864**	825	805	739	567

Table 5.1: The Quasigroup Completion Problem. We compare impact based search with and without incorporating standard deviation. We consider quasigroups with Order = 40 and Holes = 640 (the table on the top), and with Order = 50 and Holes = 1250 (the table on the bottom). We present the average running time in seconds, and the number of instances solved. We considered 100 instances for each order, and ran each instance with 10 different seeds. The time limit is set to 2,000 seconds. The numbers in bold denote the best in each row.

5.4.2 Magic Squares

Problem Definition: *A magic square [140] of order n is an $n \times n$ square that contains all numbers from 1 to n^2 such that each row, column and both main diagonals add up to the "magic sum" $n(n^2 - 1)/2$.*

Magic squares are a much studied problem in the domain of combinatorial solvers. Although polynomial-time construction methods exist for creating magic squares(e.g., general techniques of constructing even and odd squares of order n is given in [122]) the problem poses a challenge for constraint programming based approaches. The current best systematic approach was presented in [75] and can only construct magic squares of orders up to 18 efficiently.

We again evaluate the performance of impact based search (when α is 0) and impact based search incorporated with standard deviation using factors $\alpha = \{-2, -1, 1, 2\}$. We consider magic squares of orders between 5 and 16. We use 50 different seeds for each order. The time limit is set to 2,000 seconds. We compare the average runtime and the average number of successful trials.

Table 5.2 summarizes our results for the magic square problem. We observe that risk-optimistic approaches and impact-based search perform similarly, while the best performance is achieved when α is set to -2, i.e., when the most risk-optimistic strategy is used. On the contrary, risk-averse approaches depict an inferior performance in terms of both running time and number of successful trials.

α value	-2	-1	0 (IBS)	1	2
Time	**680**	691	686	705	735
# Success	**34.3**	34.2	34.2	33.8	33

Table 5.2: The Magic Square Problem. We compare impact based search with and without incorporating standard deviation. We present the average runtime in seconds, and the average number of successful trials to find magic squares orders between 5 and 16. We considered 50 instance for each order, and results are averaged over all orders. The time limit is set to 2,000 seconds. The numbers in bold denote the best in each row.

5.4.3 Costas Array

Problem Definition: *A Costas array [74] is a pattern of n marks on a $n \times n$ grid, such that each column or row contains only one mark, and all of the $n(n-1)/2$ vectors between the marks are all different.*

Costas arrays are a mathematical structure that is studied in a number of domains. It is a combinatorial structure with links to number theory, and is used to provide a template for generating radar and sonar signals with ideal ambiguity functions [34, 60].

We consider Costas arrays of orders between 10 and 19, and evaluate impact-based search, and impact-based search with standard deviation incorporated using factors $\alpha = \{-2, -1, 1, 2\}$. We use 50 different seeds for each order. The time limit is set to 2,000 seconds. We compare the average running time, and the average number of successful trials.

Our results are presented in Table 5.3. While the best performance is achieved with an optimistic approach, it is better than impact-based search with only a small margin. The risk-averse approaches again perform worse than other strategies.

Overall, we tried to determine the best way to use variance information in practice on three different constraint satisfaction problems. We showed that including variance in a risk-optimistic setting can improve the search performance in several cases, and it is never worse than original impact-based search or its counterpart. We attribute the improved performance of the optimistic strategy to its ability to select variables that have high potential to reduce the search space. This is in accordance with our observations in Chapter 4 and in [110]. Both magic squares and Costas array problems react very well to aggressive exploitation strategies, which can take place as using an optimistic strategy in systematic search, or using greedy improvements in local search.

We restate the fact that the goal of our experiments is not to achieve state-of-the-art results for all these problems, but rather to compare the relative performance of different impact-based search heuristics. For

α value	-2	-1	0 (IBS)	1	2
Time	224	**218**	220	234	234
# Success	46.8	**46.9**	46.8	46.4	46.8

Table 5.3: The Costas Array Problem. We compare impact based search with and without incorporating standard deviation. We present the average runtime in seconds, and the average number of successful trials to find Costas arrays of orders between 10 and 19. We considered 50 instance for each order, and results are averaged over all orders. The time limit is set to 2,000 seconds. The numbers in bold denote the best in each row.

completeness, we also provide experimental results on these problems using the well-known *fail-first* strategy, also known as minimum domain size heuristic. These results are presented in 5.4. We note this heuristic is significantly better on the instances we considered for the quasigroup completion problem compared to impact-based search strategies. Conversely, it performs worse on the magic squares and Costas array problems. Finally, we would like to mention that restart strategies were shown to improve performance on these constraint satisfaction problems [158]. Hence, it should be further investigated whether variance information would be useful in restarted search protocols.

5.5 Conclusion

We presented a new search heuristic which is based on a simple modification to impact-based search. The modification is to take variance information into account as well as impact values when selecting a branching variable. We considered a risk-averse and an optimistic version of impact-based search with different coefficients, and provided experimental results that compare their relative performance on three different problems. Our findings suggest that an optimistic approach can improve the search performance. Hence it has potential to be a useful search heuristic, and, in general, variance information is an easy enhancement to be considered for impact-based search implementations.

minDomain	QCP–40–640	QCP–50–1250
Time	265	52
# Success	647	929

minDomain	Magic Square	Costas Array
Time	799	307
# Success	31.17	44.9

Table 5.4: Numerical Results using the Minimum Domain Size Heuristic. We present the average running time in seconds, and the number of instances solved when minimum domain size search strategy is used. In the table on the top, we show results on quasigroup completion problems with Order = 40 and Holes = 640 and with Order = 50 and Holes = 1250. We used 100 instances for each order and ran each instance with 10 different seeds, the same setting as in the experiments conducted for the impact-based search. In the table on the bottom, we show results on orders between 5 and 16 for the magic squares problem and orders between 10 and 19 for the Costas array problem. We considered 50 instance for each order, again, used the same settings as impact-based search experiments. Similarly, the time limit is set to 2,000 seconds.

Part IV

Interplay Between Search and Inference

A Tale of Two Principles

So far we have distinguished between two fundamentally different principles for solving combinatorial satisfaction and optimization problems. The first principle was intelligent reasoning about (sub)problems, namely inference. Inference is comprised of techniques like relaxation and pruning, variable fixing, bound strengthening, and constraint filtering and propagation. And the second principle was search which is about exploring different parts of potential solutions. The search can be conducted in different fashions. In systematic solvers, the space of potential solutions is partitioned and the different parts are searched in an order. This is in contrast to non-systematic solvers that are based on local search techniques. The main aspects of systematic solvers are how the solution space is partitioned, and in what order the different parts are to be considered. Both choices have an enormous impact on solution efficiency. In local search, the question becomes how to define neighborhoods, and how to balance the desire for improving solutions with the need to diversify the search.

While these two different principles are somewhat orthogonal to each other and we can distinguish between the two in the way we described above, search and inference work hand in hand to solve combinatorial problems as practically all successful solvers in constraint programming, satisfiability, and mathematical programming do.

At this point, we should note that it is possible to strengthen inference by computing valid inequalities and, more generally, no-good learning and redundant constraint generation, as well as automatic model reformulation. In fact, inference can be strengthened to a point where it is capable of solving combinatorial problems all by itself (consider for instance Gomory's cutting plane algorithm for general integer problems [79] or the concept of k-consistency in binary constraint programming [32, 62]). However, today's most competitive solvers complement inference with an active search for solutions.

In the case of constraint programming, the solution process consists of interleaving constraint propagation with search. Reasoning about (sub-)problems, generally in terms of global constraints, is embedded in a search environment, and is executed many times during search. Hence, it is desirable to develop efficient filtering algorithms. To that end, exploiting incrementality and leveraging the knowledge about problem structure is indispensable for both theoretical and practical performance.

In the following two chapters, we present our work on designing inference mechanisms that are specifically tailored for tree search. We first consider constraints based context-free grammars. The memory requirements of the existing filtering algorithms for context-free grammar constraints is prohibitive for tree search. We present a time and space efficient filtering algorithm for context-free grammar constraints.

We then consider binary constraint satisfaction problems. In the special structure we are interested in all sets of pairs of variables share the same relation. A well-known example of this type of constraints is the ALLDIFFERENT [160] constraint which enforces the conjunction of the same binary constraint, the not-equal constraint, for every pair of variables. We study different constraint graph structures and present filtering algorithms that outperform propagating each constraint individually in both theory and practice.

Chapter Six

Efficient Context-Free Grammar Constraints

In this chapter, we follow the line of research introduced by the constraints based on finite automata. We consider constraints based on grammars higher up in the Chomsky hierarchy. Our contribution is to devise a time- *and* space-efficient incremental arc-consistency algorithm for context-free grammars. Particularly, we show how to filter a sequence of monotonically tightening problems in cubic time and quadratic space. Experiments on a scheduling problem show orders of magnitude improvements in time and space consumption. We also show how the structure of the CYK parser could be used to direct value selection. Finally, we investigate when logic combinations of grammar constraints are tractable, show how to exploit non-constant size grammars and reorderings of languages, and study where the boundaries run between regular, context-free, and context-sensitive grammar filtering.

6.1 Introduction

A major strength of constraint programming is its ability to provide the user with high-level constraints that capture and exploit problem structure. However, this expressiveness comes at the price that the user must be aware of the constraints that are supported by a solver. One way to overcome this problem is by providing highly expressive global constraints that can be used to model a wide variety of problems and that are associated with efficient filtering algorithms. As was found in [9, 25, 116, 149], a promising avenue in this direction is the introduction of constraints that are based on formal languages, which enjoy a wide range of applicability while allowing the user to focus on the desired properties of solutions rather than having to deal for herself with the problem of constraint filtering.

The first constraints in this regard were based on automata [9, 25, 116, 149]. Especially, incremental implementations of the regular membership constraint have been shown to perform very successfully on various problems and even when used to replace custom constraints for special structures which can be expressed as regular languages. In [154, 165], algorithms were devised which perform filtering for context-free grammar constraints in polynomial time. Our focus is on practical aspects when dealing with context-free grammars. In particular, we devise an incremental algorithm which combines low memory requirements with very efficient incremental behavior. Tests on a real-world shift-scheduling problem prove the practical importance of grammar constraints and show significant speed-ups achieved by the new algorithm. We improve the performance guiding the value selection with the information provided by our propagator. We also show how context-free grammar constraints can efficiently be conjoined with linear constraints to perform cost-based filtering. We then study how logic combinations of grammar constraints can be propagated efficiently. Finally, we investigate non-constant size grammars and reorderings of languages.

6.2 Basic Concepts

We start our work by reviewing some well-known definitions from the theory of formal languages. For a full introduction, we refer the interested reader to [96]. All proofs that are omitted in this thesis can also be found there.

Alphabet and Words Given sets Z, Z_1, and Z_2, with $Z_1 Z_2$ we denote the set of all *sequences* or *strings* $z = z_1 z_2$ with $z_1 \in Z_1$ and $z_2 \in Z_2$, and we call $Z_1 Z_2$ the *concatenation* of Z_1 and Z_2. Then, for all $n \in \mathbb{N}$ we denote with Z^n the set of all sequences $z = z_1 z_2 \ldots z_n$ with $z_i \in Z$ for all $1 \leq i \leq n$. We call z a *word* of length n, and Z is called an *alphabet* or set of *letters*. The empty word has length 0 and is denoted by ε. It is the only member of Z^0. We denote the set of all words over the alphabet Z by $Z^* := \bigcup_{n \in \mathbb{N}} Z^n$. In case that we wish to exclude the empty word, we write $Z^+ := \bigcup_{n \geq 1} Z^n$.

Context-Free Grammars A *grammar* is a tuple $G = (\Sigma, N, P, S_0)$ where Σ is the alphabet, N is a finite set of *non-terminals*, $P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ is the set of *productions*, and $S_0 \in N$ is the start non-terminal. We will always assume that $N \cap \Sigma = \emptyset$. Given a grammar $G = (\Sigma, N, P, S_0)$ such that $P \subseteq N \times (N \cup \Sigma)^*$, we say that the grammar G and the language L_G are *context-free*. A context-free grammar $G = (\Sigma, N, P, S_0)$ is said to be in *Chomsky Normal Form (CNF)* if and only if for all productions $(A \to \alpha) \in P$ we have that $\alpha \in \Sigma^1 \cup N^2$. Without loss of generality, we will then assume that each literal $a \in \Sigma$ is associated with exactly one unique non-literal $A_a \in N$ such that $(B \to a) \in P$ implies that $B = A_a$ and $(A_a \to b) \in P$ implies that $a = b$.

Remark We will use the following convention: Capital letters A, B, C, D, and E denote non-terminals, lower case letters a, b, c, d, and e denote letters in Σ, Y and Z denote symbols that can either be letters or non-terminals, u, v, w, x, y, and z denote strings of letters, and α, β, and γ denote strings of letters and non-terminals. Moreover, productions (α, β) in P can also be written as $\alpha \to \beta$.

Derivation and Language

- Given a grammar $G = (\Sigma, N, P, S_0)$, we write $\alpha\beta_1\gamma \underset{G}{\Rightarrow} \alpha\beta_2\gamma$ if and only if there exists a production $(\beta_1 \to \beta_2) \in P$. We write $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m$ if and only if there exists a sequence of strings $\alpha_2, \ldots, \alpha_{m-1}$ such that $\alpha_i \underset{G}{\Rightarrow} \alpha_{i+1}$ for all $1 \leq i < m$. Then, we say that α_m can be *derived* from α_1.
- The *language given by* G is $L_G := \{w \in \Sigma^* \mid S_0 \overset{*}{\underset{G}{\Rightarrow}} w\}$.

Definition 6.2 gives a very general form of grammars which is known to be Turing machine equivalent. Consequently, reasoning about languages given by general grammars is infeasible. For example, the word problem for grammars as defined above is undecidable.

Word Problem Given a grammar $G = (\Sigma, N, P, S_0)$ and a word $w \in \Sigma^*$, the *word problem* consists in answering the question whether $w \in L_G$.

Therefore, in the theory of formal languages, more restricted forms of grammars have been defined. Chomsky introduced a hierarchy of decreasingly complex sets of languages [27]. In this hierarchy, the grammars given in Definition 6.2 are called Type-0 grammars. In the following, we define the Chomsky hierarchy of formal languages.

Type 1 – Type 3 Grammars

- Given a grammar $G = (\Sigma, N, P, S_0)$ such that for all productions $(\alpha \to \beta) \in P$ we have that β is at least as long as α, then we say that the grammar G and the language L_G are *context-sensitive*. In Chomsky's hierarchy, these grammars are known as Type-1 grammars.
- Given a grammar $G = (\Sigma, N, P, S_0)$ such that $P \subseteq N \times (N \cup \Sigma)^*$, we say that the grammar G and the language L_G are *context-free*. In Chomsky's hierarchy, these grammars are known as Type-2 grammars.
- Given a grammar $G = (\Sigma, N, P, S_0)$ such that $P \subseteq N \times (\Sigma^* N \cup \Sigma^*)$, we say that G and the language L_G are *right-linear* or *regular*. In Chomsky's hierarchy, these grammars are known as Type-3 grammars.

Remark The word problem becomes easier as the grammars become more and more restricted: For context-sensitive grammars, the problem is already decidable, but unfortunately PSPACE-complete. For context-free languages, the word problem can be answered in polynomial time. For Type-3 languages, the word problem can even be decided in time linear in the length of the given word.

For all grammars mentioned above there exists an equivalent definition based on some sort of automaton that accepts the respective language. As mentioned earlier, for Type-0 grammars, that automaton is the Turing machine. For context-sensitive languages it is a Turing machine with a linearly space-bounded tape. For context-free languages, it is the so-called push-down automaton (in essence a Turing machine with a stack rather than a tape). And for right-linear languages, it is the finite automaton (which can be viewed as a Turing machine with only one read-only input tape on which it cannot move backwards). Depending on what one tries to prove about a certain class of languages, it is convenient to be able to switch back and forth between different representations (i.e. grammars or automata). In this work, when reasoning about context-free languages, it will be most convenient to use the grammar representation. For right-linear languages, however, it is often more convenient to use the representation based on finite automata:

Finite Automaton Given a finite set Σ, a *finite automaton* A is defined as a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of *states*, Σ denotes the *alphabet* of our language, $\delta \subseteq Q \times \Sigma \times Q$ defines the *transition function*, q_0 is the *start state*, and F is the set of *final states*. A finite automaton is called *deterministic* if and only if $(q, a, p_1), (q, a, p_2) \in \delta$ implies that $p_1 = p_2$.

Accepted Language The language defined by a finite automaton A is the set $L_A := \{w = (w_1, \ldots w_n) \in \Sigma^* \mid \exists\ (p_0, \ldots, p_n) \in Q^n\ \forall\ 1 \leq i \leq n :\ (p_{i-1}, w_i, p_i) \in \delta \text{ and } p_0 = q_0, p_n \in F\}$. L_A is called a *regular*

language.

Lemma 6.2.1. *For every right-linear grammar G there exists a finite automaton A such that $L_A = L_G$, and vice versa.*

6.3 Context-Free Grammar Constraints

Within constraint programming it would be convenient to use formal languages to describe certain features that we would like our solutions to exhibit. It is worth noting here that any constraint and conjunction of constraints really defines a formal language by itself when we view the instantiations of variables $X_1, \dots, X_n$ with domains $D_1, \dots, D_n$ as forming a word in $D_1 D_2 \dots D_n$. Conversely, if we want a solution to belong to a certain formal language in this view, then we need appropriate constraints and constraint filtering algorithms that will allow us to express and solve such constraint programs efficiently. We formalize the idea by defining grammar constraints.

Grammar Constraint For a given grammar $G = (\Sigma, N, P, S_0)$ and variables $X_1, \dots, X_n$ with domains $D_1 := D(X_1), \dots, D_n := D(X_n) \subseteq \Sigma$, we say that $Grammar_G(X_1, \dots, X_n)$ is true for an instantiation $X_1 \leftarrow w_1, \dots, X_n \leftarrow w_n$ if and only if it holds that $w = w_1 \dots w_n \in L_G \cap D_1 D_2 \dots D_n$.

The idea to exploit formal grammars for constraint programming by considering regular languages has been studied before [9, 25, 116, 149]. Based on the review of our knowledge of formal languages in the previous section, we can now ask whether we can also develop efficient filtering algorithms for grammar constraints of higher-orders. Clearly, for Type-0 grammars, this is not possible, since the word problem is already undecidable. For context-sensitive languages, the word problem is PSPACE complete, which means that even checking the corresponding grammar constraint is computationally intractable.

However, for context-free languages deciding whether a given word belongs to the language can be done in polynomial time. Since their recent introduction, context-free grammar constraints have already been used successfully to model real-world problems. For instance, in [153] a shift-scheduling problem was modeled and solved efficiently by means of grammar constraints. Context-free grammars come in particularly handy when we need to look for a recursive sequence of nested objects. Consider for instance the puzzle of forming a mathematical term based on two occurrences of the numbers 3 and 8, operators +, -, *, /, and brackets (,), such that the term evaluates to 24. The generalized problem is NP-hard, but when formulating the problem as a constraint program, with the help of a context-free grammar constraint we can easily express the syntactic correctness of the term formed. Or, again closer to the real-world, consider the task of organizing a group of workers into a number of teams of unspecified size, each team with one team leader and one project manager who is the head of all team leaders. This organizational structure can be captured easily by a combination

of an all-different and a context-free grammar constraint. Therefore, in this section we will develop an algorithm that propagates context-free grammar constraints.

6.3.1 Parsing Context-Free Grammars

One of the most famous algorithms for parsing context-free grammars is the algorithm by Cocke, Younger, and Kasami (CYK). It takes as input a word $w \in \Sigma^n$ and a context-free grammar $G = (\Sigma, N, P, S_0)$ in some special form and decides in time $O(n^3|G|)$ whether it holds that $w \in L_G$. The algorithm is based on the dynamic programming principle. In order to keep the recursion equation under control, the algorithm needs to assume that all productions are length-bounded on the right-hand side.

Chomsky Normal Form A Type-2 or context-free grammar $G = (\Sigma, N, P, S_0)$ is said to be in *Chomsky Normal Form* if and only if for all productions $(A \to \alpha) \in P$ we have that $\alpha \in \Sigma^1 \cup N^2$. Without loss of generality, we will then assume that each literal $a \in \Sigma$ is associated with exactly one unique non-literal $A_a \in N$ such that $(B \to a) \in P$ implies that $B = A_a$ and $(A_a \to b) \in P$ implies that $a = b$.

Lemma 6.3.1. *Every context free grammar G such that $\varepsilon \notin L_G$ can be transformed into a grammar H such that $L_G = L_H$ and H is in Chomsky Normal Form.*

The proof of this lemma is given in [96]. It is important to note that the proof is constructive but that the resulting grammar H may be exponential in size of G, which is really due to the necessity to remove all productions $A \to \varepsilon$. When we view the grammar size as constant (i.e. if the size of the grammar is independent of the word-length as it is commonly assumed in the theory of formal languages), then this is not an issue. As a matter of fact, in most references one will simply read that CYK could solve the word problem for any context-free language in cubic time. For now, let us assume that indeed all grammars given can be treated as having constant-size, and that our asymptotic analysis only takes into account the increasing word lengths.For now, let us assume that indeed all grammars given can be treated as having constant-size, and that our asymptotic analysis only takes into account the increasing word lengths. We will come back to this point later in Section 6.7 when we discuss logic combinations of grammar constraints, and in Section 6.8 when we discuss the possibility of non-constant grammars and reorderings.

Now, given a word $w \in \Sigma^n$, let us denote the sub-sequence of letters starting at position i with length j (that is, $w_i w_{i+1} \dots w_{i+j-1}$) by w_{ij}. Based on a grammar $G = (\Sigma, N, P, S_0)$ in Chomsky Normal Form, CYK determines iteratively the set of all non-terminals from where we can derive w_{ij}, i.e. $S_{ij} := \{A \in N \mid A \overset{*}{\underset{G}{\Rightarrow}} w_{ij}\}$ for all $1 \leq i \leq n$ and $1 \leq j \leq n - i$. It is easy to initialize the sets S_{i1} just based on w_i and

all productions $(A \to w_i) \in P$. Then, for j from 2 to n and i from 1 to $n - j + 1$, we have that

$$S_{ij} = \bigcup_{k=1}^{j-1} \{A \mid (A \to BC) \in P \text{ with } B \in S_{ik} \text{ and } C \in S_{i+k,j-k}\}. \tag{6.1}$$

Then, $w \in L_G$ if and only if $S_0 \in S_{1n}$. From the recursion equation it is simple to derive that CYK can be implemented to run in time $O(n^3|G|) = O(n^3)$ when we treat the size of the grammar as a constant.

6.3.2 Example

Assume we are given the following context-free, normal-form grammar $G = (\{],[\}, \{A, B, C, S_0\}, \{S_0 \to AC, S_0 \to S_0S_0, S_0 \to BC, B \to AS_0, A \to [\ , C \to\ ]\ \}, S_0)$ that gives the language L_G of all correctly bracketed expressions (like, for example, "[[][]]" or "[][[]]"). Given the word "[][[]]", CYK first sets $S_{11} = S_{31} = S_{41} = \{A\}$, and $S_{21} = S_{51} = S_{61} = \{C\}$. Then it determines the non-terminals from which we can derive sub-sequences of length 2: $S_{12} = S_{42} = \{S_0\}$ and $S_{22} = S_{32} = S_{52} = \emptyset$. The only other non-empty sets that CYK finds in iterations regarding longer sub-sequences are $S_{34} = \{S_0\}$ and $S_{16} = \{S_0\}$. Consequently, since $S_0 \in S_{16}$, CYK decides (correctly) that $[][[]] \in L_G$.

6.3.3 Context-Free Grammar Filtering

We denote a given grammar constraint $Grammar_G(X_1, \ldots, X_n)$ over a context-free grammar G in Chomsky Normal Form by $CFGC_G(X_1, \ldots, X_n)$. Obviously, we can use CYK to determine whether $CFGC_G(X_1, \ldots, X_n)$ is satisfied for a full instantiation of the variables, i.e. we could use the parser for generate-and-test purposes. In the following, we show how we can augment CYK to a filtering algorithm that achieves generalized arc-consistency for $CFGC$. An alternative filtering algorithm based on the Earley parser is presented in [154].

First, we observe that we can check the satisfiability of the constraint by making just a very minor adjustment to CYK. Given the domains of the variables, we can decide whether there exists a word $w \in D_1 \ldots D_n$ such that $w \in L_G$ simply by adding all non-terminals A to S_{i1} for which there exists a production $(A \to v) \in P$ with $v \in D_i$. From the correctness of CYK it follows trivially that the constraint is satisfiable if and only if $S_0 \in S_{1n}$. The runtime of this algorithm is the same as that for CYK.

As usual, whenever we have a polynomial-time algorithm that can decide the satisfiability of a constraint, we know already that achieving arc-consistency is also computationally tractable. A brute force approach could simply probe values by setting $D_i := \{v\}$, for every $1 \leq i \leq n$ and every $v \in D_i$, and checking

1. We run the dynamic program based on recursion equation (1) with initial sets $S_{i1} := \{A \mid (A \to v) \in P, v \in D_i\}$.
2. We define the directed graph $Q = (V, E)$ with node set $V := \{v_{ijA} \mid A \in S_{ij}\}$ and arc set $E := E_1 \cup E_2$ with $E_1 := \{(v_{ijA}, v_{ikB}) \mid \exists\, C \in S_{i+k,j-k} :\ (A \to BC) \in P\}$ and $E_2 := \{(v_{ijA}, v_{i+k,j-k,C}) \mid \exists\, B \in S_{ik} :\ (A \to BC) \in P\}$ (see Figure 6.1).
3. Now, we remove all nodes and arcs from Q that cannot be reached from v_{1nS_0} and denote the resulting graph by $Q' := (V', E')$.
4. We define $S'_{ij} := \{A \mid v_{ijA} \in V'\} \subseteq S_{ij}$, and set $D'_i := \{v \mid \exists\, A \in S'_{i1} :\ (A \to v) \in P\}$.

Algorithm 5: CFGC Filtering Algorithm

whether the constraint is still satisfiable or not. This method would result in a runtime in $O(n^4 D|G|)$, where $D \leq |\Sigma|$ is the size of the largest domain D_i.

We will now show that we can achieve a much improved filtering time. The core idea is once more to exploit Trick's method of filtering dynamic programs [183]. Roughly speaking, when applied to our CYK-constraint checker, Trick's method simply reverses the recursion process after it has assured that the constraint is satisfiable so as to see which non-terminals in the sets S_{i1} can actually be used in the derivation of any word $w \in L_G \cap (D_1 \dots D_n)$. The methodology is formalized in Algorithm 5.

Lemma 6.3.2. *In Algorithm 5:*

1. *It holds that $A \in S_{ij}$ if and only if there exists a word $w_i \dots w_{i+j-1} \in D_i \dots D_{i+j-1}$ such that $A \overset{*}{\underset{G}{\Rightarrow}} w_i \dots w_{i+j-1}$.*
2. *It holds that $B \in S'_{ik}$ if and only if there exists a word $w \in L_G \cap (D_1 \dots D_n)$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\, B\, w_{i+k} \dots w_n$.*

Proof. Proof

1. We induce over j. For $j = 1$, the claim holds by definition of S_{i1}. Now assume $j > 1$ and that the claim is true for all S_{ik} with $1 \leq k < j$. Now, by definition of S_{ij}, $A \in S_{ij}$ if and only if there exists a $1 \leq k < j$ and a production $(A \to BC) \in P$ such that $B \in S_{ik}$ and $C \in S_{i+k,j-k}$. Thus, $A \in S_{ij}$ if and only if there exist $w_{ik} \in D_i \dots D_{i+k-1}$ and $w_{i+k,j-k} \in D_{i+k} \dots D_{i+j-1}$ such that $A \overset{*}{\underset{G}{\Rightarrow}} w_{ik} w_{i+k,j-k}$.
2. We induce over k, starting with $k = n$ and decreasing to $k = 1$. For $k = n$, $S'_{1k} = S'_{1n} \subseteq \{S_0\}$, and it is trivially true that $S_0 \overset{*}{\underset{G}{\Rightarrow}} S_0$. Now let us assume the claim holds for all S'_{ij} with $k < j \leq n$. Choose any $B \in S'_{ik}$. According to the definition of S'_{ik} there exists a path from v_{1nS_0} to v_{ikB}. Let $(v_{ijA}, v_{ikB}) \in E_1$ be the last arc on any one such path (the case when the last arc is in E_2 follows analogously). By the definition of E_1 there exists a production $(A \to BC) \in P$ with $C \in S_{i+k,j-k}$. By induction hypothesis, we know that there exists a word $w \in L_G \cap (D_1 \dots D_n)$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}}$

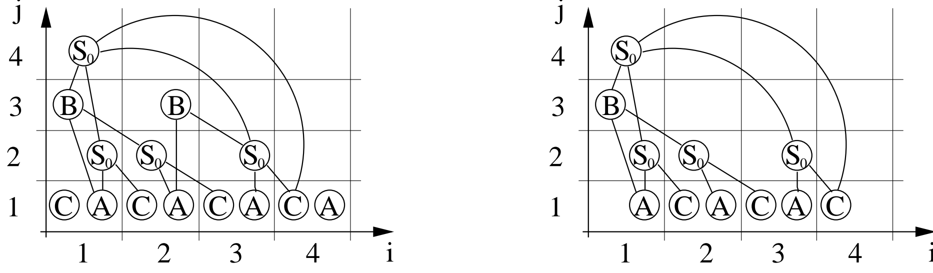


Figure 6.1: Context-Free Filtering: A rectangle with coordinates (i, j) contains nodes v_{ijA} for each non-terminal A in the set S_{ij}. All arcs are considered to be directed from top to bottom. The left picture shows the situation after step (2). S_0 is in S_{14}, therefore the constraint is satisfiable. The right picture illustrates the shrunken graph with sets S'_{ij} after all parts have been removed that cannot be reached from node v_{14S_0}. We see that the value ']' will be removed from D_1 and '[' from D_4.

$w_1 \dots w_{i-1}\ A\ w_{i+j} \dots w_n$. Thus, $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ BC\ w_{i+j} \dots w_n$. And therefore, with the readily proven fact (1) and $C \in S_{i+k,j-k}$, there exists a word $w_{i+k} \dots w_{i+j-1} \in D_{i+k} \dots D_{i+j-1}$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ B\ w_{i+k} \dots w_{i+j-1}\ w_{i+j} \dots w_n$. Since we can also apply (1) to non-terminal B, we have proven the claim.

□

Theorem 6.3.3. *Algorithm 5 achieves generalized arc-consistency for the $CFGC$.*

Proof. Proof We show that $v \notin D'_i$ if and only if for all words $w = w_1 \dots w_n \in L_G \cap (D_1 \dots D_n)$ it holds that $v \neq w_i$.

- ⇒ (Soundness) Let $v \notin D'_i$ and $w = w_1 \dots w_n \in L_G \cap (D_1 \dots D_n)$. Due to the assumption that $w \in L_G$ there must exist a derivation $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ A\ w_{i+1} \dots w_n \underset{G}{\Rightarrow} w_1 \dots w_{i-1} w_i w_{i+1} \dots w_n$ for some $A \in N$ with $(A \rightarrow w_i) \in P$. According to Lemma 6.3.2, $A \in S'_{i1}$, and thus $w_i \in D'_i$, which implies $v \neq w_i$ as $v \notin D'_i$.
- ⇐ (Completeness) Now let $v \in D'_i \subseteq D_i$. According to the definition of D'_i, there exists some $A \in S'_{i1}$ with $(A \rightarrow v) \in P$. With Lemma 6.3.2 we know that then there exists a word $w \in L_G \cap (D_1 \dots D_n)$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ A\ w_{i+1} \dots w_n$. Thus, it holds that $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ v\ w_{i+1} \dots w_n \in L_G \cap (D_1 \dots D_n)$.

□

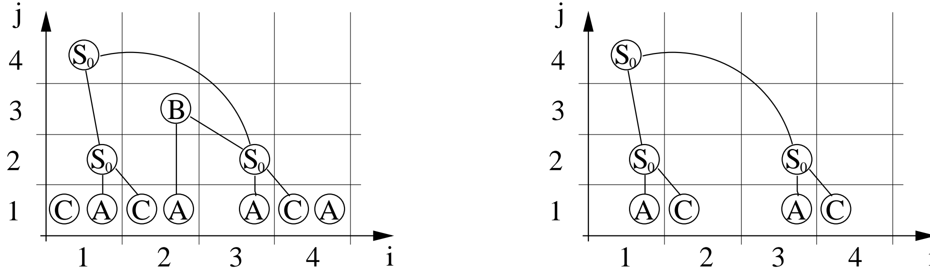


Figure 6.2: We show how the algorithm works when the initial domain of X_3 is $D_3 = \{[\}$. The left picture shows sets S_{ij} and the right the sets S'_{ij}. We see that the constraint filtering algorithm determines the only word in $L_G \cap D_1 \dots D_4$ is "[][]".

6.3.4 Example

Assume we are given the context-free grammar from Section 6.3.2 again. In Figure 6.1, we illustrate how Algorithm 5 works on the problem: First, we work bottom-up, adding non-terminals to the sets S_{ij} if they allow to generate a word in $D_i \times \cdots \times D_{i+j-1}$. Then, in the second step, we work top down and remove all non-terminals that cannot be reached from $S_0 \in S_{1n}$. In Figure 6.2, we show how this latter step can affect the lowest level, where the removal of non-terminals results in the pruning of the domains of the variables.

6.3.5 Runtime Analysis

We now have a filtering algorithm that achieves generalized arc-consistency for context-free grammar constraints. Since the computational effort is dominated by carrying out the recursion equation, Algorithm 5 runs asymptotically in the same time as CYK. In essence, this implies that checking one complete assignment via CYK is as costly as performing full arc-consistency filtering for $CFGC$. Clearly, achieving arc-consistency for a grammar constraint is at least as hard as parsing. Now, there exist faster parsing algorithms for context-free grammars. For example, the fastest known algorithm was developed by Valiant and parses context-free grammars in time $O(n^{2.8})$. While this is only moderately faster than the $O(n^3)$ that CYK requires, there also exist special purpose parsers for non-ambiguous context-free grammars (i.e. grammars where each word in the language has exactly one parse tree) that run in $O(n^2)$. It is known that there exist inherently ambiguous context-free languages, so these parsers lack some generality. However, in case that a user specifies a grammar that is non-ambiguous it would actually be nice to have a filtering algorithm that runs in quadratic rather than cubic time. It is a matter of further research to find out whether grammar constraint propagation can be done faster for non-ambiguous context-free grammars.

6.4 Efficient Context-Free Grammar Filtering

Given the fact that context-free grammar filtering entails the word problem, there is little hope that, for general context-free grammar constraints, we can devise significantly faster filtering algorithms. However, with respect to filtering performance it is important to realize that a filtering algorithm should work quickly within a constraint propagation engine. Typically, constraint filtering needs to be conducted on a sequence of problems that differ only slightly from the last. When branching decisions and other constraints tighten a given problem, state-of-the-art systems like Ilog Solver provide a filtering routine with information regarding which values were removed from which variable domains since the last call to the routine. By exploiting such condensed information, incremental filtering routines can be devised that work faster than starting each time from scratch. To analyze such incremental algorithms, it has become the custom to provide an upper bound on the total work performed by a filtering algorithm over one entire branch of the search tree (see, e.g., [113]).

Naturally, given a sequence of s monotonically tightening problems (that is, when in each successive problem the variable domains are subsets of the previous problem), context-free grammar constraint filtering for the entire sequence takes at most $O(sn^3|G|)$ steps. Using existing ideas how to perform incremental graph updates in DAGs efficiently (see for instance [49]), it is trivial to modify Algorithm 5 so that this time is reduced to $O(n^3|G|)$: Roughly, when storing additional information which productions support which arcs in the graph (whereby each production can support at most $2n$ arcs for each set S_{ij}), we can propagate the effects of domain values being removed at the lowest level of the graph to adjacent nodes without ever touching parts of the graph that do not change. The total workload for the entire problem sequence can then be distributed over all $O(|G|n)$ production supports in each of $O(n^2)$ sets, which results in a time bound of $O(n^2|G|n) = O(n^3|G|)$.

The second efficiency aspect regards the memory requirements. In Algorithm 5, they are in $\Theta(n^3|G|)$. It is again trivial to reduce these costs to $O(n^2|G|)$ simply by recomputing the sets of incident arcs rather than storing them in step 2 for step 3 of Algorithm 5. However, when we follow this simplistic approach we only trade time for space. The incremental version of our algorithm as sketched above is based on the fact that we do not need to recompute arcs incident to a node which is achieved by storing them. So while it is straight-forward and easy to achieve a space-efficient version that requires time in $O(sn^3|G|)$ and space $O(n^2|G|)$ or a time-efficient incremental variant that requires time and space in $O(n^3|G|)$, the challenge is to devise an algorithm that combines low space requirements with good incremental performance.

In the following, we will thus modify our algorithm such that the total workload of a sequence of s monotonically tightening filtering problems is reduced to $O(n^3|G|)$, which implies that, asymptotically, an entire sequence of more and more restricted problems can be filtered with respect to context-free grammars

in the same time as it takes to filter just one problem from scratch. At the same time, we will ensure that our modified algorithm will require space in $O(n^2|G|)$.

6.4.1 Space-Efficient Incremental Filtering

In Algorithm 5, we observe that it first works bottom-up, determining from which nodes (associated with non-terminals of the grammar) we can derive a legal word. Then, it works top-down determining which non-terminal nodes can be used in a derivation that begins with the start non-terminal $S_0 \in S_{1n}$. In order to save both space and time, we will modify these two steps in that every non-terminal in each set S_{ij} will perform just enough work to determine whether its respective node will remain in the shrunken graph Q' or not.

To this end, in the first step that works bottom-up we will need a routine that determines whether there exists, what we call, a *support from below*: That is, this routine determines whether a node v_{ijA} has any *outgoing* arcs in $E_1 \cup E_2$. To save space, the routine must perform this check without ever storing sets E_1 and E_2 explicitly, as this would require space in $\Theta(n^3|G|)$.

Analogously, in the second step that works top-down we will rely on a procedure that checks whether there exists a *support from above*: Formally, this procedure determines whether a node v_{ijA} has any *incoming* arcs in $E_1' \cup E_2'$, again without ever storing these sets which would require too much memory.

The challenge here is to avoid having to pay with time what we save in space. To this end, we need a methodology which prevents us from searching for supports (from above or below) that have been checked unsuccessfully before. Very much like the well-known arc-consistency algorithm AC-6 for binary constraint problems [11], we achieve this goal by ordering potential supports so that, when a support is lost, the search for a new support can start right after the last support, in the respective ordering.

According to the definition of E_1, E_2, E_1', E_2', supports of v_{ijA} (from above or below) are directly associated with productions in the given grammar G and a splitting index k. To order these supports, we cover and order the productions in G that involve non-terminal A in two lists:

- In list $Out_A := [(A \rightarrow B_1B_2) \in P]$ we store and implicitly fix an ordering on all productions with non-terminal A on the left-hand side.
- In list $In_A := [(B_1 \rightarrow B_2B_3) \in P \mid B_2 = A \ \vee \ B_3 = A]$ we store and implicitly fix an ordering on all productions where non-terminal A appears as non-terminal on the right-hand side.

Now, for each node v_{ijA} we store two production indices p_{ijA}^{Out} and p_{ijA}^{In}, two splitting indices $k_{ijA}^{Out} \in \{1,\dots,j\}$ and $k_{ijA}^{In} \in \{j,\dots,n\}$, and a flag l_{ijA}. The intended meaning of these indices is that node v_{ijA}

```
void findOutSupport(i, j, A)
p^{Out}_{ijA} ← p^{Out}_{ijA} + 1
while k^{Out}_{ijA} < j do
  while p^{Out}_{ijA} ≤ |Out_A| do
    (A → B_1 B_2) ← Out_A[p^{Out}_{ijA}]
    if (B_1 ∈ S_{i k^{Out}_{ijA}}) and (B_2 ∈ S_{i+k^{Out}_{ijA}, j−k^{Out}_{ijA}}) then
      return
    end if
    p^{Out}_{ijA} ← p^{Out}_{ijA} + 1
  end while
  p^{Out}_{ijA} ← 1, k^{Out}_{ijA} ← k^{Out}_{ijA} + 1
end while

bool findInSupport(i, j, A)
// returns true iff afterwards In_A[p^{In}_{ijA}] = (B_1 → A B_3) for some B_1, B_3 ∈ N
p^{In}_{ijA} ← p^{In}_{ijA} + 1
while k^{In}_{ijA} > j do
  while p^{In}_{ijA} ≤ |In_A| do
    (B_1 → B_2 B_3) ← In_A[p^{In}_{ijA}]
    if (A = B_2) then
      if (B_1 ∈ S_{i k^{In}_{ijA}}) and (B_3 ∈ S_{i+j, k^{In}_{ijA}−j}) then
        return true
      end if
    end if
    if (A = B_3) then
      if (B_1 ∈ S_{i+j−k^{In}_{ijA}, k^{In}_{ijA}}) and (B_2 ∈ S_{i+j−k^{In}_{ijA}, k^{In}_{ijA}−j}) then
        return false
      end if
    end if
    p^{In}_{ijA} ← p^{In}_{ijA} + 1
  end while
  p^{In}_{ijA} ← 1, k^{In}_{ijA} ← k^{In}_{ijA} − 1
end while
return false
```

Algorithm 6: Functions that incrementally (re-)compute the support from below and above for a given node v_{ijA}.

is currently supported from below by production $(A \rightarrow B_1B_2) = Out_A[p^{Out}_{ijA}]$ such that $B_1 \in S_{i,k^{Out}_{ijA}}$ and $B_2 \in S_{i+k^{Out}_{ijA},j-k^{Out}_{ijA}}$. Analogously, the current support from above is production $(B_1 \rightarrow B_2B_3) = In_A[p^{In}_{ijA}]$ such that $B_1 \in S'_{i,k^{In}_{ijA}}$ and $B_3 \in S'_{i+j,k^{In}_{ijA}-j}$ if $B_2 = A$, in which case l_{ijA} equals to true, or $B_1 \in S'_{i+j-k^{In}_{ijA},k^{In}_{ijA}}$ and $B_2 \in S'_{i+j-k^{In}_{ijA},k^{In}_{ijA}-j}$ if $B_3 = A$, in which case l_{ijA} equals to false. When node v_{ijA} has no support from below (or above), we will have $k^{Out}_{ijA} = j$ ($k^{In}_{ijA} = j$).

In Algorithm 6, we show two functions that (re-)compute the support from below and above for a given node v_{ijA}, whereby we assume that variables p^{Out}_{ijA}, p^{In}_{ijA}, k^{Out}_{ijA}, k^{In}_{ijA}, S_{ij}, Out_A, and In_A are global and initialized outside these functions. We see that both routines start their search for a new support right after the last. This is correct as within a sequence of monotonically tightening problems we will never add edges to the graphs Q and Q'. Therefore, replacement supports can only be found later in the respective ordering

```
bool filterFromScratch(D_1, ..., D_n)
// return true iff constraint can be satisfied
for r = 1 to n do
  S_r1 ← ∅
  for all a ∈ D_r do
    S_r1 ← S_r1 ∪ {A_a}
  end for
end for
for j = 2 to n do
  for i = 1 to n − j + 1 do
    for all A ∈ N do
      p^Out_ijA ← 0, k^Out_ijA ← 1
      findOutSupport(i,j,A)
      if (k^Out_ijA < j) then
        S_ij ← S_ij ∪ {A}
        lost^Out_ijA ← false
        informOutSupport(i, j, A, Add)
      end if
    end for
  end for
end for
if (S_0 ∉ S_1n) then
  return false
end if
S_1n ← {S_0}
for j = n − 1 downto 1 do
  for i = 1 to n − j + 1 do
    for all A ∈ S_ij do
      p^In_ijA ← 0, k^In_ijA ← 1
      l_ijA ← findInSupport(i, j, A)
      if (k^In_ijA > j) then
        lost^In_ijA ← false
        informInSupport(i, j, A, Add)
      else
        S_ij ← S_ij \ {A}
        lost^Out_ijA ← true
        informOutSupport(i, j, A, Remove)
      end if
    end for
  end for
end for
for r = 1 to n do
  D_r ← {a | A_a ∈ S_r1}
end for
return true
```

Algorithm 7: A method that performs context-free grammar filtering in time $O(n^3|G|)$ and space $O(n^2|G|)$.

of supports. Note that the second function which computes a new support from above is slightly more complicated as it needs to make the distinction whether the target non-terminal A appears as first or second non-terminal on the right-hand side of its support production. This information is returned by the second function to facilitate the handling of these two cases.

In Algorithm 7, we illustrate the use of the two support computing functions. The algorithm given here performs context-free grammar filtering from scratch when provided with the current domains $D_1, \dots, D_n$. We assume again that all global variables are allocated outside of this function. Again, we observe the main two phases in which the algorithm proceeds: After initializing the sets S_{i1}, the algorithm first computes supports from below for nodes on higher levels. In contrast to Algorithm 5, the modified method computes just *one* support from below rather than all of them. Of course, we use the previously devised function findOutSupport for this purpose. After checking whether the constraint can be satisfied at all (test of $S_0 \in S_{1n}$), in the second phase, the algorithm then performs the analogous computation of supports from above with the help of the previously devised function findInSupport. Note that we do not introduce sets S'_{ij} anymore as, for potentially following incremental updates, we would otherwise need to set $S_{ij} = S'_{ij}$ anyway.

In both phases, we note calls to routines informOutSupport and informInSupport. These functions are given in Algorithm 8 and are there to inform the nodes on which the support of the current node relies about this fact. This would not be necessary if we only wanted to filter the constraint once. However, we later want to propagate incrementally the effects of removed nodes, and to do this efficiently, we need a fast way of determining which other nodes currently rely on their existence. At the same time, in pointers left and right we store where the information is located at the supporting nodes so that it is easy to update it when a support should have to be replaced later (because one of the two supporting nodes is removed). Given that both functions work in constant time, the overhead of providing the infrastructure for incremental constraint filtering is minimal and invisible in the O-calculus.

Finally, in Algorithm 9 we present our function filterFromUpdate that re-establishes arc-consistency for the context-free grammar constraint based on the information which variables were affected and which values were removed from their domains. The function starts by iterating through the domain changes, whereby each node on the lowest level adds those nodes whose current support relies on its existence to a list of affected nodes (nodesListOut and nodesListIn). This is a simple task since we have stored this information before. Furthermore, by organizing the affected nodes according to the level to which they belong, we make it easy to perform the two phases (one working bottom-up, the other top-down) later, whereby a simple flag (lost) ensures that no node is added twice.

In Algorithm 10, we show how the phase that recomputes the supports from below proceeds: We iterate through the affected nodes bottom-up. First, for each node that has lost its support from below, because

void informOutSupport$(i, j, A, State)$
$(A \rightarrow B_1 B_2) \leftarrow Out_A[p^{Out}_{ijA}]$
if $State == Add$ **then**
  left$^{Out}_{ijA} \leftarrow$ listOfOutSupported$_{i,k^{Out}_{ijA},B_1}$.add(i, j, A)
  right$^{Out}_{ijA} \leftarrow$ listOfOutSupported$_{i+k^{Out}_{ijA},j-k^{Out}_{ijA},B_2}$.add$(i, j, A)$
else
  listOfOutSupported$_{i,k^{Out}_{ijA},B_1}$.remove(left$^{Out}_{ijA}$)
  listOfOutSupported$_{i+k^{Out}_{ijA},j-k^{Out}_{ijA},B_2}$.remove(right$^{Out}_{ijA}$)
end if

void informInSupport$(i, j, A, State)$
$(B_1 \rightarrow B_2 B_3) \leftarrow In_A[p^{In}_{ijA}]$
if $State == Add$ **then**
  if l_{ijA} **then**
    left$^{In}_{ijA} \leftarrow$ listOfInSupported$_{i,k^{In}_{ijA},B_1}$.add(i, j, A)
    right$^{In}_{ijA} \leftarrow$ listOfInSupported$_{i+j,k^{In}_{ijA}-j,B_3}$.add$(i, j, A)$
  else
    left$^{In}_{ijA} \leftarrow$ listOfInSupported$_{i+j-k^{In}_{ijA},k^{In}_{ijA},B_1}$.add$(i, j, A)$
    right$^{In}_{ijA} \leftarrow$ listOfInSupported$_{i+j-k^{In}_{ijA},k^{In}_{ijA}-j,B_2}$.add$(i, j, A)$
  end if
else
  if l_{ijA} **then**
    listOfInSupported$_{i,k^{In}_{ijA},B_1}$.remove(left$^{In}_{ijA}$)
    listOfInSupported$_{i+j,k^{In}_{ijA}-j,B_3}$.remove(right$^{In}_{ijA}$)
  else
    listOfInSupported$_{i+j-k^{In}_{ijA},k^{In}_{ijA},B_1}$.remove(left$^{In}_{ijA}$)
    listOfInSupported$_{i+j-k^{In}_{ijA},k^{In}_{ijA}-j,B_2}$.remove(right$^{In}_{ijA}$)
  end if
end if

Algorithm 8: Functions that inform nodes which other nodes they support from below or above.

one of its supporting nodes was lost, we inform the other supporting node that it is no longer supporting the current node (for the sake of simplicity, we simply inform both supporting nodes, even though at least one of them will never be looked at again anyway). Then, we try to replace the lost support from below by calling findOutSupport. Recall that the function seeks a new support starting at the old, so that no two potential supports are investigated more than just once. Now, if we were successful in providing a new support from below (test $k^{Out}_{ijA} < j$), we inform the new supporting nodes that the support of the current node relies on them. Otherwise, the current node is removed and the nodes that it supports are being added to the lists of affected nodes. The second phase, presented in Algorithm 11, works analogously. The one interesting difference regards the fact that, in function updateInSupport, nodes that are removed because they lost their support from above do not inform the nodes that they support from below. This is obviously not necessary as the supported nodes must have been removed before as they could otherwise provide a valid support from above. This is also the reason why, after having conducted one bottom-up phase and one top-down in filterFromUpdate, all remaining nodes must have an active support from below and above.

```
void filterFromUpdate(varSet,Δ_1, ..., Δ_n)
for r = 1 to n do
  nodeListOut[r] ← ∅, nodeListIn[r] ← ∅
end for
for all X_i ∈ varSet do
  for all a ∈ Δ_i do
    if (A_a ∈ S_i1) then
      S_i1 ← S_i1 \ {A_a}
      lost^In_ijA ← true
      informInSupport(i, j, A, Remove)
      for all (p, q, B) ∈ listOfOutSupported_{i,j,A_a} do
        if (lost^Out_pqB) then
          continue
        end if
        lost^Out_pqB ← true
        nodeListOut[q].add(p, q, B)
      end for
      for all (p, q, B) ∈ listOfInSupported_{i,j,A_a} do
        if (lost^In_pqB) then
          continue
        end if
        lost^In_pqB ← true
        nodeListIn[q].add(p, q, B)
      end for
    end if
  end for
end for
updateOutSupports()
if (S_0 ∉ S_1n) then
  return false
end if
S_1n ← {S_0}
updateInSupports()
return true
```

Algorithm 9: A method that performs context-free grammar filtering incrementally in space $O(n^2|G|)$ and amortized total time $O(n^3|G|)$ for any sequence of monotonically tightening problems.

With the complete method as outlined in Algorithms 6–11, we can now show:

Theorem 6.4.1. *For a sequence of s monotonically tightening context-free grammar constraint filtering problems, based on the grammar G in Chomsky Normal Form filtering for the entire sequence can be performed in time $O(n^3|G|)$ and space $O(n^2|G|)$.*

Proof. Proof Our algorithm is complete since, as we just mentioned, upon termination all remaining nodes have a valid support from above and below. With the completeness proof of Algorithm 5, this implies that we filter enough. On the other hand, we also never remove a node if there still exist a support from above and below: we know that, if a support is lost, a replacement can only be found later in the chosen ordering of supports. Therefore, if functions findOutSupport or findInSupport fail to find a replacement support, then none exists. Consequently, our filtering algorithm is also sound.

```
void updateOutSupports(void)
for r = 2 to n do
  for all (i, j, A) ∈ nodeListOut[r] do
    informOutSupport(i, j, A, Remove)
    findOutSupport(i,j,A)
    if (k^Out_ijA < j) then
      lost^Out_ijA ← false
      informOutSupport(i, j, A, Add)
      continue
    end if
    S_ij ← S_ij \ {A}
    lost^In_ijA ← true
    informInSupport(i, j, A, Remove)
    for all (p, q, B) ∈ listOfOutSupported_i,j,A do
      if (lost^Out_pqB) then
        continue
      end if
      lost^Out_pqB ← true
      nodeListOut[q].add(p, q, B)
    end for
    for all (p, q, B) ∈ listOfInSupported_i,j,A do
      if (lost^In_pqB) then
        continue
      end if
      lost^In_pqB ← true
      nodeListIn[q].add(p, q, B)
    end for
  end for
end for
```

Algorithm 10: Function computing new supports from below by proceeding bottom-up.

```
void updateInSupports(void)
for r = n − 1 downto 1 do
  for all (i, j, A) ∈ nodeListIn[r] do
    informInSupport(i, j, A, Remove)
    l_ijA ← findInSupport(i,j,A)
    if (k^In_ijA > j) then
      lost^In_ijA ← false
      informInSupport(i, j, A, Add)
      continue
    end if
    if (j = 1) then
      D_i ← D_i \ {a | A = A_a}
    end if
    S_ij ← S_ij \ {A}
    lost^Out_ijA ← true
    informOutSupport(i, j, A, Remove)
    for all (p, q, B) ∈ listOfInSupported_{i,j,A} do
      if (lost^In_pqB) then
        continue
      end if
      lost^In_pqB ← true
      nodeListIn[q].add(p, q, B)
    end for
  end for
end for
```

Algorithm 11: Function computing new supports from above by proceeding top-down.

With respect to the space requirements, we note that the total space needed to store, for each of the $O(n^2|G|)$ nodes, those nodes that are supported by them is not larger then the number of nodes supporting each node (which is equal to four) times the number of all nodes. Therefore, while an individual node may support many other nodes, for all nodes together the space required to store this information is bounded by $O(4n^2|G|) = O(n^2|G|)$. All other global arrays (like left, right, p, k, S, lost, and so on) also only require space in $O(n^2|G|)$.

Finally, it remains to analyze the total effort for a sequence of s monotonically tightening filtering problems. Given that, in each new iteration, at least one assignment is lost, we know that $s \leq |G|n$. For each filtering problem, we need to update the lowest level of nodes and then iterate twice (once bottom-up and once top-down) through all levels (even if levels should turn out to contain no affected nodes), which imposes a total workload in $O(|G|n + 2n|G|n) = O(n^2|G|)$. All other work is dominated by the total work done in all calls to functions findInSupport and findOutSupport. Since these functions are never called for any node for which it has been assessed before that it has no more support from either above or below, each time that one of these functions is called, the support pointer of some node is increased by at least one. Again we find that the total number of potential supports for an individual node could be as large as $\Theta(|G|n)$, while the number of potential supports for all nodes together is asymptotically not larger. Consequently, the total work performed is bounded by the number of sets S_{ij} times the number of potential supports for all nodes in each

of them. Thus, the entire sequence of filtering problems can be handled in $O(n^2|G|n) = O(n^3|G|)$. $\square$

Note that the above theorem states time- and space complexity for monotonic reductions without restorations only! Within a backtrack search, we need to also store lost supports for each node leading to the current choice point so that we can backtrack quickly. However, as we will see in the next section, in practice the amount of such additional information needed is very low.

6.5 Numerical Results

We implemented the previously outlined incremental context-free grammar constraint propagation algorithm and compared it against its non-incremental counterpart on real-world instances of the shift-scheduling problem introduced in [40]. We chose this problem because it is the only real-world problem grammar constraints have been tested on before in [155]. [1] The problem is that of a retail store manager who needs to staff his employees such that the expected demands of workers for various activities that must be performed at all times of the day are met. The demands are given as upper and lower bounds of workers performing an activity a_i at each 15-minute time period of the day.

Labor requirements govern the feasibility of a shift for each worker: 1. A work-shift covers between 3 and 8 hours of actual work activities. 2. If a work-activity is started, it must be performed for at least one consecutive hour. 3. When switching from one work-activity to another, a break or lunch is required in between. 4. Breaks, lunches, and off-shifts do not follow each other directly. 5. Off-shifts must be assigned at the beginning and at the end of each work-shift. 6. If the actual work-time of a shift is at least 6 hours, there must be two 15-minute breaks and one one-hour lunch break. 7. If the actual work-time of a shift is less than 6 hours, then there must be exactly one 15-minute break and no lunch-break. We implemented these constraints by means of one context-free grammar constraint per worker and several global-cardinality constraints ("vertical" gcc's over each worker's shift to enforce the last two constraints, and "horizontal" gcc's for each time period and activity to meet the workforce demands) in Ilog Solver 6.4. To break symmetries between the indistinguishable workers, we introduced constraints that force the ith worker to work at most as much as the i+first worker.

Table 6.1 summarizes the results of our experiments. We see that the incremental propagation algorithm vastly outperforms its non-incremental counterpart, resulting in speed-ups of up to a factor 188 while exploring *identical* search-trees! It is quite rare to find that the efficiency of filtering techniques leads to such dramatic improvements with unchanged filtering effectiveness. These results confirm the speed-ups reported in [155]. We also tried to use the decomposition approach from [155], but, due to the method's excessive

[1] Many thanks to L.-M. Rousseau for providing the benchmark!

Benchmark			Search-Tree			Non-Incremental		Incremental		Speedup
ID	#Act.	#Workers	#Fails	#Prop's	#Nodes	Time [sec]	Mem [MB]	Time [sec]	Mem [MB]	
1_1	1	1	2	136	19	79	12	1.75	24	45
1_2	1	3	135	573	281	293	38	5.7	80	51
1_3	1	4	399	982	559	455	50	8.12	106	56
1_4	1	5	17	755	181	443	60	9.08	124	46
1_5	1	4	6	598	137	399	50	7.15	104	55
1_6	1	5	6	745	147	487	60	9.1	132	53
1_7	1	6	5311	6282	5471	1948	72	16.13	154	120
1_8	1	2	29	314	102	193	26	3.57	40	54
1_9	1	1	2	144	35	80	16	1.71	18	47
1_10	1	7	18890	20511	19091	4813	82	25.57	176	188
2_1	2	2	10	12	431	66	44	7.34	88	49
2_5	2	4	30	419	109	355	44	7.37	88	48
2_6	2	5	24	604	604	168	58	9.89	106	50
2_7	2	6	44	850	158	713	84	15.14	178	47
2_8	2	2	13070	10596	13184	331	44	3.57	84	92
2_9	2	1	16	252	56	220	32	4.93	52	44
2_10	2	7	303	1123	512	900	132	17.97	160	50

Table 6.1: Shift-scheduling: We report running times on an AMD Athlon 64 X2 Dual Core Processor 3800+ for benchmarks with one and two activity types. For each worker, the corresponding grammar in CNF has 30 non-terminals and 36 productions. Column #Propagations shows how often the propagation of grammar constraints is called for. Note that this value is different from the number of choice points as constraints are usually propagated more than just once per choice point.

memory requirements, on our machine with 2 GByte main memory we were only able to solve benchmarks with one activity and one worker only (1_1 and 1_9). On these instances, the decomposition approach implemented in Ilog Solver 6.4 runs about ten times slower than our approach (potentially because of swapped memory) and uses about 1.8 GBytes memory. Our method, on the other hand, requires only 24 MBytes. Finally, when comparing the memory requirements of the non-incremental and the incremental variants, we find that the additional memory needed to store restoration data is limited in practice.

Our next computational evaluation stems from the idea of guiding the search toward satisfying solutions using the information provided by the propagator of the context-free grammar constraint. The propagator, which is based on the CYK parser, requires us to store some data structures to achieve the domain filtering. We also have to maintain these data structures during search, either in an incremental or non-incremental fashion. It is then a reasonable question whether we can also employ these data structures to guide the search as well. In other words, it would be beneficial to exploit these structures, not only for the

Benchmark			Default Search			Guided Search		
ID	#Act.	#Workers	#Fails	#Prop's	#Nodes	#Fails	#Prop's	#Nodes
1_1	1	1	2	136	19	2	136	19
1_2	1	3	135	573	281	127	571	283
1_3	1	4	399	982	559	381	968	550
1_4	1	5	17	755	181	15	761	198
1_5	1	4	6	598	137	4	608	162
1_6	1	5	6	745	147	4	753	170
1_7	1	6	5311	6282	5471	5303	6282	5485
1_8	1	2	29	314	102	29	314	102
1_9	1	1	2	144	35	2	144	35
1_10	1	7	18890	20511	19091	18850	20492	19071
2_1	2	2	10	12	431	485	853	598
2_3	2	5	–	–	–	41	771	227
2_5	2	4	30	419	109	245	674	414
2_6	2	5	24	604	604	6	653	225
2_7	2	6	44	850	158	459	1382	633
2_8	2	2	13070	10596	13184	0	452	218
2_9	2	1	16	252	56	130	444	252
2_10	2	7	303	1123	512	4	1055	390

Table 6.2: Effects of Value Ordering: We compare the search trees generated by the default search heuristic and the the search guided by our context-free grammar constraint's propagator. The default search heuristic selects minimum domain size variable and assigns it the minimum value in its domain. The guided search heuristic again selects the variable with minimum domain size, but assigns it the value for which the corresponding production rule has the most number of supports, when both *in* and *out* supports are combined. The time limit is set to 1.000 seconds. We use a dash to indicate an instance that hits the time limit.

inference mechanism, but also for search. In particular, we consider two lists, $listOfInSupported$ and $listOfOutSupported$. These lists are used to store for each node which other nodes they support from below or above. As seen in Algorithm 8, these lists are updated each time there is a change in the domains. However, we are interested only in $listOfInSupported$ and $listOfOutSupported$ of nodes in sets that corresponds to variables, i.e., sets in $\{S_{11}, \ldots, S_{n1}\}$. Notice how in Algorithm 7, S_{r1} is initialized for every variable r, with the production rules that generate the values in the domain of variable r. Then, the number of supports in these two lists provides an indicator for distinguishing between values, and hence, could be used as a value selection heuristic. For each node in $\{S_{11}, \ldots, S_{n1}\}$, its *out* supported list stores the nodes that are supported from below by the value corresponding to that node. Analogously, for each node in $\{S_{11}, \ldots, S_{n1}\}$, its *in* supported list stores the nodes that support the value corresponding to that node from above. We now have three options to consider; using the cardinality of only the $listOfInSupported$,

or only the $listOfOutSupported$, or the two combined as a summation. Also there are two directions we can favor, maximum cardinality or minimum cardinality. In combination, this gives six different value orderings to choose from. Hence, the goal of our next set of experiments is to determine the benefits of using the context-free grammar constraint's propagator for value selection in practice. The search goal used in the previous experiments was based on the well-known minimum domain search heuristic for variable selection, and the minimum value first heuristic for value selection. We augment this search goal using the value orderings based on each of the six indicators mentioned above.

We present our findings in Table 6.2. We refer to the search heuristic that uses the sum of the number of nodes in $listOfInSupported$ and $listOfOutSupported$ for value ordering as the Guided Search heuristic. This heuristic applies the same variable selection heuristic as the Default Search heuristic, but for value selection, it favors the values with maximum number of supports while breaking the ties randomly. In general, we notice a small reduction in the number of failures when guided search is applied. In one of the cases, instance:2_8, the guided search conducts a backtrack-free search, i.e., the number of failures is zero, whereas the default search hits at 13.070 failures. What is more interesting is that, instance:2_3, which consists of two activities and five workers, times out in 1.000 seconds when the default heuristic is used, but it can be solved in 5.4 seconds using guided search. We also tried considering only one of the support lists at a time rather than using their combination. We did not notice a difference in the performance in that setting. Conversely, selecting the values that have the least number of supports performed very poorly, solving only four of the instances in total.

6.6 Cost-Based Filtering for Context-Free Grammar Constraints

In our next technical section, we consider problems where context-free grammar constraints appear in conjunction with a linear objective function that we are trying to maximize. Assume that each potential variable assignment $X_i \leftarrow w_i$ is associated with a profit $p^i_{w_i}$, and that our objective is to find a complete assignment $X_1 \leftarrow w_1 \in D_1, \dots, X_n \leftarrow w_n \in D_n$ such that $CFGC_G(X_1, \dots, X_n)$ is true for that instantiation and $p(w_1 \dots w_n) := \sum_{i=1}^{n} p^i_{w_i}$ is maximized. Once we have found a feasible instantiation that achieves profit T, we are only interested in improving solutions. Therefore, we consider the conjunction of the context-free grammar constraint $CFGC_G$ with the requirement that solutions ought to have profit greater than T. This conjunction of a structured constraint (in our case the grammar constraint) with an algebraic constraint that guides our search toward improving solutions is commonly referred to as an *optimization constraint*. The task of achieving generalized arc-consistency is then often called *cost-based filtering* [58]. Optimization constraints and cost-based filtering play an essential role in constrained optimization and hybrid problem decomposition methods such as CP-based Lagrangian Relaxation [156] were proposed. In Algorithm 12, we

1. For all $1 \leq i \leq n$, initialize $S_{i1} := \emptyset$. For all $1 \leq i \leq n$ and productions $(A_a \to a) \in P$ with $a \in D_i$, set $f^{i1}_{A_a} := p(a)$, and add all such A_a to S_{i1}.
2. For all $j > 1$ in increasing order, $1 \leq i \leq n$, and $A \in N$, set $f^{ij}_A := \max\{f^{ik}_B + f^{i+k,j-k}_C \mid A \underset{G}{\Rightarrow} BC, B \in S_{ik}, C \in S_{i+k,j-k}\}$, and $S_{ij} := \{A \mid f^{ij}_A > -\infty\}$.
3. If $f^{1n}_{S_0} \leq T$, then the optimization constraint is not satisfiable, and we stop.
4. Initialize $g^{1n}_{S_0} := f^{1n}_{S_0}$.
5. For all $k < n$ in decreasing order, $1 \leq i \leq n$, and $B \in N$ set $g^{ik}_B := \max\{g^{ij}_A - f^{ij}_A + f^{ik}_B + f^{i+k,j-k}_C \mid (A \to BC) \in P, A \in S_{ij}, C \in S_{i+k,j-k}\} \cup \{g^{i-j,j+k}_A - f^{i-j,j+k}_A + f^{i-j,j}_C + f^{i,k}_B \mid (A \to CB) \in P, A \in S_{i-j,j+k}, C \in S_{i-j,j}\}$.
6. For all $1 \leq i \leq n$ and $a \in D_i$ with $(A_a \to a) \in P$ and $g^{i1}_{A_a} \leq T$, remove a from D_i.

Algorithm 12: CFGC Cost-Based Filtering Algorithm

give an efficient algorithm that performs cost-based filtering for context-free grammar constraints.

We will prove the correctness of our algorithm by using the following lemma.

Lemma 6.6.1. *In Algorithm 12:*

1. *It holds that* $f^{ij}_A = \max\{p(w_{ij}) \mid A \overset{*}{\underset{G}{\Rightarrow}} w_{ij} \in D_i \times \cdots \times D_{i+j-1}\}$*, and* $S_{ij} = \{A \mid \exists\ w_{ij} \in D_i \times \cdots \times D_{i+j-1} :\ A \overset{*}{\underset{G}{\Rightarrow}} w_{ij}\}$.
2. *It holds that* $g^{ik}_B = \max\{p(w) \mid w \in L_G \cap D_1 \times \cdots \times D_n, B \overset{*}{\underset{G}{\Rightarrow}} w_{ik}, S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \ldots w_{i-1} B w_{i+k} \ldots w_n\}$.

Proof.

1. The lemma claims that the sets S_{ij} contains all non-terminals that can derive a word supported by the domains of variables $X_i, \ldots, X_{i+j-1}$, and that f^{ij}_A reflects the value of the highest profit word $w_{ij} \in D_i \times \cdots \times D_{i+j-1}$ that can be derived from non-terminal A. To prove this claim, we induce over j. For $j = 1$, the claim holds by definition of S_{i1} and f^{i1}_A in step 1. Now assume $j > 1$ and that the claim is true for all $1 \leq k < j$. Then $\max\{p(w_{ij}) \mid A \overset{*}{\underset{G}{\Rightarrow}} w_{ij} \in D_i \times \cdots \times D_{i+j-1}\} = \max\{p(w_{ij}) \mid w_{ij} \in D_i \times \cdots \times D_{i+j-1}, A \underset{G}{\Rightarrow} BC, B \overset{*}{\underset{G}{\Rightarrow}} w_{ik}, C \overset{*}{\underset{G}{\Rightarrow}} w_{i+k,j-k}\} = \max\{p(w_{ik}) + p(w_{i+k,j-k}) \mid w_{ij} \in D_i \times \cdots \times D_{i+j-1}, A \underset{G}{\Rightarrow} BC, B \overset{*}{\underset{G}{\Rightarrow}} w_{ik}, C \overset{*}{\underset{G}{\Rightarrow}} w_{i+k,j-k}\} = \max_{(A \to BC) \in P} \max\{p(w_{ik}) \mid B \overset{*}{\underset{G}{\Rightarrow}} w_{ik} \in D_i \times \cdots \times D_{i+k-1}\} + \max\{p(w_{i+k,j-k}) \mid C \overset{*}{\underset{G}{\Rightarrow}} w_{i+k,j-k} \in D_{i+k} \times \cdots \times D_{i+j-1}\} = \max\{f^{ik}_B + f^{i+k,j-k}_C \mid A \underset{G}{\Rightarrow} BC, B \in S_{ik}, C \in S_{i+k,j-k}\} = f^{ij}_A$. Then, f^{ij}_A marks the maximum over the empty set if and only if no word in accordance with the domains of $X_i, \ldots, X_{i+j-1}$ can be derived. This proves the second claim that $S_{ij} = \{A \mid f^{ij}_A > -\infty\}$ contains exactly all those non-terminals from where a word in $D_i \times \cdots \times D_{i+j-1}$ can be derived.

2. The lemma claims that the value g^{ij}_A reflects the maximum value of any word $w \in L_G \cap D_1 \times \cdots \times D_n$ in whose derivation non-terminal A can be used to produce w_{ij}. We prove this claim by induction over k, starting with $k = n$ and decreasing to $k = 1$. We only ever get past step 3 if there exists a word $w \in L_G \cap D_1 \times \cdots \times D_n$ at all. Then, for $k = n$, with the previously proven part 1

of this lemma, $\max\{p(w) \mid w \in L_G \cap D_1 \times \cdots \times D_n, S_0 \overset{*}{\underset{G}{\Rightarrow}} w_{1n}\} = f^{1n}_{S_0} = g^{1n}_{S_0}$. Now let $k < n$ and assume the claim is proven for all $k < j \leq n$. For any given $B \in N$ and $1 \leq i \leq n$, denote with $w \in L_G \cap D_1 \times \cdots \times D_n$ the word that achieves the maximum profit such that $B \overset{*}{\underset{G}{\Rightarrow}} w_{ik}$ and $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1..w_{i-1}Bw_{i+k}..w_n$. Let us assume there exist non-terminals $A, C \in N$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1..w_{i-1}Aw_{i+j}..w_n \underset{G}{\Rightarrow} w_1..w_{i-1}BCw_{i+j}..w_n \overset{*}{\underset{G}{\Rightarrow}} w_1..w_{i-1}Bw_{i+k}..w_n$ (the case where non-terminal B is introduced in the derivation by application of a production $(A \rightarrow CB) \in P$ follows analogously). Due to the fact that w achieves maximum profit for $B \in S_{ij}$, we know that $g^{ij}_A = p(w_{1,i-1}) + f^{ij}_A + p(w_{i+j,n-i-j})$. Moreover, it must hold that $f^{ik}_B = p(w_{ik})$ and $f^{i+k,j-k}_C = p(w_{i+k,j-k})$. Then, $p(w) = (p(w_{1,i-1}) + p(w_{i+j,n-i-j})) + p(w_{ik}) + p(w_{i+k,j-k}) = (g^{ij}_A - f^{ij}_A) + f^{ik}_B + f^{i+k,j-k}_C = g^{ik}_B$.

□

Theorem 6.6.2. *Algorithm 12 achieves generalized arc-consistency on the conjunction of a $CFGC$ and a linear objective function constraint. The algorithm requires cubic time and quadratic space in the number of variables.*

Proof. We show that value a is removed from D_i if and only if for all words $w = w_1 \dots w_n \in L_G \cap (D_1 \dots D_n)$ with $a = w_i$ it holds that $p(w) \leq T$.

- ⇒ (Soundness) Assume that value a is removed from D_i. Let $w \in L_G \cap (D_1 \dots D_n)$ with $w_i = a$. Due to the assumption that $w \in L_G$ there must exist a derivation $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ A_a\ w_{i+1} \dots w_n \underset{G}{\Rightarrow} w_1 \dots w_{i-1} w_i w_{i+1} \dots w_n$ for some $A_a \in N$ with $(A_a \rightarrow a) \in P$. Since a is being removed from D_i, we know that $g^{i1}_{A_a} \leq T$. According to Lemma 6.6.1, $p(w) \leq g^{i1}_{A_{w_i}} \leq T$.
- ⇐ (Completeness) Assume that for all $w = w_1 \dots w_n \in L_G \cap (D_1 \dots D_n)$ with $w_i = a$ it holds that $p(w) \leq T$. According to Lemma 6.6.1, this implies $g^{i1}_{A_a} \leq T$. Then, a is removed from the domain of X_i in step 6.

Regarding the time complexity, it is easy to verify that the workload is dominated by steps 2 and 5, both of which require time in $O(n^3|G|)$. The space complexity is dominated by the memorization of values f^{ij}_A and g^{ij}_A, and it is thus limited by $O(n^2|G|)$. □

6.7 Logic Combinations of Grammar Constraints

We define regular grammar constraints analogously to $CFGC$, but as in [149] we base it on automata rather than right-linear grammars:

Regular Grammar Constraint Given a finite automaton A and a right-linear grammar G with $L_A = L_G$, we set

$$RGC_A(X_1, \dots, X_n) := Grammar_G(X_1, \dots, X_n).$$

Efficient arc-consistency algorithms for RGCs have been developed in [116, 149]. Now equipped with efficient filtering algorithms for regular and context-free grammar constraints, in the spirit of [12, 126] we focus on certain questions that arise when a problem is modeled by logic combinations of these constraints. An important aspect when investigating logical combinations of grammar constraints is under what operations the given class of languages is closed. For example, when given a conjunction of regular grammar constraints, the question arises whether the conjunction of the constraints could not be expressed as one global RGC. This question can be answered affirmatively since the class of regular languages is known to be closed under intersection. In the following we summarize some relevant, well-known results for formal languages (see for instance [96]).

Lemma 6.7.1. *For every regular language L_{A^1} based on the finite automaton A^1 there exists a deterministic finite automaton A^2 such that $L_{A^1} = L_{A^2}$.*

Proof. Proof When $Q^1 = \{q_0, \dots, q_{n-1}\}$, we set $A^2 := (Q^2, \Sigma, \delta^2, q_0^2, F^2)$ with $Q^2 := 2^{Q^1}$, $q_0^2 = \{q_0^1\}$, $\delta^2 := \{(P, a, R) \mid R = \{r \in Q^1 \mid \exists\, p \in P : \ (p, a, r) \in \delta^1\}\}$, and $F^2 := \{P \subseteq Q^1 \mid \exists\, p \in P \cap F^1\}$. With this construction, it is easy to see that $L_{A^1} = L_{A^2}$. □

We note that the proof above gives a construction that can change the properties of the language representation, just like we had noted it earlier for context-free grammars that we had transformed into Chomsky Normal Form first before we could apply CYK for parsing and filtering. And just like we were faced with an exponential blow-up of the representation when bringing context-free grammars into normal-form, we see the same again when transforming a non-deterministic finite automaton of a regular language into a deterministic one.

Theorem 6.7.2. *Regular languages are closed under the following operations: Union, Intersection, and Complement.*

Proof. Proof Given two regular languages L_{A^1} and L_{A^2} with respective finite automata $A^1 = (Q^1, \Sigma, \delta^1, q_0^1, F^1)$ and $A^2 = (Q^2, \Sigma, \delta^2, q_0^2, F^2)$, without loss of generality, we may assume that the sets Q^1 and Q^2 are disjoint and do not contain symbol q_0^3.

- We define $Q^3 := Q^1 \cup Q^2 \cup \{q_0^3\}$, $\delta^3 := \delta^1 \cup \delta^2 \cup \{(q_0^3, a, q) \mid (q_0^1, a, q) \in \delta^1 \text{ or } (q_0^2, a, q) \in \delta^2)\}$, and $F^3 := F^1 \cup F^2$. Then, it is straight-forward to see that the automaton $A^3 := (Q^3, \Sigma, \delta^3, q_0^3, F^3)$ defines $L_{A^1} \cup L_{A^2}$.

- We define $Q^3 := Q^1 \times Q^2$, $\delta^3 := \{((q^1, q^2), a, (p^1, p^2) \mid \exists (q^1, a, p^1) \in \delta^1, (q^2, a, p^2) \in \delta^2\}$, and $F^3 := F^1 \times F^2$. The automaton $A^3 := (Q^3, \Sigma, \delta^3, (q_0^1, q_0^2), F^3)$ defines $L_{A^1} \cap L_{A^2}$.
- According to Lemma 6.7.1, we may assume that A^1 is a deterministic automaton. Then, $(Q^1, \Sigma, \delta^1, q_0^1, Q^1 \setminus F^1)$ defines $L_{A^1}^C$.

□

The results above suggest that any logic combination (disjunction, conjunction, and negation) of RGCs can be expressed as one global RGC. While this is true in principle, from a computational point of view, the size of the resulting automaton needs to be taken into account. In terms of disjunctions of RGCs, all that we need to observe is that the algorithm developed in [116] actually works with non-deterministic automata as well. In the following, denote by m an upper bound on the number of states in all automata involved, and denote the size of the alphabet Σ by D. We obtain our first result for disjunctions of regular grammar constraints:

Lemma 6.7.3. *Given RGCs $R_1, \ldots, R_k$, all over variables $X_1, \ldots, X_n$ in that order, we can achieve arc-consistency for the global constraint $\bigvee_i R_i$ in time $O((km+k)nD) = O(nDk)$ for automata with constant state-size m.*

If all that we need to consider are disjunctions of RGCs, then the result above is subsumed by the well known technique of achieving arc-consistency for disjunctive constraints which simply consists in removing, for each variable domain, the intersection of all values removed by the individual constraints. However, when considering conjunctions over disjunctions the result above is interesting as it allows us to treat a disjunctive constraint over RGCs as one new RGC of slightly larger size.

Now, regarding conjunctions of RGCs, we find the following result:

Lemma 6.7.4. *Given RGCs $R_1, \ldots, R_k$, all over variables $X_1, \ldots, X_n$ in that order, we can achieve arc-consistency for the global constraint $\bigwedge_i R_i$ in time $O(nDm^k)$.*

Finally, for the complement of a regular constraint, we have:

Lemma 6.7.5. *Given an RGC R* based on a deterministic automaton*, we can achieve arc-consistency for the constraint $\neg R$ in time $O(nDm) = O(nD)$ for an automaton with constant state-size.*

Proof. Proof Lemmas 6.7.3- 6.7.5 are an immediate consequence of the results in [116] and the constructive proof of Theorem 6.7.2. □

Note that the lemma above only covers RGCs for which we know a deterministic finite automaton. However, when negating a disjunction of regular grammar constraints, the automaton to be negated is non-deterministic. Fortunately, this problem can be entirely avoided: When the initial automata associated with

the elementary constraints of a logic combination of regular grammar constraints are deterministic, we can apply the rule of DeMorgan so as to only have to apply negations to the original constraints rather than the non-deterministic disjunctions or conjunctions thereof. With this method, we have:

Corollary 6.7.6. *For any logic combination (disjunction, conjunction, and negation) of deterministic $RGCs$ $R_1, \ldots, R_k$, all over variables $X_1, \ldots, X_n$ in that order, we can achieve generalized arc-consistency in time $O(nDm^k)$.*

Regarding logic combinations of context-free grammar constraints, unfortunately we find that this class of languages is not closed under intersection and complement, and the mere disjunction of context-free grammar constraints is not interesting given the standard methods for handling disjunctions. We do know, however, that context-free languages are closed under intersection with regular languages. Consequently, these conjunctions are tractable as well.

6.8 Limits of the Expressiveness of Grammar Constraints

So far we have been very careful to mention explicitly how the size of the state-space of a given automaton or how the size of the set of non-terminals of a grammar influences the running time of our filtering algorithms. From the theory of formal languages' viewpoint, this is rather unusual, since here the interest lies purely in the asymptotic runtime with respect to the word-length. For the purposes of constraint programming, however, a grammar may very well be generated on the fly and may depend on the word-length, whenever this can be done efficiently. This fact makes grammar constraints even more expressive and powerful tools from the modeling perspective. Consider for instance the context-free language $L = \{a^n b^n\}$ that is well-known not to be regular. Note that, within a constraint program, the length of the word is known — simply by considering the number of variables that define the scope of the grammar constraint. Now, by allowing the automaton to have $2n + 1$ states, we can express that the first n variables shall take the value a and the second n variables shall take the value b by means of a regular grammar constraint. Of course, larger automata also result in more time that is needed for propagation. However, as long as the grammar is polynomially bounded in the word-length, we can still guarantee a polynomial filtering time.

The second modification that we can safely allow is the reordering of variables. In the example above, assume the first n variables are $X_1, \ldots, X_n$ and the second n variables are $Y_1, \ldots, Y_n$. Then, instead of building an automaton with $2n+1$ states that is linked to $(X_1, \ldots, X_n, Y_1, \ldots, Y_n)$, we could also build an automaton with just two states and link it to $(X_1, Y_1, X_2, Y_2, \ldots, X_n, Y_n)$ (see Figure 6.3). The same ideas can also be applied to $\{a^n b^n c^n\}$ which is not even context-free but context-sensitive. The one thing that we really need to be careful about is that, when we want to exploit our earlier results on the combination of grammar constraints, we need to make sure that the ordering requirements specified in the respective

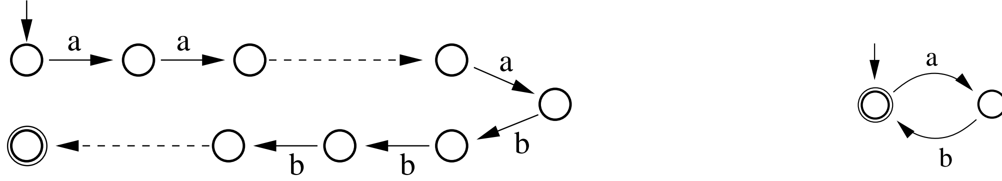


Figure 6.3: Regular grammar filtering for $\{a^n b^n\}$. The left figure shows a linear-size automaton, the right an automaton that accepts a reordering of the language.

theorems are met (see for instance Lemmas 6.7.3 and 6.7.4).

While these ideas can be exploited to model some required properties of solutions by means of grammar constraints, they make the theoretical analysis of which properties can or cannot be modeled by those constraints rather difficult. Where do the boundaries run between languages that are suited for regular or context-free grammar filtering? The introductory example, as uninteresting as it is from a filtering point of view, showed already that the theoretical tools that have been developed to assess that a certain language cannot be expressed by a grammar on a lower level in the Chomsky hierarchy fail. The well-known pumping lemmas for regular and context-free grammars for instance rely on the fact that grammars be constant in size. As soon as we allow reordering and/or non-constant size grammars, they do not apply anymore.

To be more formal: what we really need to consider for propagation purposes is not an entire infinite set of words that form a language, but just a slice of words of a given length. I.e., given a language L what we need to consider is just $L_{|n} := L \cap \Sigma^n$. Since $L_{|n}$ is a finite set, it really is a regular language. In that regard, our previous finding that $\{a^n b^n\}$ for fixed n can be modeled as regular language is not surprising. The interesting aspect is that we can model $\{a^n b^n\}$ by a regular grammar of size *linear in* n, or even of *constant size* when reordering the variables appropriately.

Suitedness for Grammar Filtering Given a language L over the alphabet Σ, we say that L is *suited for regular (or context-free) grammar filtering* if and only if there exist constants $k, n \in \mathbb{N}$ such that there exists a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and a finite automaton A (or normal-form context-free grammar G) such that both σ and A (G) can be constructed in time $O(n^k)$ with $\sigma(L_{|n}) = \sigma(L \cap \Sigma^n) := \{w_{\sigma(1)} \ldots w_{\sigma(n)} \mid w_1 \ldots w_n \in L\} = L_A$ ($\sigma(L_{|n}) = L_G$).

Remark Note that the previous definition implies that the size of the automaton (grammar) constructed is in $O(n^k)$. Note further that, if the given language is regular (context-free), then it is also suited for regular (context-free) grammar filtering.

Now, we have the terminology at hand to express that some properties cannot be modeled efficiently by regular or context-free grammar constraints. We start out by proving the following useful Lemma:

Lemma 6.8.1. *Denote with* $N = \{S_0, \ldots, S_r\}$ *a set of non-terminal symbols and* $G = (\Sigma, N, P, S_0)$ *a*

context-free grammar in Chomsky-Normal-Form. Then, for every word $w \in L_G$ of length n, there must exist $t, u, v \in \Sigma^*$ and a non-terminal symbol $S_i \in N$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} tS_iv$, $S_i \overset{*}{\underset{G}{\Rightarrow}} u$, $w = tuv$, and $n/4 \leq |u| \leq n/2$.

Proof. Proof Since $w \in L_G$, there exists a derivation $S_0 \overset{*}{\underset{G}{\Rightarrow}} w$ in G. We set $h_1 := 0$. Assume the first production used in the derivation of w is $S_{h_1} \to S_{k_1}S_{k_2}$ for some $0 \leq k_1, k_2 \leq r$. Then, there exist words $u_1, u_2 \in \Sigma^*$ such that $w = u_1u_2$, $S_{k_1} \overset{*}{\underset{G}{\Rightarrow}} u_1$, and $S_{k_2} \overset{*}{\underset{G}{\Rightarrow}} u_2$. Now, either u_1 or u_2 fall into the length interval claimed by the lemma, or one of them is longer than $n/2$. In the first case, we are done, the respective non-terminal has the claimed properties. Otherwise, if $|u_1| < |u_2|$ we set $h_2 := k_2$, else $h_2 := k_1$. Now, we repeat the argument that we just made for S_{h_1} for the non-terminal S_{h_2} that derives to the longer subsequence of w. At some point, we are bound to hit a production $S_{h_m} \to S_{k_m}S_{k_{m+1}}$ where S_{h_m} still derives to a subsequence of length greater than $n/2$, but both $S_{k_m}, S_{k_{m+1}}$ derive to subsequences that are at most $n/2$ letters long. The longer of the two is bound to have length greater than $n/4$, and the respective non-terminal has the desired properties. □

Now consider the language

$$L_{\text{AllDiff}} := \{w \in \mathbb{N}^* \mid \forall\, 1 \leq k \leq |w| :\ \exists\, 1 \leq i \leq |w| :\ w_i = k\}.$$

Since the word problem for L_{AllDiff} can be decided in linear space, L_{AllDiff} is (at most) context-sensitive.

Theorem 6.8.2. *L_{AllDiff} is not suited for context-free grammar filtering.*

Proof. Proof We observe that reordering the variables linked to the constraint has no effect on the language itself, i.e. we have that $\sigma(L_{\text{AllDiff}|n}) = L_{\text{AllDiff}|n}$ for all permutations σ. Now assume that, for all $n \in \mathbb{N}$, we could actually construct a minimal normal-form context-free grammar $G = (\{1, \dots, n\}, \{S_0, \dots, S_r\}, P, S_0)$ that generates $L_{\text{AllDiff}|n}$. We will show that the minimum size for G is exponential in n. Due to Lemma 6.8.1, for every word $w \in L_{\text{AllDiff}|n}$ there exist $t, u, v \in \{1, \dots, n\}^*$ and a non-terminal symbol S_i such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} tS_iv$, $S_i \overset{*}{\underset{G}{\Rightarrow}} u$, $w = tuv$, and $n/4 \leq |u| \leq n/2$. Now, let us count for how many words non-terminal S_i can be used in the derivation. Since from S_i we can derive u, all terminal symbols that are in u must appear in one block in any word that can use S_i for its derivation. This means that there can be at most $(n - |u|)(n - |u|)!(|u|)! \leq \frac{3n}{4}(\frac{n}{2}!)^2$ such words. Consequently, since there exist $n!$ many words in the language, the number of non-terminals is bounded from below by

$$r \quad \geq \quad \frac{n!}{\frac{3n}{4}(\frac{n}{2}!)^2} \quad = \quad \frac{4(n-1)!}{3(\frac{n}{2}!)^2} \quad \approx \quad \frac{4\sqrt{2}}{3\sqrt{\pi}}\frac{2^n}{n^{3/2}} \quad \in \quad \omega(1.5^n).$$

□

Now, the interesting question arises whether there exist languages at all that are fit for context-free, but not for regular grammar filtering? If this wasn't the case, then the algorithms developed in Sections 6.3 and 6.4 would be utterly useless. What makes the analysis of suitedness so complicated is the fact that the modeler has the freedom to change the ordering of variables that are linked to the grammar constraint — which essentially allows him or her to change the language almost ad gusto. We have seen an example for this earlier where we proposed that $a^n b^n$ could be modeled as $(ab)^n$.

Theorem 6.8.3. *The set of languages that are suited for context-free grammar filtering is a strict superset of the set of languages that are suited for regular grammar filtering.*

Proof. Proof Consider the language $L = \{ww^R\#vv^R \mid v, w \in \{0,1\}^*\} \subseteq \{0,1,\#\}^*$ (where x^R denotes the reverse of a word x). Obviously, L is context-free with the grammar $(\{0,1,\#\}, \{S_0, S_1\}, \{S_0 \to S_1\#S_1,\ S_1 \to 0S_10,\ S_1 \to 1S_11,\ S_1 \to \varepsilon\}, S_0)$. Consequently L is suited for context-free grammar filtering.

Note that, when the position $2k+1$ of the sole occurrence of the letter $\#$ is fixed, for every position i containing a letter 0 or 1, there exists a *partner position* $p^k(i)$ so that both corresponding variables are forced to take the same value. Crucial to our following analysis is the fact that, in every word $x \in L$ of length $|x| = n = 2l+1$, every even (odd) position is linked in this way exactly with *every* odd (even) position for some placement of $\#$. Formally, we have that $\{p^k(i) \mid 0 \le k \le l\} = \{1, 3, 5, \ldots, n\}$ ($\{p^k(i) \mid 0 \le k \le l\} = \{2, 4, 6, \ldots, 2l\}$) when i is even (odd).

Now, assume that, for every odd $n = 2l+1$, there exists a finite automaton that accepts some reordering of $L \cap \{0,1,\#\}^n$ under variable permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$. For a given position $2k+1$ of $\#$ (in the original ordering), by $dist_\sigma^k := \sum_{i=1,3,\ldots,2l+1} |\sigma(i) - \sigma(p^k(i))|$ we denote the total distance of the pairs after the reordering through σ. Then, the average total distance after reordering through σ is

$$\begin{aligned} \tfrac{1}{l+1} \textstyle\sum_{0 \le k \le l} dist_\sigma^k &= \tfrac{1}{l+1} \textstyle\sum_{0 \le k \le l} \sum_{i=1,3,\ldots,2l+1} |\sigma(i) - \sigma(p^k(i))| \\ &= \tfrac{1}{l+1} \textstyle\sum_{i=1,3,\ldots,2l+1} \sum_{0 \le k \le l} |\sigma(i) - \sigma(p^k(i))|. \end{aligned}$$

Now, since we know that every odd i has l even partners, even for an ordering σ that places all partner positions in the immediate neighborhood of i, we have that

$$\sum_{0 \le k \le l} |\sigma(i) - \sigma(p^k(i))| \ge 2 \sum_{s=1,\ldots,\lfloor l/2 \rfloor} s = (\lfloor l/2 \rfloor + 1)\lfloor l/2 \rfloor.$$

Thus, for sufficiently large l, the average total distance under σ is

$$\begin{aligned}
\frac{1}{l+1}\sum_{0\le k\le l} dist_\sigma^k &\ge \frac{1}{l+1}\sum_{i=1,3,\dots,2l+1}(\lfloor l/2\rfloor+1)\lfloor l/2\rfloor \\
&\ge \frac{l}{l+1}(\lfloor l/2\rfloor+1)\lfloor l/2\rfloor \\
&\ge l^2/8.
\end{aligned}$$

Consequently, for any given reordering σ, there must exist a position $2k+1$ for the letter $\#$ such that the total distance of all pairs of linked positions is at least the average, which in turn is greater or equal to $l^2/8$. Therefore, since the maximum distance is $2l$, there must exist at least $l/16$ pairs that are at least $l/8$ positions apart after reordering through σ. It follows that there exists an $1 \le r \le n$ such that there are at least $l/128$ positions $i \le r$ such that $p^k(i) > r$. Consequently, after reading r inputs, the finite automaton that accepts the reordering of $L \cap \{0, 1, \#\}^n$ needs to be able to reach at least $2^{l/128}$ different states. It is therefore not polynomial in size. It follows: L is not suited for regular grammar filtering. $\square$

6.9 Conclusion

We investigated the idea of basing constraints on formal languages. Particularly, we devised an incremental space- and time-efficient arc-consistency algorithm for grammar constraints based on context-free grammars in Chomsky Normal Form. For an entire sequence of monotonically tightening problems, we can now perform filtering in quadratic space and the same worst-case time as it takes to parse a context-free grammar by the Cooke-Younger-Kasami algorithm (CYK). We showed experimentally that the new algorithm is equally effective but massively faster in practice than its non-incremental counterpart. We further improved performance on this benchmark using our propagator for value selection. We also gave a new algorithm that performs cost-based filtering when a context-free grammar constraint occurs in combination with a linear objective function. This algorithm has again the same cubic worst-case complexity of CYK. Further research is needed to determine whether it is possible to devise an incremental version of this algorithm. We studied logic combinations of grammar constraints and showed where the boundaries run between regular, context-free, and context-sensitive grammar constraints when allowing non-constant grammars and reorderings of variables. Our hope is that grammar constraints can serve as powerful, highly expressive modeling entities for constraint programming in the future, and that our theory can help to better understand and tackle the computational problems that arise in the context of grammar constraint filtering.

CHAPTER SEVEN

Same-Relation Constraints

The ALLDIFFERENT constraint was one of the first global constraints [160] and it enforces the conjunction of one binary constraint, the not-equal constraint, for every pair of variables. By looking at the set of all pairwise not-equal relations at the same time, AllDifferent offers greater filtering power. The natural question arises whether we can generally leverage the knowledge that sets of pairs of variables all share the same relation. This chapter studies exactly this question. We study in particular special constraint graphs like cliques, complete bipartite graphs, and directed acyclic graphs, whereby we always assume that the *same* constraint is enforced on all edges in the graph. In particular, we study whether there exists a tractable GAC propagator for these global Same-Relation constraints and show that AllDifferent is a huge exception: most Same-Relation Constraints pose NP-hard filtering problems. We present algorithms, based on AC-4 and AC-6, for one family of Same-Relation Constraints, which do not achieve GAC propagation but outperform propagating each constraint individually in both theory and practice.

7.1 Introduction

The ALLDIFFERENT constraint was one of the first global constraints [160] and it enforces the conjunction of one binary constraint, the not-equal constraint, for every pair of variables. By looking at the set of all pairwise not-equal relations at the same time, AllDifferent offers greater filtering power while incurring the same worst-case complexity as filtering and propagating the effects of not-equal constraints for each pair of variables individually. The natural question arises whether we can leverage the knowledge that sets of pairs of variables all share the same relation in other cases as well. We investigate in particular binary constraint satisfaction problems (BCSPs) with special associated constraint graphs like cliques (as in AllDifferent), complete bipartite graphs (important when a relation holds between all variables X in a subset of I and Y in J), and directed acyclic graphs (apart from bounded tree width graphs the simplest generalization of trees), whereby we always assume that the *same* constraint is enforced on all edges in the graph. We refer to the conjunction of the same binary relation over any set of pairs in a BCSP as a *Same-Relation Constraint*.

7.2 Theory Background

We study the complexity of achieving GAC on binary CSPs with one same-relation constraint. The classes of structures considered are all constraint graphs as defined in [80]. The ultimate goal is to classify, for a given constraint graph, which same-relation constraints admit a polynomial-time GAC, and which do not. It is well known that the CSP is equivalent to the HOMOMORPHISM problem between relation structures [51]. Most

theoretical research has been done on the case where either the domain size or the arities of the constraints are bounded.

We deal with the problem where both the domain size and the arities of the constraints are unbounded (so-called *global constraints*). Since we are interested in the complexity of achieving GAC, we allow all unary constraints. In mathematical terms, we study the LIST HOMOMORPHISM problem. For a CSP instance $\mathcal{P}$, achieving GAC on $\mathcal{P}$ is equivalent to checking solvability of $\mathcal{P}$ with additional unary constraints [14]. Note that showing that a problem does not have a polynomial-time decision algorithm implies that this problem also does not have a polynomial-time algorithm for achieving GAC, which is a more general problem.

Generalising the result of Freuder [63], Dalmau et al. showed that CSPs with bounded treewidth modulo homomorphic equivalence are solvable in polynomial time [36]. Grohe showed[1] that this is the only tractable class of bounded arity, defined by structure [81]. In other words, a class of structures with unbounded treewidth modulo homomorphic equivalence is not solvable in polynomial time.

In the case of binary CSPs, we are interested in classes of constraint graphs with a same-relation constraint. The first observation is that classes of graphs with bounded treewidth are not only tractable, but also permit achieving GAC in polynomial time (even with separate domains, and even with different constraints: different domains are just unary constraints which do not increase the treewidth). The idea is that such graphs have bounded-size separating sets, and the domains on these separating sets can be explicitly recorded in polynomial space (dynamic programming approach) [39, 61, 81]. Therefore, we are interested in classes of graphs with unbounded treewidth.

7.3 Clique Same-Relation

First we look at cliques. The famous ALLDIFFERENT constraint is an example of a same-relation constraint which, if put on a clique, is tractable – in case of AllDifferent, because the microstructure is perfect [164] and also has a polynomial-time GAC [160].

Definition Given a set of values D and a set of variables $\{X_1, \dots, X_n\}$, each associated with its own domain $D_i \subseteq D$, and a binary relation $R \subseteq D \times D$, an assignment $\sigma : \{X_1, \dots, X_n\} \to D$ satisfies the *Clique Same-Relation Constraint* CSR on the relation R if and only if for all i and j such that $1 \leq i, j \leq n, i \neq j$, it holds that $(\sigma(X_i), \sigma(X_j)) \in R$.

[1]Assuming a standard assumption from parameterized complexity theory FPT $\neq$ W[1], see [48] for more details.

7.3.1 Complexity of Achieving GAC

Despite the tractability of AllDifferent, in general enforcing the same binary relation over all pairs of variables of a CSP yields a hard filtering problem.

Theorem 7.3.1. *Deciding whether CSR is satisfiable for an arbitrary binary relation is NP-hard.*

Proof. We reduce from the CLIQUE problem. Assume we are given an undirected graph $G = (V, E)$ and a value $k \in \mathbb{N}$ and need to decide whether G contains a clique of size k. We construct a CSP with just one CSR constraint in the following way. We introduce k variables $X_1, \ldots, X_k$, each associated with domain V. The relation R is defined as $R \leftarrow \{(a, b) \in V^2 \mid a \neq b, \{a, b\} \in E\}$. We claim that CSR on the relation $(X_1, \ldots, X_k, R)$ is satisfiable if and only if G contains a clique of size k.

"$\Rightarrow$" Assume there is an assignment $\sigma : \{X_1, \ldots, X_k\} \rightarrow V$ that satisfies CSR. Then, $C \leftarrow \{\sigma(X_1), \ldots, \sigma(X_k)\} \subseteq V$ is a clique because CSR enforces R for each pair of variables, and thus that there exists an edge between all pairs of nodes in C. Furthermore, $|C| = k$ since R forbids that the same node is assigned to two different variables.

"$\Leftarrow$" Now assume there exists a clique $C = \{v_1, \ldots, v_k\} \subseteq V$ with $|C| = k$. Setting $\sigma(X_i) \leftarrow v_i$ gives a satisfying assignment to CSR because for all $i \neq j$ we have that $(\sigma(X_i), \sigma(X_j)) = (v_i, v_j) \in E$ with $v_i \neq v_j$, and thus $(\sigma(X_i), \sigma(X_j)) \in R$.

□

Corollary 7.3.2. *Achieving GAC for the CSR is NP-hard.*

In fact, we can show more: Even when we limit ourselves to binary symmetric relations which, for each value, forbid *only one other value*, deciding the satisfiability of the CSR is already intractable. This shows what a great exception AllDifferent really is. Even the slightest generalization already leads to intractable filtering problems.

Theorem 7.3.3. *Deciding CSR is NP-hard even for relations where each value appears in at most one forbidden tuple.*

Proof. We reduce from SAT. Given a SAT instance with k clauses over n variables, we consider an instance of CSR with k variables, each corresponding to one clause. Let D be $\{\langle 1, T\rangle, \langle 1, F\rangle, \ldots, \langle n, T\rangle, \langle n, F\rangle\}$. We define $R \subseteq D \times D$ to be the binary symmetric relation which forbids, for every $1 \leq i \leq n$, the set of tuples $\{\langle\langle i, T\rangle, \langle i, F\rangle\rangle, \langle\langle i, F\rangle, \langle i, T\rangle\rangle\}$. Note that R is independent of the clauses in the SAT instance.

Each clause in the SAT instance is encoded into the domain restriction on the corresponding variable. For instance, the clause $(x_1 \vee \neg x_2 \vee x_3)$ encodes as the domain $\{\langle 1, T\rangle, \langle 2, F\rangle, \langle 3, T\rangle\}$.

Any solution to this CSR instance, which can contain at most one of $\langle i, T\rangle$ and $\langle i, F\rangle$ for any $1 \leq i \leq n$, gives a solution to the SAT instance (as each variable must be assigned a literal in its clause). SAT variables which are not assigned a value can be given any value without compromising satisfiability. Analogously, a feasible assignment to the SAT formula maps back to a satisfying assignment to CSR in the same way: in any clause, take any of the literals in the solution which satisfy that clause and assign the variable that value. □

7.3.2 Restriction on the Size of Domain

Our proof that CSR is intractable required both an increasing number of variables and increasing domain size. The question arises whether the problem becomes tractable when the domain size is limited. The following shows that the CSR is indeed tractable when the domain size is bounded.

Lemma 7.3.4. *For a constraint CSR for a symmetric relation R, it is possible to check if an assignment satisfies the constraint given only:*

- *a promise that any domain value* d *such that* $\langle d, d\rangle \notin R$*, is used at most once, and*
- *the set* S *of domain values assigned.*

By ensuring that there are no two distinct values $s_1, s_2 \in S$ *such that* $\langle s_1, s_2\rangle \notin R$*.*

Proof. If CSR is violated, there must be two variables which do not satisfy R. This could occur either because two variables are assigned the same domain value d such that the assignment $\langle d, d\rangle$ is forbidden by R, or two variables are assigned different values d_1, d_2 such that the tuple $\langle d_1, d_2\rangle$ which do not satisfy R. □

Lemma 7.3.4 provides a useful tool for characterizing the satisfying assignments to CSR, which we will use to devise a general filtering algorithm.

Theorem 7.3.5. *Achieving GAC for the CSR is tractable for bounded domains.*

Proof. Since the definition of CSR requires that the relation holds in both directions, it is sufficient to consider symmetric relations only. Then, Lemma 7.3.4 shows that satisfying assignments can be expressed by the set of allowed values and only using values d such that $\langle d, d\rangle$ is forbidden by R at most once. We shall show how given a CSR constraint, given a set of values S which satisfies Lemma 7.3.4 and a list of sub-domains for the variables in the scope of the constraint, we can find if an assignment with values only in S exists in polynomial time.

Given such a set of domain values S, we call values d such that $\langle d, d \rangle \in R$ *sink values*. Note that in any assignment which satisfies the CSR constraint and contains assignments only in S, changing the assignment of any variable to a sink value will produce another satisfying assignment. Therefore, without loss of generality, we can assume every variable which could be assigned any sink value in S is assigned such a value.

This leaves only variables whose domains contain only values which can occur at most once. This is exactly equivalent to an ALLDIFFERENT constraint, and can be solved as such.

Finally, note that for a domain of size d, there are 2^d subsets of the domain, and this places a very weak bound on the subsets of the domain which will satisfy the conditions of Lemma 7.3.4. Therefore, for any domain size there is a fixed bound on how many subsets have to be checked. □

Theorem 7.3.5 shows that achieving GAC for the CSR is tractable for bounded domains, although the algorithm presented here is not practical. There are a number of simple ways its performance could be improved which we will not consider here as we are merely interested in theoretical tractability.

An interesting implication of our result is the following. Consider a symmetric relation with at most k allowed tuples for each domain value; that is, given $R \subseteq D \times D$, we require that for each $d \in D$, $|\{\langle d, . \rangle \in R\}| \leq k$ for some k.

Corollary 7.3.6. *Let k be a bound on the number of allowed tuples involving each domain value in R. If k is bounded, then achieving GAC for the CSR is tractable. If k is unbounded, then solving CSR is NP-hard.*

Proof. If k is bounded, then after assigning a value to an arbitrary variable achieving GAC for the CSR reduces to the bounded domain case (see Theorem 7.3.5) . The unbounded case follows from Theorem 7.3.1. □

7.4 Bipartite Same-Relation

After studying complete constraint graphs in the previous section, let us now consider the complete bipartite case. This is relevant for CSPs where a set of variables is partitioned into two sets A and B and the same binary constraint is enforced between all pairs of variables possible between A and B.

Definition Given a set of values D and two sets of variables $A = \{X_1, \ldots, X_n\}$ and $B = \{X_{n+1} \ldots, X_m\}$, each associated with its own domain $D_i \subseteq D$, and a binary relation $R \subseteq D \times D$, an assignment $\sigma : \{X_1, \ldots, X_n\} \rightarrow D$ satisfies the *Bipartite Same-Relation Constraint* BSR on relation R if and only if $\forall\ X_i \in A, X_i \in B$ it holds that $(\sigma(X_i), \sigma(X_j)) \in R$.

7.4.1 Complexity of Achieving GAC

At first, the BSR appears trivially tractable because once an allowed assignment is found between any pair of variables in both parts of the bipartite graph, these values can be assigned to all variables. Indeed, as any bipartite graph (with at least one edge) is homomorphically equivalent to a single edge, such CSPs instance are easy to solve [36,81] (using the fact that CSPs are equivalent to the HOMOMORPHISM problem).

However, we have to take into account that the domains of the variables may be different; in other words, unary constraints are present. This fact causes the problem of achieving GAC for the BSR to become intractable. In mathematical terms, instead of facing a HOMOMORPHISM problem (which is trivial on bipartite graphs), we must deal with the LIST-HOMOMORPHISM problem.

Theorem 7.4.1. *Deciding whether BSR is satisfiable is NP-hard.*

Proof. We reduce from the CSR satisfaction problem which we showed previously is NP-hard. Assume we are given the CSR constraint on relation R over variables $\{X_1, \ldots, X_n\}$ with associated domains $D_1, \ldots, D_n$. We introduce variables $Y_1, \ldots, Y_n$, $Z_1, \ldots, Z_n$ and set $A \leftarrow \{Y_1, \ldots, Y_n\}$, $B \leftarrow \{Z_1, \ldots, Z_n\}$. The domain of variables Y_i and Z_i is $\{(i, k) \mid k \in D_i\}$. Finally we define the relation P over the tuples $((i, k), (j, l))$ where $1 \leq i, j \leq n$ and either $i = j \wedge k = l$ or $i \neq j \wedge (k, l) \in R$. We claim that BSR on A, B and P is satisfiable if and only if CSR on R and the X_i is.

"⇒" Let σ denote a solution to BSR on A,B and P. For all i the initial domains and the definition of P imply that $\sigma(Y_i) = \sigma(Z_i) = (i, k_i)$ for some $k_i \in D_i$. Define $\tau : \{X_1, \ldots, X_n\} \to D$ by setting $\tau(X_i) \leftarrow k_i$. Let $1 \leq i, j \leq n$ with $i \neq j$. Then, since $((i, k_i), (j, k_j)) \in P$, $(\tau(X_i), \tau(X_j)) = (k_i, k_j) \in R$. And therefore, τ satisfies CSR for the relation R.

"⇐" Let τ denote a solution to CSR on R and the X_i. Then, σ with $\sigma(Y_i) \leftarrow \sigma(Z_i) \leftarrow (i, \tau(X_i))$ satisfies BSR on A, B and P.

□

Corollary 7.4.2. *Achieving GAC for the BSR is NP-hard.*

7.5 DAG Same-Relation

In the previous sections we showed that achieving GAC for cliques and complete bipartite graphs is hard. Now we go on to show that a simple generalization of trees to directed graphs is intractable. When the binary relation that we consider is not symmetric, each edge in the constraint graph is directed. The generalization of trees (which we know are tractable) to the directed case then results in directed acyclic graphs

(DAGs).

Definition Let D be a set of values, X be a set of variables $X = \{X_1, \ldots, X_n\}$ and G be a directed acyclic graph (DAG) $G = \langle X, A\rangle$. Each variable is associated with its own domain $D_i \subseteq D$. Given a binary relation $R \subseteq D \times D$, an assignment $\sigma : \{X_1, \ldots, X_n\} \rightarrow D$ satisfies the *DAG Same-Relation Constraint* DSR on relation R if and only if $\forall\, 1 \leq i, j \leq n$ such that $(i, j) \in A$, it holds that $(\sigma(X_i), \sigma(X_j)) \in R$.

7.5.1 Complexity of Achieving GAC

Somewhat surprisingly, we find that even the simple graph structure of DAGs yields intractable filtering problems: bipartite graphs, with the orientation of all edges from one partition to the other, form a DAG. Therefore, Theorem 7.4.1 proves that solving DSR is NP-hard.

The question arises whether DAGs become tractable when we know that the direction on every arc is truly enforced by the constraints. Let us consider anti-symmetric relations. A relation R is anti-symmetric if for all a and b, $(a, b) \in R$ and $(b, a) \in R$ implies $a = b$. First we show that irreflexive antisymmetric relations can be NP-hard on DAGs.

Lemma 7.5.1. *Deciding satisfiability if DSR is NP-hard even for irreflexive antisymmetric relations.*

Proof. We use the equivalence between the CSP and the HOMOMORPHISM problem [51]. Solving an instance of DSR on relation R is equivalent to the question of whether there is a homomorphism[2] between the digraph A and digraph R. This problem is known as ORIENTED GRAPH COLORING [174]. The complexity of this problem was studied in [120], and Swart showed that the ORIENTED GRAPH COLORING problem is polynomial-time solvable for R on at most 3 vertices, and NP-complete otherwise, even when restricted to DAGs [180]. Note this proves more that almost all asymmetric relations are NP-hard on DAGs. □

Note that it follows[3] from a recent result of Hell et al. that solving DSR is NP-hard also for reflexive antisymmetric relations [88].

7.6 Grid Same-Relation

Another interesting class of graphs are *grids*. For $m, n \geq 1$, the $(m \times n)$-grid is the graph with vertex set $\{(i, j) | 1 \leq i \leq m, 1 \leq j \leq n\}$ and an edge between two different vertices (i, j) and (i', j') if $|i - i'| + |j - j'| = 1$.

[2]A homomorphism between two directed graphs (digraphs) $G = \langle V(G), A(G)\rangle$ and $H = \langle V(H), A(H)\rangle$ is a mapping $f : V(G) \rightarrow V(H)$ which preserves arcs, that is, $(u, v) \in A(G)$ implies $(f(u), f(v)) \in A(H)$.

[3]Private communication with A. Rafiey.

Definition Given a set of values D and a set of variables

$$\{X_{1,1}, \ldots, X_{1,m}, \ldots, X_{m,1}, \ldots, X_{m,n}\},$$

each associated with its own domain $D_i \subseteq D$, and a binary relation $R \subseteq D \times D$, an assignment σ : $\{X_{1,1}, \ldots, X_{m,n}\} \to D$ satisfies the *Grid Same-Relation Constraint GSR* on relation R if and only if for all distinct pairs of points (i, j) and (i', j') such that $|i - i'| + |j - j'| = 1$, it holds that $\sigma(X_{i,j}, X_{i',j'}) \in R$.

Once more, we find:

Lemma 7.6.1. *Deciding satisfiability of GSR is NP-hard.*

Proof. We reduce from the CLIQUE problem. Let $\langle G, k\rangle$ be an instance of CLIQUE, where $G = \langle V, E\rangle$ is an undirected graph without loops and $V = \{1, \ldots, n\}$. The goal is to determine whether there is a clique of size k in G. We define an instance of GSR with variables $X_{1,1}, \ldots, X_{k,k}$ and a relation R. For every $1 \leq i \leq k$, the domain of $X_{i,i}$ is $\{\langle i, i, u, u\rangle \mid 1 \leq u \leq n\}$. For every $1 \leq i \neq j \leq k$, the domain of $X_{i,j}$ is $\{\langle i, j, u, v\rangle \mid \{u, v\} \in E\}$. We define the relation R as follows: $\{\langle i, j, u, v\rangle, \langle i', j', u', v'\rangle\}$ belongs to R if and only if the following two conditions are satisfied:

1. $i = i' \Rightarrow [(u = u')\ \&\ (v \neq v')]$
2. $j = j' \Rightarrow [(v = v')\ \&\ (u \neq u')]$

We claim that G contains a clique of size k if and only if GSR on the $X_{i,j}$ and R is satisfiable.

"⇒" Assume there exists a clique $C = \{v_1, \ldots, v_k\} \subseteq V$ in G with $|C| = k$. We claim that setting $\sigma(X_{i,i}) \leftarrow \langle i, i, v_i, v_i\rangle$ for all $1 \leq i \leq k$, and $\sigma(X_{i,j}) \leftarrow \langle i, j, v_i, v_j\rangle$ for all $1 \leq i \neq j \leq k$ gives a satisfying assignment to GSR. Let $X_{i,j}$ and $X_{i',j'}$ be two variables such that $|i - i'| + |j - j'| = 1$. Let $\sigma(X_{i,j}) = \langle i, j, u, v\rangle$ and $\sigma(X_{i',j'}) = \langle i', j', u', v'\rangle$. If $i = i'$, then $u = u'$ and $v \neq v'$ from the definition of σ. If $j = j'$, then $v = v'$ and $u \neq u'$ from the definition of σ. Hence in both cases, $\{\langle i, j, u, v\rangle, \langle i', j', u', v'\rangle\} \in R$.

"⇐" Assume there is a solution σ to GSR on relation $(X_{1,1}, \ldots, X_{k,k}, R)$. From the definition of R, observe that for every fixed i there exists u_i such that $\sigma(X_{i,j}) = \langle i, j, u_i, .\rangle$ for every j. (In other words, the third argument of every row is the same.) Similarly, for every fixed j there exists v_j such that $\sigma(X_{i,j}) = \langle i, j, ., v_j\rangle$ for every i. (In other words, the fourth argument of every column is the same.) By these two simple observations, for every $1 \leq i \leq k$, there is no v and $j \neq j'$ such that $\sigma(X_{i,j}) = \langle i, j, u_i, v\rangle$ and $\sigma(X_{i,j'}) = \langle i, j', u_i, v\rangle$. (In other words, the fourth arguments of every row are all different.) Assume, for contradiction, that $\sigma(X_{i,j}) = \langle i, j, u_i, v\rangle$ and $\sigma(X_{i,j'}) = \langle i, j', u_i, v\rangle$ for some $v \neq u_i$ and $j' \neq j$; that is, the value v occurs more than once in the i-th row. But then $\sigma(X_{j',j'}) = \langle j', j', v, v\rangle$ as $X_{i,j'}$ and $X_{j',j'}$ are in the same column, and $\sigma(X_{j',j}) = \langle j', j, v, x\rangle$ as

$X_{j',j}$ and $X_{j',j'}$ are in the same row.. But as $X_{i,j}$ and $X_{j',j}$ are in the same column, and $\sigma(X_{i,j}) = \langle i, j, u_i, v\rangle$, we get $x = v$. In other words, $\sigma(X_{j',j}) = \langle j', j, v, v\rangle$. But this is a contradiction as σ would not be a solution to GSR. Similarly, we can prove the same results for columns. Moreover, the same argument shows that if $\sigma(X_{i,j}) = \langle i, j, u, v\rangle$, then $\sigma(X_{j,i}) = \langle j, i, v, u\rangle$. Using this kind of reasoning repeatedly shows that there is a set of k different values $C = \{u_1, \ldots, u_k\}$ such that $\delta(X_{i,j}) = \langle i, j, u_i, u_j\rangle$. From the definition of R, C forms a clique in G.

□

Corollary 7.6.2. *Achieving GAC for the GSR is intractable.*

7.7 Decomposing Same Relation Constraints

We proved a series of negative results for Same Relation Constraints (SRCs). Even for simple constraint graphs like DAGs and grids, SRCs pose intractable filtering problems. In this section, we investigate whether we can exploit the fact that the same relation is enforced on all edges of a constraint graph to achieve GAC on the corresponding binary CSP, where we consider the *collection* of individual binary constraints. This will achieve the same propagation as propagating each constraint in isolation, unlike for example the AllDiff constraint [160], which propagates the conjunction constraint. However, by making use of the added structure of SRC, we will show how both theoretical and practical performance gains can be achieved. We begin with the clique same-relation constraint.

7.7.1 Decomposing CSR

Using AC-4 [137] or any of its successors to achieve GAC, we require time $O(n^2d^2)$ for a network of n^2 binary constraints over n variables with domain size $|D| = d$. By exploiting the fact that the *same* relation holds for all pairs of variables, we can speed up this computation.

AC-4 Approach

We follow in principle the approach from AC-4 to count the number of supports for each value. The core observation is that a value l has the same number of supports $k \in D_i$ *no matter to which* D_j l *belongs*. Therefore, it is sufficient to introduce counters $supCount[i, l]$ in which we store the number of values in D_i which support $l \in D_j$ for any $j \neq i$. In Algorithm 13 we show how these counters are initialized at the root by counting the number of values in the domain of each variable that supports any given value l.

```
1: Init-CSR (X_1, ..., X_n, R)
2: for all l ∈ D do
3:    S_l ← {k ∈ D | (k, l) ∈ R}
4: end for
5: for all l ∈ D do
6:    for all 1 ≤ i ≤ n do
7:       supCount[i, l] ← |D_i ∩ S_l|
8:    end for
9: end for
```

Algorithm 13: Root-Node Initialization for the CSR Constraint

In Algorithm 14, we show how to filter the collection of binary constraints represented by the CSR so that we achieve GAC on the collection. The algorithm proceeds in two phases. In the first phase, lines 2-12, we update the counters based on the values that have been removed from variable domains since the last call to this routine. We assume that this incremental data is given in $\Delta_1, \ldots, \Delta_n$. For each value l that has lost all its support in some domain D_i as indicated by the corresponding counter $supCount[i, l]$ becoming 0, we add the tuple (i, l) to the set Q. The tuple means that l has to be removed from all D_j where $j \neq i$. In the second phase, lines 13-26, we iterate through Q to perform these removals and to update the counters accordingly. If new values must be removed as a consequence, they are added to Q.

Lemma 7.7.1. *Algorithm 14 achieves GAC on the collection of binary constraints represented by the CSR constraint in time $O(nd^2)$ and space $O(nd)$ where n is the number of variables and d is the size of the value universe D.*

Proof.

- **GAC:** The method is sound and complete as it initially counts all supports for a value and then and only then removes this value when all support for it is lost.
- **Complexity:** The space needed to store the counters is obviously in $\Theta(nd)$, which is linear in the input when the initial domains for all variables are stored explicitly. Regarding time complexity, the dominating steps are step 6 and step 19. Step 6 is carried out $O(nd^2)$ times. The number of times Step 19 is carried out is $O(nd)$ times the number of times that step 13 is carried out. However, step 13 can be called no more than $2d$ times as the same tuple (i, l) cannot enter Q more than twice: after a value $l \in D$ has appeared in two tuples in Q it is removed from all variable domains.

□

AC-6 Approach

As it is based on the same idea as AC-4, the previous algorithm is practically inefficient in that it always computes all supports for all values. We can improve the algorithm by basing it on AC-6 [11, 13] rather than

```
1:  Filter-CSR (X_1, ..., X_n, R, Δ_1, ..., Δ_n)
2:  Q ← ∅
3:  for all i = 1 ... n do
4:      for all k ∈ Δ_i do
5:          for all l ∈ S_k do
6:              supCount[i, l] ← supCount[i, l] − 1
7:              if supCount[i, l] == 0 then
8:                  Q ← Q ∪ {(i, l)}
9:              end if
10:         end for
11:     end for
12: end for
13: while Q ≠ ∅ do
14:     (i, l) ∈ Q, Q ← Q \ {(i, l)}
15:     for all j ≠ i do
16:         if l ∈ D_j then
17:             D_j ← D_j \ {l}
18:             for all k ∈ S_l do
19:                 supCount[j, k] ← supCount[j, k] − 1
20:                 if supCount[j, k] == 0 then
21:                     Q ← Q ∪ {(j, k)}
22:                 end if
23:             end for
24:         end if
25:     end for
26: end while
```

Algorithm 14: AC-4-based Filtering Algorithm for the CSR Constraint

AC-4. Algorithms 15-17 realize this idea. In Algorithm 15 we show how to modify the initialization part. First, the set of potential supports S_l for a value $l \in D$ is now an ordered tuple rather than a set. Second, support counters are replaced with support indices. The new variable $supIndex[i, l]$ tells us the index of the support in S_l which is currently supporting l in D_i. Algorithm 16 shows how to find a new support for a given value l from the domain of the variable i. The algorithm iterates through the ordered support list until it reaches to the end of it, line 6, in which case it returns a failure or until it finds a new support value k. Set-variable $T_{v,k}$ is used to store the set of values that are currently being supported by value k that is in the domain of variable v. In case a new support value l is found, $T_{v,k}$ is extended by the value l, line 10. These three new data structures, S_l, $supIndex[i, l]$ and $T_{v,k}$, allow us to quickly find values for which a new support needs to be computed, which can be done incrementally as a replacement support may only be found after the current support in the chosen ordering. This way, the algorithm never needs to traverse more than all potential supports for all values.

Finally, Algorithm 17 provides a general outline to our AC-6 based propagator which again works in two

```
1:  initSup (X_1, ..., X_n, R)
2:  for all l ∈ D do
3:      S_l ← (k ∈ D | (k, l) ∈ R)
4:  end for
5:  T ← ∅
6:  for all l ∈ D do
7:      for all 1 ≤ i ≤ n do
8:          supIndex[i, l] ← 0
9:          newSup(X_1, ..., X_n, R, i, l)
10:     end for
11: end for
```

Algorithm 15: Support Initialization for the CSR Constraint

phases, similar to its AC-4 based counter-part. In the first phase, lines 2-12, we scan the values that have been removed from variable domains since the last call to this routine. We assume that this incremental data is given in $\Delta_1, \ldots, \Delta_n$. In line 6, we look for a new support for each value l that was previously supported by a value k from the domain of variable i, which is now lost. If the value l is not supported anymore, it is removed from the set $T_{i,k}$ and the tuple (i,l) is added to the queue which means that l has to be removed from all D_j where $j \neq i$. In the second phase, lines 13-26, we iterate through Q to perform these removals and to update the set variables $T_{j,l}$ accordingly. If new values must be removed as a consequence, they are added to Q.

7.7.2 Decomposing BSR

Analogously to our results on the CSR Constraint, we can exploit again the knowledge that there is the same relation on all edges in the complete bipartite constraint graph. Again, the time that AC-4 or AC-6 would

```
1:  bool newSup (X_1, ..., X_n, R, i, l)
2:  supIndex[i, l] ← supIndex[i, l] + 1
3:  while supIndex[i, l] ≤ |S_l| and S_l[supIndex[i, l]] ∉ D_i do
4:      supIndex[i, l] ← supIndex[i, l] + 1
5:  end while
6:  if supIndex[i, l] > |S_l| then
7:      return false
8:  else
9:      k ← S_l[supIndex[i, l]]
10:     T_{v,k} ← T_{v,k} ∪ {l}
11:     return true
12: end if
```

Algorithm 16: Support Replacement for the CSR Constraint

```
1:  Filter-CSR (X_1, ..., X_n, R, Δ_1, ..., Δ_n)
2:  Q ← ∅
3:  for all i = 1 ... n do
4:      for all k ∈ Δ_i do
5:          for all l ∈ T_{i,k} do
6:              if !newSup(X_1, ..., X_n, R, i, l) then
7:                  Q ← Q ∪ {(i, l)}
8:                  T_{i,k} ← T_{i,k} \ {l}
9:              end if
10:         end for
11:     end for
12: end for
13: while Q ≠ ∅ do
14:     (i, l) ∈ Q, Q ← Q \ {(i, l)}
15:     for all j ≠ i do
16:         if l ∈ D_j then
17:             D_j ← D_j \ {l}
18:             for all k ∈ T_{j,l} do
19:                 if !newSup(X_1, ..., X_n, R, j, k) then
20:                     Q ← Q ∪ {(j, k)}
21:                     T_{j,l} ← T_{j,l} \ {k}
22:                 end if
23:             end for
24:         end if
25:     end for
26: end while
```

Algorithm 17: AC-6-based Filtering Algorithm for the CSR Constraint

need to achieve GAC on the collection of binary constraints that is represented by the BSR is in $O(n^2d^2)$. Following the same idea as for CSR, we can reduce this time to $O(nd^2)$.

We can devise an AC-6 based propagator for BSR in line with Algorithms 15-17. We can still use the set of potential supports S_l for a value which stores ordered tuples and support indices $supIndex[i, l]$ which tell us the index of the support in S_l which is currently supporting l in D_i. The only required modification is that set variables $T_{v,k}$ now has to be replaced with $T^A_{v,k}$ and $T^B_{v,k}$ to distinguish between the partitions of constraint graph.

Lemma 7.7.2. *Achieving GAC on the collection of binary constraints represented by the CSR constraint in time $O(nd^2)$ and space $O(nd)$ where n is the number of variables and d is the size of the value universe D.*

7.8 Numerical Results

The purpose of our experiments is to demonstrate the hypothesis that our CSR propagator brings substantial practical benefits, and that therefore this area of research has both theoretical as well as practical merit. To this end, we study two problems that can be modeled by the CSR. We show that filtering can be sped up significantly by exploiting the knowledge that all pairs of variables are constrained in the same way. Note that the traditional AC6 and the improved version for CSR achieve the exact same consistency, thus causing identical search trees to be explored. We therefore compare the time needed per choice point, without comparing how the overall model compares with the state-of-the-art for each problem as this is beyond the scope of this chapter.

We performed a number of comparisons between our AC-4 and AC-6 based algorithms, and found that the AC-6 algorithm always outperformed the AC-4 algorithm. This is not surprising, based both on our theoretical analysis, and previous work showing AC-6 outperforms AC-4 [11]. Because of this and space limitations we only present experiments using our AC-6 based algorithm.

All experiments are implemented using the Minion constraint solver on a Macbook with 4GB RAM and a 2.4GHz processor. These experiments show that CSR is a very robust and efficient propagator, never resulting in a slow-down in any of our experiments and producing over a hundred times speedup for larger instances.

Clique Same-Relation Constraint: The first problem we consider is the Stable Marriage Problem. It consists in pairing n men and n women into couples, so that no husband and wife in different couples prefer each other to their partner. We refer the reader to [65] for a complete definition and discussion of previous work on this problem. We use the hardest variant of this problem, where the men and women are allowed ties in their preference lists, and to give a list of partners they refuse to be married to under any circumstances. These two extensions make the problem NP-hard.

The standard model used in the CP literature keeps a decision variable for each man and woman where each domain consists of the indices of the corresponding preference list. The model then posts a constraint for each man-woman pair consisting of a set of no good pairs of values. We use the following, alternative model. Our model has one decision variable for each couple, whose domain is all possible pairings. We post between every pair of variables that the couples are 'stable', and also do not include any person more than once.

This model is inferior to specialized n-ary constraint for stable marriage problem [184], the intention is not to provide a new efficient model for the stable marriage problem. The reason we consider this model here is to show the benefits of using a CSR constraint in place of a clique of binary constraints.

		Couples								
		10	15	20	25	30	35	40	45	50
Probability	0.6	2.3	4.8	6.3	9.1	11.0	11.4	12.4	12.2	12.3
of attraction	0.9	3.8	4.9	6.5	6.9	8.6	9.2	10.1	11.4	12.9

Table 7.1: Average Speed-up for the Stable Marriage Problem.

The instances we consider are generated at random, with a fixed probability that any person will refuse to be married to another. This allows us to vary the number of tuples in the constraints. As the search space is the same size regardless of if we use our specialized CSR propagator or a clique of constraints (they achieve the same level of consistency after all), we show only the speed-up that our algorithm provides per search node. So, 2.0 means that our algorithm solved the problem twice as fast, or searched twice as many search nodes per second. The CSR propagator was never slower in any experiment we ran. For each instance we generated and solved 20 problems and took the average speed-up per choice point.

Table 7.1 shows the results. We note that CSR always provides a sizable improvement in performance, that only increases as problem instances get larger and harder, increasing up to over 10 times faster for larger instances.

The reason that the gain begins to reach a limit is that the size of the domains of the variable increases as the square of the number of people, meaning the cliques of size 50 have variables of domain 2500. Book-keeping work in the solver and algorithm for such large domains begins to dominate. Nevertheless, our algorithm is still over 10 times faster for these large problems.

Clique Same-Relation Constraint: The second problem we consider is the *Table Planning Problem*. The Table Planning Problem is the problem of sitting a group of people at tables, so that constraints about who will sit with each other are satisfied. Problems like this one often occur in the planning of events.

An instance of the Table Planning Problem (TPP) is a triple $\langle T, S, R \rangle$ where T is the number of tables and S is the size of each table. This implies there are $S \times T$ people to sit. R is a symmetric relation on the set $\{1, \ldots, S \times T\}$, which i is related to j if people i and j are willing to sit on the same table. A solution to the TPP therefore is a partition of people, denoted by the set $\{1, \ldots, S \times T\}$, where each part of the partition represents a table. Therefore in any solution each member of this partition must be of size S and all pairs of numbers within it must satisfy R.

We consider instances of TPP with three tables and where R is generated randomly, with some fixed probability of each edge, and its symmetric image, being added. The model we use is an $S \times T$ matrix, with each variable having domain $\{1, \ldots, S \times T\}$. The constraints are that each row (representing a table) has a clique of the constraint R and a single AllDifferent constraint on all the variables. We consider representing the cliques of the constraint R either as separate constraints, or using our propagator.

		People Per Table							
		30	40	50	60	70	80	90	100
Probability of edge	0.4	15	37	58	76	90	105	117	136
	0.5	11	33	51	66	80	81	83	91
	0.6	13	31	49	63	77	76	77	78
	0.8	7	18	21	27	33	34	36	38
	0.9	5	6	8	14	15	19	20	22

Table 7.2: Average Speed-up for the Table Planning Problem.

As we know that the size of search will be identical, regardless of how the cliques of R are implemented, we show only the speed-up achieved by our improved propagator. We run each of the following experiments for ten different randomly generated relations, and take an average of the speed-ups achieved. We measure speed-up based on the number of nodes per second searched in ten minutes, or how long it takes to find the first solution or prove no solution exists, whichever is fastest.

Our results are presented in Table 7.2. We observe large, scalable gains for larger problems, with over 20 times speed-up for the densest problems. For sparser constraints, we even achieve over 100 times speed-up using our CSR propagator instead of a clique of constraints. This shows again how well the CSR propagator scales for larger problems, achieving immense practical improvements.

7.9 Conclusion

We have looked at generalizing the famous AllDifferent to cliques of other constraints, and also other standard patterns such as bipartite, directed acyclic, and grid graphs. Unlike with the AllDifferent case, these constraints pose intractable filtering problems. By making use of the structure however, we can still provide substantial improvements in both theoretical and practical performance using new, generic algorithms. We have performed benchmarking across two problems using an AC-6 based decomposition algorithm on the CSR constraint. The experimental results show substantial gains in performance, proving that is worth exploiting the structure of same-relation constraints.

In the future, now we have laid down a theoretical framework, we will consider further benchmarks. In particular, we are interested to study how same-relation constraints interact with other global constraints.

Part V

Concluding Remarks

Chapter Eight

Related Work

All sciences characterize the essential nature of systems they study. These characterizations are invariably qualitative in nature, for they set the terms with which more detailed knowledge can be developed....

The study of logic and computers has revealed to us that intelligence resides in physical symbol systems. This is computer science's most basic law of qualitative structure. Symbol systems are collections of patterns and processes, the latter being capable of producing, destroying and modifying the former. The most important property is that they can designate objects, processes or other patterns, and that when they designate processes they can be interpreted....

A second law of qualitative structure for artificial intelligence is that symbol systems solve problems by generating potential solutions and testing them —that is by searching. Solutions are usually sought by creating symbolic expressions and modifying them sequentially until they satisfy the conditions for a solution.

— Newell and Simon, *Turing Award Lecture* (1976)

In their Turing award lecture, apart from the symbolic patterns to represent significant aspects of a problem domain, Newell and Simon argue that intelligent activity is achieved through the use of:

– Operations on symbolic patterns to generate potential solutions to problems.

– Search to select a solution from among these possibilities.

A major focus of research has been defining symbol structures and operations necessary for intelligent problem solving and developing strategies to efficiently and correctly search potential solutions generated by these structures and operations. These are the interrelated issues of inference and search; together, they are at the heart of many efficient combinatorial solvers. As such, our thesis in which we studied efficient search procedures for solving combinatorial problems is built upon many of the previous work done in this direction. Our goal is to make the connection with previous work with respect to each chapter of this thesis.

In Chapter 3, we studied dichotomic search which augments a feasibility solver to address an optimization problem by repeatedly partitioning the interval in which the possible optimal solution can lie. We addressed the trade-off between a typically much-smaller number of calls as in binary search versus much smaller cost of the calls as in linear search. This was previously studied in the context of binary search trees in [20]. The authors investigated skewed binary search trees. It was observed that a dominating factor over the running time for a search query in a binary search tree is the number of cache faults performed. Then, an appropriate memory layout of a binary search tree was claimed to have potential to reduce the number of

cache faults. Similar to observations in phase transition experiments [35, 91, 119, 138], that finding a satisfying solution and proving infeasibility incur different costs, the authors of [20] are motivated by the fact that, when conducting a search, branching to the left or right at a node does not necessarily have the same cost. They consider skewed binary search trees where the ratio between the size of the left subtree and the size of the tree is a fixed constant (a ratio of 1/2 gives perfect balanced trees). Experimental results showed that skewed binary search trees can perform better than perfect balanced search trees in various memory layouts. Interestingly, the authors conclude that the improvements in the running time are on the order of 15%, which is similar to gains that we observed in the context of solving optimization problems in Chapter 3. There are also binary search trees of bounded balance [145]. This class of binary search trees contain a parameter which can be varied to compromise between short search time and infrequent restructuring. There also exist self-adjusting binary trees [171]. An evaluation of self-adjusting binary search tree techniques can be found in [10].

Our dichotomic search protocol for constrained optimization can be seen as a dynamic/adaptive search strategy, and there exists a wealth of research on search algorithms that adopts itself dynamically as search progress. While conceptually our protocol is a complete search procedure, adaptive mechanism are studied extensively in local search approaches. In particular, the adaptive search algorithm was proposed as a generic, domain independent constraint-based local search method in [28, 29]. Another adaptive strategy is the greedy randomized adaptive search procedures (GRASP) which was introduced in [53, 85]. The details of this algorithm can be found in Section 2.4.3. There is a variant of GRASP which is known as reactive GRASP [152]. In this variant, the size of the restricted candidate list is dynamically adjusted depending on the quality of recently generated solutions. Similarly, the reactive variant of the tabu search algorithm [6] considers dynamically adjusting the search parameters based on the search history. In particular, the size of the tabu list is automatically updated when some configurations are repeated too often.

In Boolean satisfiability domain, dynamic local search algorithms are popular with adaptive extensions. The well-known WalkSAT algorithm [169] has an extension with an adaptive noise mechanism which dynamically adjusts the noise setting based on the time elapsed since the last improvement in the number of satisfied clauses has been achieved [94]. Similarly, the reactive scaling and probabilistic smoothing algorithm [98], assigns a clause penalty to each clause in boolean formula, and the search evaluation function is the sum of the clause penalties of unsatisfied clauses. The algorithm reactively changes the smoothing parameter during the search process whenever search stagnation is detected.

Next, we introduced dialectic search which is in close relation with many existing local search metaheuristics. The goal of merging thesis and antithesis in the best way so as to find an improving solution; namely the synthesis, can be viewed as an optimization problem. In that regard, dialectic search can be seen as an iterated local search(ILS) [177]. In fact, in the extreme case where the antithesis is defined as

modifying the thesis based on only one solution component, we reduce to an iterated local search algorithm. Notice also that, in that extreme case, the synthesis would become an antithesis that does degrade the solution value of the thesis. In the other extreme case, where the antithesis is defined as a random solution in the search space, dialectic search becomes a restarted greedy heuristic.

It is found empirically that optimization problems often exhibit a correlation between the fitness of local optima and their average distance to each other. This is known as the big valley structure [17]. The idea to merge thesis and antithesis aims at capturing this structure. Now, there might be multiple ways of how to merge two given solutions. In Section 4.2.2, we presented a procedure that computes a synthesis. This procedure closely resembles the path-relinking technique [71]. The fact that variables which have already been assigned their target value become tabu and are no longer allowed to change their values can be viewed as a kind of tabu search [6]. There is also a clear connection between the cross-over operator in genetic algorithms, where an intermediate solution is generated from parent solution, and our merge process to obtain a synthesis. When the problem of generating a synthesis is considered as an optimization problem itself, and the goal is to seek for the best possible combination of thesis and antithesis, then, dialectic search be related to variable neighborhood search [83], and very large scale neighborhood search [2, 181]. Conceptually, dialectic search is a single-point local search algorithm. As such, it differs from multi-point local search algorithms such as genetic algorithms and ant-colony optimization. However, one commonality between dialectic search and multi-point local search algorithms is the fact that they naturally lend to parallelization. For dialectic search algorithm, the phase where we generate antithesis solutions from the thesis can be parallelized so that the modification of the thesis and the merge process that follows, can be carried out for many different antithesis solutions simultaneously.

In terms of computational results on constraint satisfaction problems, after our paper [110], other researchers in the constraint programming community were also interested in solving the Costas arrays problem. Generating Costas arrays is a highly combinatorial problem, and the combination of `alldifferent` constraints admits only very few feasible solutions. This problem is related to three well-known CSPs: the *nqueens* problem, the *all-interval series* problem and the *Golomb's ruler* problem but it is much harder to solve. In [41] an adaptive search algorithm was proposed for the problem. This algorithm improves over our results presented in [110]. The authors conduct further experiments where they consider a parallel version of their adaptive search algorithm [42]. Experimental results showed that nearly linear speedups can be achieved on several hundreds of cores.

The work presented in Chapter 5 investigates variable and value selection heuristics, and it is solely based on the impact-based search strategy [158]. One important aspect of black-box solvers is a domain independent search procedure. Impact-based search (IBS) strategy was proposed to this end as a generic and robust

search algorithm. It is motivated by the strong branching and pseudo-cost branching concepts in mathematical programming. It works by averaging the observed search space reduction due to constraint propagation after each variable-value assignment, and favoring the variable-value assignments with high impact. IBS was shown to perform well across different classes of problems. Other impact measures have also been designed. An alternative impact definition is based on the solution counting technique which exploits the structure of global constraints in CP. This search algorithm is based on exact or estimated solution counts. Another general purpose search goal is known as conflict-directed heuristics which was proposed in [18]. This strategy is also known as *weighted-degree* heuristic. It uses weights for each constraint to identify variables which are the sources of inconsistencies in a problem. In the beginning, all constraints are given an initial weight of 1. Each time the constraint causes a failure during search, its weight is incremented by 1. The weighted degree of a variable is the sum of the weights of constraints that includes this variable in its scope, and has at least one other uninstantiated variable. The heuristic chooses the variable with largest weighted degree. The main drawback of this heuristic is that it has the least information available when making its most important choices, i.e., its first few selections. In fact the heuristic has no weight information, other than the degrees of the variables, up until at least one failure has occurred. We observed the same issue when incorporating variance information in impact-based search strategy. In order to overcome this problem, we computed both local impacts, i.e., the actual impact value of each variable-value assignment, and the node impacts which are the estimated impact values.

Recently, activity-based search is proposed in [135]. The idea behind this strategy is to count the activity of a variable during propagation. A counter for each variable is initialized by a probing process and updated during search. The key features of this algorithm are; first, the activity counters are independent of the variable domains, second, it is not based on specific constraints contrary to solution-centric approaches, and third, it does not treat every constraint equally as in weighted-degree heuristic which increments the counter of each constraint by 1, although only a subset of them might be related to the inconsistency. It would be interesting to further study whether our work on incorporation variance information could be extended to the activity-based search.

With respect to our work on grammar constraint in Chapter 6, the idea of specifying constraints based on formal languages has been studied previously for regular grammars in [9, 25, 116, 149]. The regular constraint specifies an assignment of a sequence of variables that forms a string from a regular language. Many constraints can be encoded as a `regular` constraint. One drawback of this constraint is that it might not be expressive enough to succinctly encode some specifications, or the resulting automata can be very large in some cases. This, in turn, causes extensive memory consumption and slows down the search [154]. `Grammar` constraints can be more expressive, but this expressiveness comes at the price of more expensive filtering algorithms.

In Chapter 6, we investigated a time-and space efficient incremental filtering algorithm for context-free grammar constraints. Another filtering algorithm based on AND/OR graphs is presented in [155]. This approach, however, might have extensive memory consumption that can be prohibitive for performance as was the case in our experiments on shift-scheduling benchmarks. There has been also some work done to impose restriction on the specification of the context-free grammar constraints to ensure some properties, such as, permitting linear parsing algorithms while being more expressive than regular grammars [114].

There is some work done in order to extend of grammar constraints for the cases that exhibit preferences among feasible solutions. There are two variants that address this setting. One approach is due to our work in Section 6.6 which investigates cost-based filtering when there is cost value associated with each value. The other approach is presented in [115] where each production rule is associated with a cost value. The latter approach is then extended to enforce the `soft_grammar` constraint [115]. Similarly, the `cost_regular` constraint is presented in [40]. In general, the combination of a global constraint with an algebraic constraint that guides the search toward improving solutions is commonly referred to as an *optimization constraint*. The task of achieving generalized arc-consistency is then often called *cost-based filtering* [58]. Optimization constraints and cost-based filtering play an essential role in constrained optimization and hybrid problem decomposition methods such as CP-based Lagrangian Relaxation [156] were studied. Inspired by the global constraints based on formal languages in CP domain, there has been some work proposed to use formal languages for modeling of such substructures in MIP to solve optimization problems [30]. Lastly, in a theoretical study, it was shown that how grammar constraints can be linearized in a way that the resulting polytope has only integer extreme points [117].

Finally, in Chapter 7 we studied whether there exists tractable propagators for the same-relation constraints on various constraint networks. The study of tractable cases has been an important line of research in CP. We can identify two general approaches. The first one is to identify forms of constraints that ensure tractability regardless of how they interact with each other [22, 51]. The work in this line has lead to identifying the algebraic property known as polymorphisms [103]. The second approach is to identify constraint networks which ensure tractability regardless of the forms of constraints that are enforced [38]. The latter approach has been used to characterize tractable cases of bounded-arity CSPs [36]. A special case of same-relation constraints is studied in [52]. In this work, the authors study CSPs with multiple `all_different` constraints whereby they consider assumptions such as the variables being linearly ordered so that all `all_different` constraints are defined over intervals of variables in this order.

Chapter Nine

Conclusions

9.1 Efficient Search Procedures for Solving Combinatorial Problems

In the introduction, we started with the following observation:

> On one hand, we have extremely difficult combinatorial problems due to their intrinsic complexity, and on the other hand, we have provably polynomial inference algorithms. As such, the solution process heavily depends on (exponential) search.

Assuming $P \neq NP$, this of course means that we cannot guarantee perfect choices during search. However, the important realization was that this is a worst-case statement, and the goal was to make very good choices on average. Accordingly, our claim was that there is an immense potential for improvement by boosting the average-case search performance.

We believe that we have covered important aspects related to search, and our results support the statement above. We developed efficient search procedures that can be used in a tree search approach, designed a dichotomic search protocol for constrained optimization, and introduced a novel local search meta-heuristic. We presented substantial performance gains on important problems from different domains; constraint satisfaction, continuous optimization, and constrained optimization. Below we summarize the main contributions of our work.

Part - I: Complete Search

We studied skewed binary search for constrained optimization. Under a cost model where negative trials incur more cost than positive trials, we proposed a search algorithm that is optimal in the expected and the worst case. We showed how to benefit from this theoretical study in practice using a heuristic search algorithm.

We believe this method has potential to change the way constraint systems perform binary search for finding optimal solutions. As long as an optimization system offers a timeout or fail limit facility, our dichotomic search protocol can be implemented fairly simply, and should provide substantial speed ups over binary search, and over improving the objective function one unit at a time.

Part - II: Incomplete Search

Next, we introduced dialectic search as a local search meta-heuristic. We showed how to develop simple yet effective stochastic local search algorithms for generating Costas arrays and for the set covering problem. The concept of dialectic search allowed us to develop functions for exploitation and exploration in separation.

We improved over the state-of-the-art set covering solvers and provided previously unknown bounds on well-known OR-Library instances. Hence, dialectic search stands as an appealing framework for devising highly efficient local search algorithms for practitioners working on constraints.

Part - III: Variable and Value Selection

We proposed a simple modification to impact-based search which improved the performance of this general-purpose search heuristic. Rather than considering the mean reduction only, we considered the idea of incorporating the variance in reduction. Experimental results showed that including variance can be beneficial in several cases. This is an easy enhancement to be considered for impact-based search implementations.

Part - IV: The Interplay between Search and Inference

Lastly, we studied the interaction between search and inference. Our incremental filtering algorithm for context-free grammar constraints bridges the gap between having a propagator in theory, and embedding it in an active tree search for solving problems in practice. We also showed how to use our propagator for guiding the value selection during search. Grammar constraints can serve as powerful, highly expressive modeling entities for constraint programming in the future, and the incremental algorithm we developed can help to tackle computational problems that arise in the context of grammar constraint filtering.

For binary constraint satisfaction problems, we considered various constraint graphs where the sets of pairs of variables all share the same relation. From a theoretical point of view, we proved that most same-relation constraints pose NP-hard filtering problems. From a practical point of view, we showed how to leverage this structure to develop generic filtering algorithms that can be used in a tree search approach.

We now conclude our thesis by giving advice to practitioners, and pointing out future directions for research.

9.2 General Advice for Practitioners

Based on our experience when dealing with various combinatorial problems, we provide some general advice to practitioners who would like to design search procedures for their problems.

9.2.1 Dichotomic Search Protocols

Search Interval: When dealing with constrained optimization problems, the gap between the upper and lower bound is of great practical importance. While it is desirable to have a smaller gap, our experience is that the skewing procedure cannot realize its full potential when the gap is too narrow. For example, a constrained optimization problem where the optimal should be 1 to 50, assume we are solving a map coloring problem with 50 nodes, the lower and upper end is too close to observe the effect of the skewing procedure. In such cases, skewed binary search and binary search consider almost identical trials.

Cost of Negative Trials: There are some issues that needs to be addressed when the theoretical model presented in Section 3.2 is to be applied in practice. First of all, failures do not generally incur costs that are a constant factor higher than those of positive trials. We could of course try to estimate such a ratio based on our experience with past trials. However, when the skewed search actually works well we hope to avoid negative trials as best as we can, so there will be very little statistical data to work with.

Our experience is that the Streeter-Smith strategy [175] neatly addresses this issues which is due to the disconnect between the theoretical model and the optimization practice. Using a fail-limit and considering potentially incomplete trials provide the necessary observations during search to estimate the cost of negative trials over positive trials. We can then employ our skewing procedure based on that cost ratio.

Fail-Limit: Another issue that is of importance for practical efficiency is regarding how to update the fail-limit parameter. The Streeter-Smith strategy doubles the fail-limit once the search interval is consumed and no improving solution is found. For the constraint satisfaction problems we considered, our observation is that updating fail-limits more rapidly, but in a linear fashion, yield better results. We have chosen to increase the fail-limit parameter after each inconclusive trial using a constant fail limit step.

Backtracking Search: In practice, we might hope to benefit from a clustering of good solutions. That is, once we find a new upper bound, there may be other solutions that further improve the upper bound and can be found quickly by investing more on search. We found this to be quite often the case when an improving solution is found. Hence, it is beneficial not to terminate the search immediately after a feasible solution is found, but instead to explore further within the current fail-limit.

Improving the Bounds: When CP-based approaches are used to deal with optimization problems, there is a problem with loose lower bounds. For example, in MIP approaches certain relaxations, e.g., the linear relaxation, can provide tight lower bounds. Missing such bounding procedures in CP, the lower bounds that we can compute may be so bad that we may not even strive to find and prove an optimal solution. The goal then may be to compute high quality solutions as quickly as possible.

9.2.2 Dialectic Search

While the general outline of dialectic search as presented in Section 4.2 leaves open doors on how to define the thesis, the antithesis, and the synthesis, ww find the following approach to work well in general.

The thesis can be initialized with a random solution, similar to many other local search algorithms. Applying a greedy heuristic to improve the initial thesis is often useful. It is not a concern to intensify the search aggressively, since the antithesis will provide the necessary exploration mechanism and force a randomized move in the solution space. After all, the global optimum is also a local optimum, hence, applying greedy heuristics help in finding improving solutions. In general, the antithesis is a randomized perturbation of parts of the thesis and the greedy improvement consists in moving to the best neighbor until no more such moves are available. When moving from the thesis to antithesis, considering the steps in a greedy fashion is also useful. The best solution found in this path; namely, synthesis, can again be improved with a greedy heuristic. Finally, it is important to also accept solutions that does not improve over the thesis, as long as they are equally good. This provides the search with the necessary freedom to move around neighboring solutions.

9.2.3 Impact-Based Search

It is important that impacts of variables are initialized before the search starts. In [158] the author presents different strategies on how impact values can be initialized. In general, it is costly to consider local impacts, i.e., the actual impact values found by committing a particular assignment. However, we found out that using local impacts pays off at the root node as the decisions made higher in tree are important.

9.2.4 Context-Free Grammar Constraints

While the size of the context-free grammar does not matter for the asymptotic worst-case complexity of the grammar constraint filtering, it does matter in practice. It is therefore beneficial to reduce the size of the grammar. For that purpose, we can apply minimization techniques from the theory of formal languages [96] before initiating the grammar constraints. Also, we noticed that in many rostering and scheduling benchmarks symmetries play a significant role. Consider for instance an employee scheduling problem with equally skilled employees. As we mentioned in Section 6.8, reordering of variables in the given sequence is one possibility to address this issue. Other than breaking symmetries, reordering can further help in reducing the size of the grammar, or the number of states of the finite state automaton in the case of `regular` constraint.

9.2.5 Clique-Same Relation Constraint

We found the propagator based on AC-6 algorithm for the clique-same relation constraint to be superior to its counterpart based on AC-4. We also note that, while linear speed up can be achieved in theory exploiting knowledge that every pair of variables share the same relation, the improvements seen in practice start to level out when the variables have domains of size more than two thousand values. This is because the book keeping in the algorithm (and potentially in the constraint solver) starts to dominate.

9.3 Future Work

We have presented significant theoretical and practical contributions and answered a variety of important questions. However many directions for future research remain unexplored. Search, in general, is a rich and fruitful field and offers opportunities for further improvement. We thus hope that other researchers will join in the quest for solving constraint satisfaction and constrained optimization problems, and advance the existing analysis, techniques, and applications. For future work, several interesting directions can be investigated. These include, but are not limited to:

- The main drawback when variance information is incorporated in impact-based search is that it has the least information available when making its most important choices, i.e., its first few selections. This is also a problem for other search strategies, see e.g., weighted degree heuristic [18]. This drawback can be addressed by combining the heuristic with restarting strategies such as a sampling process for gathering information. In principle this aims at learning with restarting in order to improve a non-restarted learning strategy.
- Conversely, as the search progresses we may want to base our guesses on recent variance information about the impact values, rather than considering old statistics gathered in the early stages of the search which might not be indicative anymore, or worse, might have deteriorating effect. For example, the recent activity-based search heuristics [135] relies on a decaying sum to forget the oldest statistics progressively. This idea can be adapted in our strategy.
- The idea of using statistical measures, such as variance, to improve the accuracy of search heuristics can be further applied to other general-purpose search algorithms. One possibility is to consider the recently introduced activity-based search heuristic [135].
- There is no clear understanding regarding the difference (or the similarity) between the variance seen in variable impacts and value impacts. It remains an open question whether both should be threated in the same way. The fact that we might get lucky sometimes and exploit good value orderings that

lead to feasible solutions further complicates the experimental results, and makes it difficult to make the distinction between the variable and value impacts. To reduce such side effects to a minimum, it would be interesting to work on almost satisfiable infeasible instances for which the whole search space needs to be exhausted.

- We note that the idea behind skewed binary search can be extended for the branch-and-bound algorithm which can be parametrized to improve bounds more aggressively. Based on the hardness of the current sub-problem the bounding procedure can consider aggressive or defensive bounds, and adjust itself dynamically during search.

- The dialectic search algorithm lends itself naturally to parallelization. We can generate many antithesis solutions upon the same thesis solution in parallel. The best solution found among simultaneous merge processes, becomes the starting point of the next parallel execution. Notice that each antithesis is an independent source of exploration and there is no communication between parallel computations except when all syntheses solutions are computed and one has to select the best one among them.

In conclusion, we believe that it is possible to achieve the same level of success and excellence in search as in inference. In its entirety, we consider the work done in this thesis as a contribution toward this goal, and hope that further research will enhance the development of efficient search procedures for solving challenging combinatorial problems that arise in the context of constraint satisfaction and constrained optimization.

APPENDIX A

A.1 Experimental Results – Set Covering Problem

We present the details of the experimental results given in Chapter 4 comparing dialectic search, HEGEL, with the iterative greedy algorithm, ITEG, from [132] and tabu search, TS, from [141] on 70 well-known Set Covering instances from the OR-Library [7]. These instances involve up to 400 items and 4000 sets. In order to compare with ITEG and TS which were developed for the uni-cost SCP, the costs of all sets are set to one.

ITEG was run on a multi-user Silicon Graphics IRIX Release 6.2 IP25, 194MHz MIPS R10000 processor and TS was run on a Pentium 4 with 2.4GHz. When comparing with ITEG, we use again our Pentium III 733MHz machine and we divide the cutoff times reported for ITEG by a factor of 4 which corresponds to the SPECint95 ratio of the two machines used. For the comparison with TS, we use an AMD Athlon 64 X2 Dual Core Processor 3800 2.0 GHz machine which is slightly slower than the machine used in [141].

Table A.1: Numerical Results for the Set Cover Problem. We present the average solution (standard deviation), best solution (standard deviation), average time to find the best solution, and the time limit used. Hegel was run 50 times on each instance, ITEG data were taken from [132] who ran their algorithms 10 times on each instance. The authors of ITEG do not report average runtime for each instance which is denoted with a dash.

	AvgSol		BestSol		AvgTime		TimeLimit		
Class	ITEG	HEGEL	ITEG	HEGEL	ITEG	HEGEL	ITEG	HEGEL	Speedup
scpa1	39.10	**39.02** (0.14)	39.00	39.00	-	2.12 (1.82)	7.50	7.50	1.00
scpa2	39.10	**39.06** (0.24)	39.00	39.00	-	1.56 (1.00)	7.50	7.50	1.00
scpa3	39.00	39.00 (0.00)	39.00	39.00	-	0.66 (0.64)	7.50	7.50	1.00
scpa4	38.00	38.00 (0.00)	38.00	38.00	-	1.94 (1.47)	7.50	7.50	1.00
scpa5	**38.70**	38.78 (0.41)	38.00	38.00	-	1.61 (1.87)	7.50	7.50	1.00
scpb1	22.00	22.00 (0.00)	22.00	22.00	-	0.40 (0.11)	15.00	2.50	6.00
scpb2	22.00	22.00 (0.00)	22.00	22.00	-	0.23 (0.05)	15.00	2.50	6.00
scpb3	22.00	22.00 (0.00)	22.00	22.00	-	0.36 (0.19)	15.00	2.50	6.00
scpb4	**22.00**	22.04 (0.20)	22.00	22.00	-	0.78 (0.45)	15.00	2.50	6.00
scpb5	22.20	**22.00** (0.00)	22.00	22.00	-	0.57 (0.30)	15.00	2.50	6.00
scpc1	43.50	**43.40** (0.49)	43.00	43.00	-	3.37 (2.74)	10.00	10.00	1.00
scpc2	**43.50**	43.70 (0.46)	43.00	43.00	-	2.23 (2.88)	10.00	10.00	1.00
scpc3	43.60	**43.36** (0.48)	43.00	43.00	-	3.30 (2.60)	10.00	10.00	1.00
scpc4	43.10	**43.04** (0.20)	43.00	43.00	-	4.10 (2.30)	10.00	10.00	1.00
scpc5	**43.50**	43.70 (0.46)	43.00	43.00	-	2.16 (2.05)	10.00	10.00	1.00
scpd1	25.00	**24.34** (0.47)	25.00	**24.00**	-	1.51 (1.29)	27.50	5.00	5.50
scpd2	25.00	25.00 (0.00)	25.00	25.00	-	0.36 (0.07)	27.50	5.00	5.50
scpd3	25.00	**24.98** (0.14)	25.00	**24.00**	-	0.48 (0.48)	27.50	5.00	5.50
scpd4	25.00	**24.98** (0.14)	25.00	**24.00**	-	0.73 (0.33)	27.50	5.00	5.50
scpd5	25.00	25.00 (0.00)	25.00	25.00	-	0.67 (0.28)	27.50	5.00	5.50
scpe1	5.00	5.00 (0.00)	5.00	5.00	-	0.00 (0.00)	2.50	0.10	25.00
scpe2	5.00	5.00 (0.00)	5.00	5.00	-	0.01 (0.01)	2.50	0.10	25.00
scpe3	5.00	5.00 (0.00)	5.00	5.00	-	0.01 (0.01)	2.50	0.10	25.00
scpe4	5.00	5.00 (0.00)	5.00	5.00	-	0.01 (0.01)	2.50	0.10	25.00
scpe5	5.00	5.00 (0.00)	5.00	5.00	-	0.01 (0.01)	2.50	0.10	25.00
scp41	**38.00**	38.82 (0.38)	38.00	38.00	-	0.45 (0.56)	2.50	2.50	1.00
scp42	37.00	37.00 (0.00)	37.00	37.00	-	0.24 (0.17)	2.50	2.50	1.00

scp43	38.00	38.00 (0.00)	38.00	38.00	-	0.27 (0.15)	2.50	2.50	1.00
scp44	**39.10**	39.30 (0.50)	39.00	**38.00**	-	0.80 (0.64)	2.50	2.50	1.00
scp45	**38.00**	38.58 (0.49)	38.00	38.00	-	0.74 (0.71)	2.50	2.50	1.00
scp46	**37.80**	37.86 (0.35)	37.00	37.00	-	0.49 (0.64)	2.50	2.50	1.00
scp47	**38.40**	38.94 (0.47)	38.00	38.00	-	0.94 (0.68)	2.50	2.50	1.00
scp48	**37.70**	37.88 (0.32)	37.00	37.00	-	0.68 (0.46)	2.50	2.50	1.00
scp49	**38.10**	38.88 (0.32)	38.00	38.00	-	0.38 (0.49)	2.50	2.50	1.00
scp410	**38.60**	39.00 (0.00)	**38.00**	39.00	-	0.70 (0.48)	2.50	2.50	1.00
scp51	**34.90**	34.98 (0.14)	34.00	34.00	-	0.39 (0.29)	2.50	2.50	1.00
scp52	**34.70**	34.92 (0.27)	34.00	34.00	-	0.45 (0.43)	2.50	2.50	1.00
scp53	**34.00**	34.16 (0.37)	34.00	34.00	-	0.85 (0.64)	2.50	2.50	1.00
scp54	**34.00**	34.10 (0.30)	34.00	34.00	-	1.02 (0.59)	2.50	2.50	1.00
scp55	**34.10**	34.32 (0.47)	34.00	34.00	-	0.86 (0.63)	2.50	2.50	1.00
scp56	34.50	34.50 (0.50)	34.00	34.00	-	0.83 (0.70)	2.50	2.50	1.00
scp57	**34.00**	34.04 (0.20)	34.00	34.00	-	0.68 (0.46)	2.50	2.50	1.00
scp58	34.90	**34.28** (0.45)	34.00	34.00	-	1.03 (0.64)	2.50	2.50	1.00
scp59	**35.00**	35.14 (0.35)	35.00	35.00	-	0.85 (0.65)	2.50	2.50	1.00
scp510	**34.60**	34.72 (0.45)	34.00	34.00	-	0.66 (0.50)	2.50	2.50	1.00
scp61	21.00	21.00 (0.00)	21.00	21.00	-	0.09 (0.03)	15.00	2.50	6.00
scp62	20.30	**20.02** (0.14)	20.00	20.00	-	0.59 (0.52)	15.00	2.50	6.00
scp63	21.00	21.00 (0.00)	21.00	21.00	-	0.09 (0.03)	15.00	2.50	6.00
scp64	21.00	**20.76** (0.43)	21.00	**20.00**	-	0.38 (0.60)	15.00	2.50	6.00
scp65	21.00	21.00 (0.00)	21.00	21.00	-	0.11 (0.06)	15.00	2.50	6.00
scpnre1	17.00	17.00 (0.00)	17.00	17.00	-	0.48 (0.13)	8.50	1.00	8.50
scpnre2	17.00	17.00 (0.00)	17.00	17.00	-	0.45 (0.09)	8.50	1.00	8.50
scpnre3	17.00	17.00 (0.00)	17.00	17.00	-	0.37 (0.01)	8.50	1.00	8.50
scpnre4	17.00	17.00 (0.00)	17.00	17.00	-	0.38 (0.03)	8.50	1.00	8.50
scpnre5	17.20	**17.00** (0.00)	17.00	17.00	-	0.46 (0.16)	8.50	1.00	8.50
scpnrf1	**10.30**	10.40 (0.49)	10.00	10.00	-	0.57 (0.20)	16.50	1.00	16.50
scpnrf2	**10.40**	10.42 (0.49)	10.00	10.00	-	0.61 (0.25)	16.50	1.00	16.50
scpnrf3	10.60	**10.54** (0.50)	10.00	10.00	-	0.52 (0.15)	16.50	1.00	16.50
scpnrf4	10.50	**10.46** (0.50)	10.00	10.00	-	0.56 (0.21)	16.50	1.00	16.50
scpnrf5	10.70	**10.36** (0.48)	10.00	10.00	-	0.60 (0.21)	16.50	1.00	16.50
scpnrg1	62.40	**62.22** (0.54)	62.00	**61.00**	-	2.86 (0.90)	6.50	5.00	1.30
scpnrg2	62.50	**62.20** (0.45)	62.00	**61.00**	-	2.99 (1.11)	6.50	5.00	1.30

scpnrg3	62.80	**62.70** (0.50)	62.00	62.00	-	2.83 (1.03)	6.50	5.00	1.30
scpnrg4	63.70	**62.84** (0.42)	62.00	62.00	-	2.93 (0.99)	6.50	5.00	1.30
scpnrg5	**62.70**	62.82 (0.43)	62.00	62.00	-	2.70 (0.91)	6.50	5.00	1.30
scpnrh1	34.80	**34.50** (0.50)	34.00	34.00	-	1.60 (0.53)	15.00	2.50	6.00
scpnrh2	34.70	**34.48** (0.50)	34.00	34.00	-	1.51 (0.38)	15.00	2.50	6.00
scpnrh3	34.80	**34.42** (0.49)	34.00	34.00	-	1.73 (0.60)	15.00	2.50	6.00
scpnrh4	35.00	**34.56** (0.50)	34.00	34.00	-	1.58 (0.52)	15.00	2.50	6.00
scpnrh5	34.60	**34.48** (0.50)	34.00	34.00	-	1.64 (0.54)	15.00	2.50	6.00

Table A.2: Numerical Results for the Set Cover Problem. We present the average runtime (standard deviation) in seconds for finding the best solution in each run, as well as the average solution quality and the best solution quality. Hegel was run 50 times on each instance and TS data were taken from [141] who ran their algorithms 10 times on each instance.

	AvgSol		BestSol		AvgTime		
Class	TS	HEGEL	TS	HEGEL	TS	HEGEL	Speedup
scpa1	**39.00** (0.00)	39.02 (0.14)	39.00	39.00	3.20 (2.90)	**2.11** (1.81)	1.52
scpa2	**39.00** (0.00)	39.04 (0.20)	39.00	39.00	4.70 (3.10)	**1.70** (1.31)	2.76
scpa3	39.10 (0.30)	**39.00** (0.00)	39.00	39.00	1.80 (1.80)	**0.66** (0.64)	2.74
scpa4	**37.80** (0.40)	38.00 (0.00)	**37.00**	38.00	5.70 (6.60)	**1.95** (1.48)	2.92
scpa5	**38.40** (0.50)	38.68 (0.47)	38.00	38.00	6.10 (4.50)	**2.37** (2.77)	2.57
scpb1	22.00 (0.00)	22.00 (0.00)	22.00	22.00	8.30 (8.80)	**0.40** (0.11)	20.84
scpb2	22.00 (0.00)	22.00 (0.00)	22.00	22.00	2.00 (4.20)	**0.23** (0.05)	8.64
scpb3	22.00 (0.00)	22.00 (0.00)	22.00	22.00	1.10 (3.50)	**0.36** (0.19)	3.02
scpb4	22.10 (0.30)	**22.00** (0.00)	22.00	22.00	11.60 (9.50)	**0.88** (0.58)	13.21
scpb5	22.00 (0.00)	22.00 (0.00)	22.00	22.00	12.10 (8.90)	**0.57** (0.30)	21.13
scpc1	43.50 (0.40)	**43.40** (0.49)	43.00	43.00	5.90 (5.70)	**3.37** (2.74)	1.75
scpc2	**43.40** (0.50)	43.70 (0.46)	43.00	43.00	9.50 (8.50)	**2.24** (2.89)	4.24
scpc3	43.40 (0.50)	**43.36** (0.48)	43.00	43.00	10.20 (10.40)	**3.29** (2.58)	3.10
scpc4	43.30 (0.50)	**43.04** (0.20)	43.00	43.00	11.60 (9.20)	**4.10** (2.30)	2.83
scpc5	43.90 (0.30)	**43.68** (0.47)	43.00	43.00	**2.10** (2.00)	2.32 (2.31)	0.91
scpd1	25.10 (0.30)	**24.12** (0.32)	25.00	**24.00**	6.40 (10.70)	**3.00** (2.57)	2.13
scpd2	25.00 (0.00)	25.00 (0.00)	25.00	25.00	2.20 (1.40)	**0.36** (0.07)	6.15
scpd3	**24.90** (24.90)	24.98 (0.14)	24.00	24.00	21.60 (27.00)	**0.48** (0.48)	45.08
scpd4	25.00 (0.00)	**24.98** (0.14)	25.00	**24.00**	17.70 (16.40)	**0.73** (0.33)	24.37
scpd5	25.00 (0.00)	25.00 (0.00)	25.00	25.00	24.10 (16.40)	**0.67** (0.28)	35.85
scpe1	5.00 (0.00)	5.00 (0.00)	5.00	5.00	**0.00** (0.00)	0.01 (0.01)	0.00
scpe2	5.00 (0.00)	5.00 (0.00)	5.00	5.00	**0.00** (0.00)	0.01 (0.00)	0.00
scpe3	5.00 (0.00)	5.00 (0.00)	5.00	5.00	**0.00** (0.00)	0.01 (0.00)	0.00
scpe4	5.00 (0.00)	5.00 (0.00)	5.00	5.00	**0.00** (0.00)	0.00 (0.00)	0.00
scpe5	5.00 (0.00)	5.00 (0.00)	5.00	5.00	**0.00** (0.00)	0.01 (0.00)	0.00
scp41	**38.10** (0.30)	38.50 (0.50)	38.00	38.00	**0.50** (0.70)	2.23 (2.83)	0.22
scp42	37.00 (0.00)	37.00 (0.00)	37.00	37.00	**0.00** (0.00)	0.24 (0.17)	0.00

scp43	38.00 (0.00)	38.00 (0.00)	38.00	38.00	**0.00** (0.00)	0.27 (0.15)	0.00
scp44	**38.60** (0.50)	38.94 (0.24)	38.00	38.00	**0.70** (1.10)	2.09 (1.67)	0.34
scp45	**38.00** (0.00)	38.16 (0.37)	38.00	38.00	**0.40** (1.00)	2.70 (2.42)	0.15
scp46	**37.20** (0.40)	37.74 (0.44)	37.00	37.00	**0.80** (0.90)	1.32 (2.28)	0.61
scp47	**38.40** (0.50)	38.62 (0.49)	38.00	38.00	**1.10** (0.70)	2.37 (2.54)	0.46
scp48	37.60 (0.50)	**37.56** (0.50)	37.00	37.00	**1.00** (1.30)	2.39 (2.78)	0.42
scp49	**38.00** (0.00)	38.54 (0.50)	38.00	38.00	**1.00** (1.20)	2.12 (2.70)	0.47
scp410	**38.30** (0.50)	39.00 (0.00)	**38.00**	39.00	1.20 (1.40)	**0.69** (0.48)	1.73
scp51	**34.70** (0.50)	34.84 (0.37)	34.00	34.00	**1.00** (2.20)	1.16 (2.14)	0.86
scp52	**34.20** (0.40)	34.62 (0.49)	34.00	34.00	3.20 (2.60)	**2.01** (2.65)	1.59
scp53	34.00 (0.00)	34.00 (0.00)	34.00	34.00	**0.80** (1.40)	1.42 (1.27)	0.56
scp54	34.00 (0.00)	34.00 (0.00)	34.00	34.00	1.60 (2.00)	**1.38** (1.07)	1.16
scp55	34.10 (0.30)	**34.00** (0.00)	34.00	34.00	**2.20** (3.00)	2.35 (2.11)	0.93
scp56	34.10 (0.30)	**34.06** (0.24)	34.00	34.00	3.10 (3.00)	**3.09** (2.55)	1.00
scp57	34.00 (0.00)	34.00 (0.00)	34.00	34.00	**0.60** (1.10)	0.78 (0.61)	0.77
scp58	34.40 (0.50)	**34.00** (0.00)	34.00	34.00	2.20 (3.60)	**1.94** (1.35)	1.13
scp59	35.60 (1.00)	**35.00** (0.00)	35.00	35.00	**0.60** (1.00)	1.31 (1.10)	0.46
scp510	34.50 (0.50)	**34.26** (0.44)	34.00	34.00	3.40 (3.60)	**3.04** (2.84)	1.12
scp61	21.00 (0.00)	21.00 (0.00)	21.00	21.00	**0.00** (0.00)	0.10 (0.03)	0.00
scp62	20.00 (0.00)	20.00 (0.00)	20.00	20.00	0.70 (0.80)	**0.64** (0.59)	1.10
scp63	21.00 (0.00)	21.00 (0.00)	21.00	21.00	**0.00** (0.00)	0.09 (0.03)	0.00
scp64	20.90 (0.30)	**20.28** (0.45)	20.00	20.00	**0.60** (1.90)	2.97 (2.95)	0.20
scp65	21.00 (0.00)	21.00 (0.00)	21.00	21.00	**0.00** (0.00)	0.11 (0.06)	0.00
scpnre1	17.30 (0.50)	**17.00** (0.00)	17.00	17.00	2.20 (3.00)	**0.48** (0.13)	4.60
scpnre2	17.10 (0.30)	**16.98** (0.14)	17.00	**16.00**	1.50 (2.30)	**0.54** (0.58)	2.80
scpnre3	17.10 (0.30)	**17.00** (0.00)	17.00	17.00	16.50 (38.80)	**0.37** (0.01)	44.62
scpnre4	17.20 (0.40)	**16.96** (0.20)	17.00	**16.00**	5.00 (7.80)	**0.65** (1.35)	7.69
scpnre5	17.00 (0.00)	17.00 (0.00)	17.00	17.00	4.50 (4.60)	**0.46** (0.16)	9.82
scpnrf1	10.70 (0.50)	**10.00** (0.00)	10.00	10.00	17.30 (28.90)	**1.20** (0.76)	14.37
scpnrf2	10.50 (0.50)	**10.00** (0.00)	10.00	10.00	43.90 (63.30)	**1.39** (1.00)	31.69
scpnrf3	10.60 (0.50)	**10.00** (0.00)	10.00	10.00	48.70 (112.60)	**1.36** (0.78)	35.77
scpnrf4	10.70 (0.50)	**10.00** (0.00)	10.00	10.00	17.90 (31.50)	**1.21** (0.84)	14.74
scpnrf5	10.60 (0.50)	**10.00** (0.00)	10.00	10.00	29.40 (73.50)	**1.15** (0.85)	25.63
scpnrg1	62.40 (0.80)	**61.94** (0.47)	61.00	61.00	27.30 (24.10)	**4.01** (2.01)	6.81
scpnrg2	62.30 (0.50)	**61.92** (0.34)	62.00	**61.00**	29.80 (34.40)	**4.26** (2.03)	7.00

scpnrg3	62.90 (0.60)	**62.36** (0.56)	62.00	**61.00**	20.80 (24.50)	**4.33** (2.16)	4.80
scpnrg4	63.10 (0.70)	**62.50** (0.50)	62.00	62.00	41.80 (42.70)	**4.59** (2.33)	9.11
scpnrg5	62.80 (0.40)	**62.56** (0.54)	62.00	**61.00**	40.20 (35.70)	**4.08** (2.52)	9.84
scpnrh1	34.90 (0.60)	**34.06** (0.24)	34.00	34.00	8.70 (16.30)	**3.51** (2.21)	2.48
scpnrh2	34.90 (0.30)	**34.00** (0.20)	34.00	**33.00**	7.80 (21.20)	**3.31** (2.10)	2.36
scpnrh3	34.90 (0.30)	**34.02** (0.14)	34.00	34.00	19.10 (32.20)	**3.35** (2.03)	5.70
scpnrh4	34.90 (0.60)	**34.06** (0.24)	34.00	34.00	26.10 (67.70)	**3.69** (2.42)	7.08
scpnrh5	34.80 (0.40)	**34.02** (0.14)	34.00	34.00	50.30 (150.00)	**3.38** (1.99)	14.90

Appendix B

B.1 Dialectic Search Procedure – Costas Array Problem

We present the details of our dialectic search algorithm for the Costas array problem presented in Section 4.3.1. In Algorithm B.1 we outline the constraint-based local search model for this problem. Then, in Algorithm B.2 we present the dialectic search procedure to solve this problem.

```
1  import cotls;
2  int argc = System.argc();
3  int n = argv[2].toInt();
4  float start = System.getCPUTime();
5  range Size = 1..n;
6
7  int np = 0;
8  forall(i in Size)
9    forall(j in i+1..n)
10     np++;
11
12 range npRange = 1..np;
13 int pairA[npRange];
14 int pairB[npRange];
15 int index = 1;
16 forall(i in Size){
17   forall(j in i+1..n)
18     pairA[index] = i; pairB[index] = j; index++;
19
20 Solver<LS> m();
21 ConstraintSystem<LS> S(m);
22 RandomPermutation perm(Size);
23 var{int} costas[Size](m, Size) := perm.get();
24 S.post(alldifferent(costas));
25 forall(i in 1..n-2)
26   S.post(alldifferent (all (j in 1..n-i)(costas[j]-costas[j+i])));
27 m.close();
```

Code B.1: The Constraint-Based Local Search Model for the Costas Array Problem.

```
1   Solution solution(m);
2   int noImp = 0;
3   int res = 0;
4
5   while (true){
6
7     while(true){
8       selectMin(i in Size, j in Size, delta = S.getSwapDelta(costas[i],
9       costas[j]) : i != j) (delta){
10        if (delta >= 0) break;
11        costas[i] :=: costas[j];
12      }
13    }
14    if (S.violations() == 0) { cout << "SUCCESS" << endl;break;}
15
16    /// ALLOWED SWAPS
17    UniformDistribution dist(npRange);
18    int siz = dist.get();
19    RandomPermutation pairPerm(npRange);
20    int allows[npRange] = pairPerm.get();
21    set{int} allowed();
22    forall(i in 1..siz) {
23          if ( S.violations( costas[ pairA[i]]) > 0 ||
24               S.violations( costas[pairB[i]]) > 0)
25              allowed.insert(allows[i]);
26    }
27
28    solution = new Solution(m);
29    int origCost = S.violations();
30    int switchPair;
31
32    if(noImp++ < noImpLimit) {  /// GREEDILY SWAP
33      while(allowed.getSize() != 0) {
34        selectMin(i in allowed,
35        delta = S.getSwapDelta(costas[ pairA[i]],
36        costas[pairB[i]])) (delta){
37          switchPair = i;
38          costas[ pairA[i] ] :=: costas[ pairB[i] ];
39        }
40
41        allowed.delete(switchPair);
42        if (S.violations() <= origCost) {
43          solution = new Solution(m);
44            noImp = 0;
```

```
45          if (S.violations() < origCost)
46            origCost = S.violations();
47        }
48      }
49      solution.restore();
50    } else {  /// RANDOMIZE
51      noImp = 0;
52      if(res++ < resLimit) {
53        perm.reset();
54        int selectK[Size] = perm.get();
55        RandomPermutation place(1..k);
56        int p[1..k] = place.get();
57        int per[Size];
58        forall(i in Size)
59          per[i] = costas[i];
60
61        with delay(m) {
62          forall(i in 1..k)
63          costas[ selectK[i] ] := per[ selectK[p[i]] ];
64        }
65      } else { /// RESTART
66        res = 0;
67        with delay(m){
68          perm.reset();
69          forall(i in Size)
70          costas[i] := perm.get();
71        }
72      }
73    }
74  }
75
76  float finish = System.getCPUTime();
77  cout << costas << endl;
78  cout << A Costas array of order << n << (finish-start)/1000.0 << endl;
79  forall(i in 1..n-2)
80    cout << all (j in 1..n-i)(costas[j]-costas[j+i]) << endl;
```

Code B.2: Dialectic Search Algorithm (in: noImpLimit, in: resLimit, in: k)

Index

ACO, *see* ant-colony optimization
ant-colony optimization
    phermone, 19
Artificial Intelligence, 3

big valley structure, 46
binary constraint satisfaction problems, 6
binary search
    skewed binary search, 5
binary search tree, 124
    self-adjusting binary search tree, 124
    skewed binary search tree, 30
Bioinformatics, 3
Boolean Satisfiability, 3
    DPLL, 3
    unit propagation, 3, 12

Constrained Optimization, 5, 22
Constraint Programming, 3, 60
    AC-4, 114
    AC-6, 84, 115
    activity-based search, 126
    cost-based filtering, 95
    impact-based search, 60, 61, 126
    symmetry breaking, 35
Constraint Satisfaction, 5
Continuous Optimization, 5, 49
    Alpine, 49
    De Jong's noiseless function, 49
    Rastrigin, 49
CP-based Lagrangian Relaxation, 96

decision problems, 11
Dialectic Search, 42, 43
    antithesis, 45
    exploitation, 45
    exploration, 45
    synthesis, 45
    thesis, 45
Dichotomic Search, 22, 124
    skewed dichotomic search for constrained optimization, 30
Discrete Optimization, 5
DS, *see* Dialectic Search

evolutionary algorithms, 18
exploitation-exploration trade-off, 42

feasibility solver, 22
feasible solution, 13
Formal Languages
    accepted language, 76
    alphabet, 75
    Chomsky Normal Form, 78

context-free grammar, 127
context-free grammars, 75
derivation, 75
finite automaton, 76
language, 75
regular grammar, 127
Type 1 grammars, 76
Type 3 grammars, 76
word problem, 75

genetic algorithms, 18
chromosome, 18
cross-over, 18
offspring, 18
population, 18
global constraint
`all_different` constraint, 106, 128
`bipartite` `same-relation` constraint, 110
`clique` `same-relation` constraint, 107
`context-free` `grammar` constraint, 6, 77
`cost_regular` constraint, 128
`DAG` `same-relation` constraint, 111
`grid` `same-relation` constraint, 112
`regular` constraint, 74
`soft_grammar` constraint, 128
Grammar Constraints, 74
efficient context-free grammar constraints, 74
efficient context-free grammar filtering, 82
GRASP, *see also* greedy randomized adaptive search procedure, 125
reactive GRASP, 18, 125
restricted candidate list, 18
greedy randomized adaptive search procedure, 18
restricted candidate list, 18

heavy-tailed distribution, 13, 35

impact-based search, 60, *see also* pseudo-cost branching, strong branching
incrementality, *see also* Grammar Constraints, Same-Relation Constraints
inference, 3
deductive reasoning, 3
Instance Specific Algorithm Configuration, 54
GGA, 54
stochastic off-line programming, 54
ISAC, *see* Instance Specific Algorithm Configuration

Linear Programming, 26
local search
big valley structure, 126
Local Search Meta-Heuristics, 12, *see also* search, Systemmatic Search
ant-colony optimization, 19
dialectic search, 42
evolutionary algorithms, 18
genetic algorithms, 18
greedy randomized adaptive search procedure, 18, 125
iterated local search, 41, 125
simulated annealing, 16, 41
tabu search, 17, 41, 125
variable neighborhood search, 41
very large scale neighborhood search, 41
WalkSAT, 125

Mathematical Programming, *see also* Operations Research, Linear Programming
pseudo-cost branching, 61
strong branching, 61
meta-heuristic, *see also* heuristic

Operations Research, *see also* Mathematical Programming
optimal solution, 13
optimization constraint, 96
optimization problems, 11

path relinking, *see also* tabu search
problem
- all-interval series problem, 126
- Alpine, 49
- Costas array, 46, 66
- De Jong's noiseless function, 49
- Golomb's ruler problem, 126
- magic square problem, 65
- n-queens problem, 126
- Quasigroup completion problem, 65
- Rastrigin, 49
- set covering problem, 5, 50
- shift-scheduling problem, 92
- stable marriage problem, 119
- table planning problem, 120
- travelling salesman problem, 19
- weighted magic square problem, 34
- weighted Quasigroup completion problem, 34

pseudo-cost branching, *see also* strong branching

SA, *see* simulated annealing
Same-Relation Constraints, 106
search, *see also* Systemmatic Search, Local Search Meta-Heuristics
- backtracking, 12
- big valley structure, 46
- binary search, 22
- branch-and-bound, 12, 22
- complete search, 4
- completeness, 12
- constructive search, 11
- dichotomic search, 22
- heuristic, *see also* meta-heuristic
- impact-based search, 6
- incomplete, 12
- incomplete search, 4
- local search, 12, 41
- perturbative search, 11
- restarted search, 13
- search tree, 12
- stochastic local search, 12
- stochastic systematic search, 13
- systematic search, 12

SIMANN, 49
simulated annealing, 16
- cooling schema, 16
- Metropolis criterion, 16
- reheats, 41
- temperature, 16

Steeter and Smith strategy, 5, 30
strong branching, *see also* pseudo-cost branching
symmetry breaking, 35
Systematic Search, *see also* search, Local Search Meta-Heuristics
- A search, 15
- A* search, 15
- best-first search, 14
- breadth-first search, 13, 14
- British museum procedure, 13
- depth-first search, 13, 14
- FIFO search strategy, *see also* breadth-first search
- iterative deepening search, 15
- LIFO search strategy, *see also* depth-first search
- linear search, *see also* depth-first search
- search toward the bottom, *see also* depth-first search

smaller cost first search, *see also* uniform cost search
uniform cost search, 14

tabu search, 17
aspiration criteria, 41
ejection chaining, 41
path relinking, 17
reactive tabu search, 17
strategic oscillation, 17
tabu list, 17, 41
tabu tenure, 41
TS, *see* tabu search

upper confidence trees, 63

Bibliography

[1] J. Agustí and R. Lopez. Inteligencia Artificial. Conceptos, Técnicas y Aplicaciones. *Marcombo-Boixareu*, 1987.

[2] R.K. Ahuja, O. Ergun, J.B. Orlin, A.P. Punnen. A Survey of Very Large Scale Neighborhood Search Techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.

[3] K.R. Apt. The Rough Guide to Constraint Propagation, In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP-99)*, LNCS, 1713:1–23, Springer, 1999.

[4] T. Bäck. Evolutionary Algorithms in Theory and Practice. *Oxford University Press New York, NJ, USA*, 1996.

[5] A. Barr and E.A. Feigenbaum. The Handbook of Artificial Intelligence. *William Kaufman, Inc.*, 1981.

[6] B. Battiti and G. Tecchiolli. The Reactive Tabu Search. *INFORMS Journal on Computing*, 6(2):126–140, 1994.

[7] J.E. Beasley. OR-Library: Distributing Test Problems by Electronic Mail. *Operations Research Society*, 41:1069–1072, 1990.

[8] J.E. Beasley. Population Heuristics. In *P.M. Pardalos and M.G.C. Resende (Eds), Handbook of Applied Optimization*, New York:Oxford University Press, 38–156, 2002.

[9] N. Beldiceanu, M. Carlsson, T. Petit. Deriving Filtering Algorithms from Constraint Checkers. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP-04*, LNCS, 3258:107–122, Springer, 2004.

[10] J. Bell and G. Gupta. An Evaluation of Self-adjusting Binary Search Tree Techniques. *Software Practice and Experienc*, 23:369–382, 1993.

[11] C. Bessière and M.O. Cordier. Arc-Consistency and Arc-Consistency Again. *AAAI*, 108–113, 1993.

[12] C. Bessière and J-C. Régin. Local consistency on conjunctions of constraints. *Workshop on Non-Binary Constraints*, 1998.

[13] C. Bessière and J.C. Régin. Using Constraint Metaknowledge to Reduce Arc Consistency Computation. *Artificial Intelligence*, 107(1):125–148, 1999.

[14] C. Bessière, E. Hebrard, B. Hnich, T. Walsh. The Complexity of Reasoning with Global Constraints. *Constraints*, 12(2):239–259, 2007.

[15] C. Bessière, A. Chmeiss, L. Sais. Neighborhood-based Variable Ordering Heuristics for Constraint Satisfaction Problem. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP-01)*, LNCS, 565–569, Springer, 2001.

[16] C. Bessière and J-C. Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems, In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming (CP-96)*, LNCS, 61–75, 1996.

[17] K.D. Boese. Cost Versus Distance in the Traveling Salesman Problem. *Techinal Report*, CSD-950018, UCLA, 1995.

[18] F. Boussemart, F. Hemery, C. Lecoutre, L. Sais. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI-2004)*, 146–150, 2004

[19] D. Brélaz. New Methods to Color the Vertices of a Graph, *ACM*, 22:251–256, 1979.

[20] G.S. Brodal and G. Moruz. Skewed Binary Search Trees. *ESA*, 708–719, 2006.

[21] A. Bulatov. Tractable Conservative Constraint Satisfaction Problems. LICS, 321-330, 2003.

[22] A. Bulatov, A. Krokhin, P. Jeavons. Classifying the Complexity of Constraints using Finite Algebras/ *SIAM Journal on Computing*, 34:3, 720–742, 2005.

[23] Hadrien Cambazard and Narendra Jussien. Identifying and Exploiting Problem Structures Using Explanation-Based Constraint Programming. *Constraints*, 11(4):295–313, 2006.

[24] A. Caprara, M. Fischetti, P. Toth, D. Vigo, P.L. Guida. Algorithms for Railway Crew Management. *Mathematical Programming*, 79:125–141, 1997.

[25] M. Carlsson and N. Beldiceanu. From Constraints to Finite Automata to Filtering Algorithms. *The European Symposium on Programming (ESOP)*,94–108, 2004.

[26] V. C̃erny. Thermodynamics Approach to the Travelling Salesman Problem: An Efficient Simulation Algorithm. *Optimization Theory Applications.*, 45:41–51, 1985.

[27] N. Chomsky. Three Models for the Description of Language. *IRE Transactions on Information Theory* 2:113–124, 1956.

[28] P. Codognet and D. Diaz. Yet Another Local Search Method for Constraint Solving. *International Symposium on Stochastic Algorithms: Foundations and Applications*, K. Steinhöfel (Ed.), LNCS, 2246, 1, 2001.

[29] P. Codognet and D. Diaz. An Efficient Library for Solving CSP with Local Search. In *Proceedings of the fifth International Conference on Metaheuristic (MIC-2003)*, 2003.

[30] M.C. Côté, B. Gendron, C.G. Quimper and L.M. Rousseau. Formal Languages for Integer Programming Modeling of Shift Scheduling Problems. *Constraints*, 54–76, 2011.

[31] A. Colmerauer. Prolog in 10 Figures. *Communication of the ACM*, 28:1296-1310, 1985.

[32] M.C. Cooper. An Optimal k-Consistency Algorithm, *AI*, 78–87, 2001.

[33] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms (3. ed.), *MIT Press*, I-XIX, 1–1292, 2009.

[34] J.P. Costas. A Study of a Class of Detection Waveforms Having Nearly Ideal Range-Doppler Ambiguity Properties. In *Proceedings of the IEEE*, 72(8), 996–1009, 1984.

[35] J.A. Crawford and L.D. Auton. Experimental Results on the Cross-Over Point in Random 3-SAT. *Artificial Intelligence Journal*, 81:31–57, 1996.

[36] V. Dalmau, P.G. Kolaitis, M.Y. Vardi. Constraint Satisfaction, Bounded Treewidth, and Finite-Variable Logics. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP-02)*, LNCS, 2470: 310–326, 2002.

[37] M. Davis, G. Logemann, D. Loveland. A Machine Program for Theorem Proving. *Communication of the ACM*, 5(7):394–397, 1962.

[38] R. Dechter and J. Pearl. Network-based Heuristics for Constraint Satisfaction Problems. *Artificial Intelligence*, 34:1, 1–38, 1988.

[39] R. Dechter and J. Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:3, 353–366, 1989.

[40] S. Demassey, G. Pesant, L.-M. Rousseau. A Cost-Regular Based Hybrid Column Generation Approach. *Constraints*, 11(4):315–333, 2006.

[41] D. Diaz, F.Richoux, P. Codognet, Y. Caniou, S. Abreu. Constraint-Based Local Search for the Costas Array Problem. *Learning and Intelligent OptimizatioN Conference (LION-6)*, LNCS, p. 6, 2012.

[42] D. Diaz, F.Richoux, P. Codognet, Y. Caniou, S. Abreu. Parallel Local Search for the Costas Array Problem. *IEEE Workshop on New Trends in Parallel Computing and Optimization (PCO'12), held in conjunction with IPDPS 2012*, 2012.

[43] E.W. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numerische Matematik*, 1(5):269–27, 1959.

[44] M. Dorigo. Optimization, Learning and Natural Algorithms. *Ph.D. Thesis*, 1992.

[45] M. Dorigo and G.Di Garo. The Ant-Colony Optimization Meta-Heuristic. *New Ideas in Optimization.*, London:McGraw-Hill, 11–32, 1999.

[46] M. Dorigo, G.Di Garo and L.M. Gambardella. Ant Algorithms for Discrete Optimization. *Artificial Life.* 5:137–172, 1999.

[47] M. Dorigo and T. Stützle. The Ant-Colony Optimization Meta-Heuristic: Algorithms, Applications and Advances. *Handook of Metaheuristics*, Boston:Kluwer, 251–285, 2003.

[48] R. Downey, and M. Fellows. Parametrized Complexity. Springer, 1999.

[49] T. Fahle, U. Junker, S.E. Karisch, N. Kohl, M. Sellmann, and B. Vaaben. Constraint Programming Based Column Generation for Crew Assignment. *Journal of Heuristics*, 8(1):59–81, 2002.

[50] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry Breaking. *7th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS, 2239:93–107, Springer, 2001.

[51] T. Feder and M. Vardi. The Computational Structure of Monotone Monadic SNP and Constraint Satisfaction *SIAM* Journal on Computing, Vol. 28(1), 57–104, 1998.

[52] M.R. Fellows, T. Friedrich, D. Hermelin, N. Narodytska, F.A. Rosamond. Constraint Satisfaction Problems: Convexity Makes AllDifferent Constraints Tractable. In*Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI-2011)*, 522–527 2011.

[53] T.A. Feo and M.G.C. Resende. A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem. *Operations Research Letters*, 8:67–71, 1989.

[54] T.A. Feo and M.G.C. Resende. Greedy Randomized Adaptive Procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[55] S. Ferré and R.D. King. A Dichotomic Search Algorithm for Mining and Learning in Domain-Specific Logics. *Fundamenta Informaticae*, 66(1–2):1–32, 2005.

[56] J.G. Fichte. Wissenschaftslehre (Science of Knowledge), 1794.

[57] C. Fleurent and F. Glover. Improved Constructive Multistart Strategies for the Quadratic Assignment Problem using Adaptive Memory. *INFORMS Journal on Computing*, 11:198–204, 1999.

[58] F. Focacci, A. Lodi, and M. Milano. Cost-Based Domain Filtering. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP-99)*, LNCS, 1713:189–203, Springer, 1999.

[59] F. Focacci and M. Milano. Global Cut Framework for Removing Symmetries. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP-01)*, LNCS, 2239:75–92, Springer, 2001.

[60] A. Freedman and N. Levanon. Staggered Costas Signals. *IEEE Trans. Aerosp. Electron Syst*, AES-22(6):695–701, 1986.

[61] E. Freuder. A Sufficient Condition for Backtrack-bounded Search. *Journal of the ACM*, 32(4):755-761, 1985.

[62] E. Freuder. Backtrack-Free and Backtrack-Bounded Search. *In Search in Artificial Intelligence*, 343–369, 1988.

[63] E. Freuder. Complexity of K-Tree Structured Constraint Satisfaction Problems. *AAAI*, 4–9, 1990.

[64] M. Garey and D. Johnson. *Computers and Tractability: A Guide to the Theory of NP-Completeness. San Francisco: Freeman*, 1979.

[65] I.P. Gent, R.W. Irving, D.F. Manlove, P. Prosser, B.M. Smith. A Constraint Programming Approach to the Stable Marriage Problem. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP-01)*, LNCS, 2239:225–239, Springer, 2001.

[66] C. A. Gil, M. Sellmann and K. Tierney. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP-09)*, LNCS, 5732:142–157, Springer, 2009.

[67] F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers & Operations Research*, 5:533–549, 1986.

[68] F. Glover. Tabu Search - Part I. *ORSA Journal on Computing*, 1:190–206, 1989.

[69] F. Glover. Tabu Search - Part II. *ORSA Journal on Computing*, 2:4–32, 1990.

[70] F. Glover and M. Laguna. Tabu Search, Kluwer, Boston, 1997.

[71] F. Glover, M. Laguna and R. Marti. Fundamentals of Scatter Search and Path Relinking. Control and Cybernetics, 39:653–684, 2000.

[72] W.L. Goffe, G.D. Ferrier,J. Rogers. Global Optimization and Statistical Functions with Simulated Annealing. *Journal of Econometrics*, 60(1-2):65–99, 1994.

[73] D.E. Goldberg. Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, 1989.

[74] S.W. Golomb and H. Taylor. Two Dimensional Synchronization Patterns for Minimum Ambiguity. *IEEE Trans. Information Theory*, IT-28(4):600–604, 1982

[75] C.P. Gomes and M. Sellmann. Streamlined constraint reasoning. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP-04)*, LNCS, 274–287, Springer, 2004.

[76] C.P. Gomes, B. Selman, H. Kautz. Boosting Combinatorial Search Through Randomization. In *Proceedings of Fifteenth National Conference on Artificial Intelligence, AAAI-98*, 431–437, 1998.

[77] C.P. Gomes, B. Selman, N. Crato, H. Kautz. Heavy-tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24(1–2):67–100,2000.

[78] C. P. Gomes, D. Shmoys. Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In In *Proceedings of Computational Symposium on Graph Coloring and Generalizations*, 2002.

[79] R.E. Gomory. Outline of an Algorithm for Integer Solutions to Linear Programs, American Mathematical Society, 64–275, 1958.

[80] G. Gottlob and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124(2):243–282, 1999.

[81] M. Grohe. The Complexity of Homomorphism and Constraint Satisfaction Problems Seen From the Other Side. *Journal of the ACM*, 54(1), 2007.

[82] J. Gu. Efficient Local Search for very Large-Scale Satisfability Problems. *SIGART Bulletin*, 3(1):8-12, 1992.

[83] P. Hansen and N. Mladenovic. Variable Neighbourhood Search: Principles and Applications. *European Journal of Operational Research*, 130:49–467, 2001.

[84] R. Haralick and G. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems, Artificial Intelligence, 14:263–313, 1990.

[85] J.P. Hart and A.W. Shogan. Semi-Greedy Heuristics: An Empirical Study. *Operations Research Letters*, 6:107–114, 1987.

[86] W. Harvey. Symmetry Breaking and the Social Golfer Problem. *In P. Flener, J. Pearson (Eds.), Proceedings of SymCon'01: Symmetry in Constraints*, 2001.

[87] G.W.F. Hegel. Phänomenologie des Geistes (Phenomenology of Spirit), 1807.

[88] P. Hell, J. Huang, A. Rafiey. List Homomorphism to Reflexive Digraphs : Dichotomy Classification. *Submitted*, 2009.

[89] A. Hertz and D.de Werra. A Tutorial on Tabu Search. *Local Search in Combinatorial Optimization*, 121–136. John Wiley&Sons, Chichester, UK, 1997.

[90] K.L. Hoffmann and M.W. Padberg. Solving Airline Crew Scheduling Problems by Branch-and-Cut. *Management Science*, 39(6):657–682, 1993.

[91] T. Hogg, B.A. Huberman, and C.P. Williams. Phase Transitions and Complexity. *Artificial Intelligence*, 81, 1996.

[92] J.H. Holland. Adaptation in Natural and Artificial Systems. *The University of Michigan Press*, 1975.

[93] J.N. Hooker. Logic-based Methods for Optimization: Combining Optimization and Constraint Satisfcation. *John Wiley and Sons, Inc, New York.*, 2000.

[94] H.H. Hoos. An Adaptive Noise Mechanism for WalkSAT. In *Proceedings of the Eighteenth National Conference in Artificial Intelligence (AAAI-02)*, 655–660, 2002.

[95] H.H. Hoos and T. Stützle. Stochastic Local Search Foundations and Applications. *Morgan Kaufmann, Elsevier*, 2004.

[96] J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory, Languages and Computation, *Addison Wesley*, 1979.

[97] E. Housos and T. Elmoth. Automatic Optimization of Subproblems in Scheduling Airline Crews. *Interfaces*, 27(5):68–77, 1997.

[98] F. Hutter, D.A.D. Tompkins and H.H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. *CP 02*, 233–248, 2002.

[99] T. Ibaraki. Theoretical Comparisons of Search Strategies in Branch-and-Bound Algorithms. Phase Transitions and Complexity. *International Journal of Computer and Information Sciences*, 5:315–344, 1976.

[100] T. Ibaraki. Enumarative Approaches to Combinatorial Optimization. Part I. *International Journal of Computer and Information Sciences*, 7:315–343, 1987.

[101] IBM.: *IBM ILOG* Reference Manual and User Manual. V6.4, IBM 2009.

[102] IBM.: IBM ILOG CPLEX Optimization Studio 12.2 (2011)

[103] P. Jeavons. On the Algebraic Structure of Combinatorial Problems. *Theoretical Computer Science*, 185–204, 1998.

[104] C. Jefferson, S. Kadioglu, K.E. Petrie, M. Sellmann and S. Živný. Same-Relation Constraints. In *Proceedings of the Fifteenth International Conference on the Principles and Practice of Constraint Programming (CP-2009)*, 470–485, 2009.

[105] T. Jones and S. Forrest. Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, 184–192, Morgan Kaufmann, 1995.

[106] N. Jussien and O. Lhomme. Dynamic Domain Splitting for Numeric CSPs. In *Proceedings of Seventh European Conference on Artificial Intelligence (ECAI-98)*, 224–228, 1998.

[107] S. Kadioglu, Y.Malitsky, K. Tierney and M. Sellmann. ISAC – Instance Specific Algorithm Configuration. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI-2010)*, 751–756, 2010.

[108] S. Kadioglu, E. O'Mahony, P. Refalo and M. Sellmann. Incorporating Variance in Impact-Based Search. In *Proceedings of the Seventeenth International Conference on the Principles and Practice of Constraint Programming (CP-2011)*, 470–477, 2011.

[109] S. Kadioglu and M. Sellmann. Efficient Context-Free Grammar Constraints. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence (AAAI-08)*, 310–316, 2008.

[110] S. Kadioglu and M. Sellmann. Dialectic Search. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP-09)*, , LNCS, 5732:486–500, Springer, 2009.

[111] S. Kadioglu and M. Sellmann. A Local Search Meta-Heuristic for Non-Specialists. In *Proceedings of CP-2009 Satellite Workshop on Constraint Reasoning and Optimization for Computational Sustainability (CROCS-09)*, 7–8, 2009.

[112] S. Kadioglu and M. Sellmann. Grammar Constraints. *Constraints*, 15(1):117–144, 2010.

[113] I. Katriel, M. Sellmann, E. Upfal, P. Van Hentenryck. Propagating Knapsack Constraints in Sublinear Time. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, 2007.

[114] G. Katsirelos, S. Maneth, N. Narodytska and T. Walsh. Restricted Global Grammar Constraints. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP-094)*, LNCS, 501–508, 2009.

[115] G. Katsirelos, N. Narodytska and T. Walsh. The Weighted CFG Constraint. *CoRR*, abs/0909.4456, 2009.

[116] G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP-04)*, LNCS 3258:482–495, 2004.

[117] G. Pesant, C.G. Quimper, L.M. Rousseau and M. Sellmann. The Polytope of Context-Free Grammar Constraints. In *Proceedings of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-2009)*, LNCS, 3258:482–495, 2009.

[118] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671-680, 1983.

[119] S. Kirkpatrick and B. Selman. Critical Behavior in the Satisfiability of Random Boolean Expressions. *Science*, 264:1297–1301, 1994.

[120] W. Klostermeyer and G. MacGillivray. Homomorphisms and Oriented Colorings of Equivalence Classes of Oriented Graphs. *Discrete Mathematics*, 274(1-3):161–172, 2004

[121] D.-E Knuth. The Art of Computer Programming. *Addison-Wesley Longman Publishing Co., Inc.*, 2(3):seminumerical algorithms, 232, 1997

[122] M. Kraitchik. Magic Squares, *Mathematical Recreations, New York: Norton*, 7:142–192, 1942.

[123] V. Kumar. Algorithms for Constraint Satisfaction Problems: A Survey, *The AI Magazine, AAAI*, 13:32–44, 1992.

[124] G.T. Leong. Constraint Programming for the Traveling Tournament Problem. *Project Thesis*, National University of Singapore, 2003.

[125] K. Levente, S. Csaba Bandit Based Monte-Carlo Planning. *Machine Learning ECML*, 2006

[126] O. Lhomme. Arc-consistency Filtering Algorithms for Logical Combinations of Constraints. *Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, LNCS 3011:209–224, 2004.

[127] C.J. Liao. A New Node Selection Strategy in the Branch-and-Bound Procedure. *Computers and Operations Research*, 21:1095-1101, 1994.

[128] S. Lin and B.W. Kernighan. An Effective Heuristic Algorithm for the Travelling Salesman Problem. *Operations Research*, 21(2):498-516, 1973.

[129] I. Lustig and J.-F. Puget. Constraint Programming. *Encyclopedia of Operations Research and Management Science*, 136–140, Kluwer, 2001.

[130] W.G. Macready and D.H. Wolpert. Bandit Problems and the Exploration/Exploitation Tradeoff. *IEEE Transactions on Evolutionary Computation*, 2(2):2–22, 1998.

[131] Y. Malitsky and M. Sellmann. Stochastic Offline Programming. *Proceedings of the Twenty-first International Conference on Tools with Artificial Intelligence (ICTAI-2009* 784–791, 2009.

[132] E. Marchiori and A. Steenbeek. An Iterated Heuristic Algorithm for the Set Covering Problem. *Algorithm Engineering, WEA*, 155–166, 1998.

[133] K. Marriott and P.J. Stuckey. Programming with Constraints: An Introduction, The MIT Press, 1998.

[134] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equations of State Calculations by Fast Computing Machines. *Chemical Physics*, 21(6):1087–1092, 1953.

[135] L.D. Michel and P. Van Hentenryck. Activity-Based Search for Black-Box Contraint-Programming Solvers. *CoRR*, abs/1105.6314, 2011.

[136] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, 58(1-3):161-205, 1992.

[137] R. Mohr and T. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*. 28(2):225–233, 1986.

[138] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, L. Troyansky. Determining Computational Complexity From Characteristic 'Phase Transitions'. *Nature* 400:133–137, 1999.

[139] E.F. Moore. The Shortest Path Through a Maze. In *Proceedings of an International Symposium on the Theory of Switching*, Part II, 285–292, 1959.

[140] J. Moran. The Wonders of Magic Squares. *New York, Vintage*, 1982.

[141] N. Musliu. Local Search Algorithm for Unicost Set Covering Problem. *IEA/AIE*, 302–311, 2006

[142] G.L. Nemhauser and A. Wolsey. Integer and Combinatorial Optimization. *John Wiley and Sons, Inc, New York.*, 1988.

[143] A. Newell, J.C. Shaw and H.A. Simon. Elements of a Theory of Human Problem Solving. *Psychological Review*, 65(3):151–166, 1958.

[144] A. Newell and H.A. Simon. Computer Science as Empirical Inquiry: Symbols and Search. *Communications of the ACM*, 19:3:113–126, 1976.

[145] J. Nievergelt and E.M. Reingold. Binary Search Trees of Bounded Balance. *Annual ACM Symposium on Theory of Computing*, 137–142, 1972.

[146] I.H. Osman and J.P. Kelly. *Meta-Heuristics: Theory and Applications*, Kluwer, Boston, 1996.

[147] G.E. Ottoson, E.S. Thorsteinsson, J.N. Hooker. Mixed Global Constraints and Inference in Hybrid and Inference in Hybrid CLP-IP Solvers. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming, Post-Conference Workshop on Large Scale Combinatorial Optimization and Constraints*, , 1999.

[148] J. Pearl. Heuristics: Intelligent Search Strategies for Computer Problem Solving, *Addison-Wesley*, 1984.

[149] G. Pesant. A Regular Language Membership Constraint for Sequences of Variables. *Modelling and Reformulating Constraint Satisfaction Problems*, 110–119, 2003.

[150] G. Polya. How to Solve it. *Princeton: Princeton University Press*, 1945.

[151] J.Y. Potvin. Genetic Algorithms for the Traveling Salesman Problem. *Annals of Operations Research*, 63:339–370, 1996.

[152] M. Prais and C.C. Riberio. Reactive GRASP: An Application to a Matrix Decomposition Problem. *INFORMS Journal on Computing*, 12:164–176, 2000.

[153] C.-G. Quimper and L.-M. Rousseau. Language Based Operators for Solving Shift Scheduling Problems. *Metaheuristics*, 2007.

[154] C.-G. Quimper and T. Walsh. Global Grammar Constraints. In *Proceedings of the Twelveth International Conference on Principles and Practice of Constraint Programming (CP-06)* , LNCS, 3258:751–755, 2006.

[155] C.-G. Quimper and T. Walsh. Decomposing Global Grammar Constraints. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP-07)* , LNCS, 4741:590-604, Springer, 2007.

[156] M. Sellmann. Theoretical Foundations of CP-based Lagrangian Relaxation. *CP*, LNCS 3258:634–647, 2004.

[157] C.R. Reeves and T. Yamada. Genetic Algorithms, Path Relinking and the Flowshop Sequencing Problem. *Evolutionary Computation*, 6:45–60, 1998.

[158] P. Refalo. Impact-Based Search Strategies for Constraint Programming. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP-04)* , LNCS, 557–571, 2004.

[159] P. Refalo. Learning in Search. In *Hybrid Optimization*, Springer, 2011.

[160] J.C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. *AAAI*, 1:362–367, 1994.

[161] Y. Rochat and E.D. Taillard. Probabilistic Diversification and Intensification in Local Search for Vehicle Routing. *Journal of Heuristics*, 1:147–167, 1995.

[162] S.J. Russell and P. Norvig. Artificial Intelligence: A Modern Approach, *Upper Saddle River, New Jersey: Prentice Hall*, 2003.

[163] S.J. Russell and P. Norvig. Artificial Intelligence – A Modern Approach (Third International Edition). *Pearson Education*, I-XVIII, 1–1132, 2010.

[164] A.Z. Salamon and P.G Jeavons. Perfect Constraints Are Tractable. In *Proceedings of the Fourteenth International Conference on Principles and Practice of Constraint Programming (CP-08)* , LNCS, 5202:524–528, 2008.

[165] M. Sellmann. The Theory of Grammar Constraints. In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP-06)* , LNCS, 4204:530–544, Springer, 2006.

[166] M. Sellmann, C. Ansótegui. Disco - Novo - GoGo: Integrating Local Search and Complete Search with Restarts. In *Proceedings of Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, 1051–1056, 2006.

[167] M. Sellmann and S. Kadioglu. Dichotomic Search Protocols for Constrained Optimization. In *Proceedings of the Fourteenth International Conference on Principles and Practice of Constraint Programming (CP-08)* , LNCS, 5202:251–265, Springer, 2008.

[168] B. Selman, H. Kautz and B. Cohen. Noise Strategies for Improving Local Search. In *Proceedings of the Conference on Artificial Intelligence (AAAI-94)*, 337–343, 1994.

[169] B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the Conference on Artificial Intelligence (AAAI-93)*, 440–446, 1992.

[170] SIMANN. Fortran Simulated Annealing code. `http://wueconb.wustl.edu/ goffe`, 2004.

[171] D.D. Sleator and R.E Tarjan. Self-Adjusting Binary Search Trees. In *J. ACM*, 32:3:652–686, 1985.

[172] B. Smith. The Brelaz Heuristic and Optimal Static Ordering. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP-99)* , 504–418, 1999.

[173] B. Smith. Reducing Symmetry in a Combinatorial Design Problem. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 351–360, 2001.

[174] E. Sopena. Oriented graph coloring. *Discrete Mathematics*, 229:(1-3), 359–369, 2001.

[175] M. Streeter and S.F. Smith. Using Decision Procedures Efficiently for Optimization. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling(ICAPS)*, 312–319, AAAI, 2007.

[176] P.J. Stuckey. Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving. In *Proceedings of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-10)* 6140:5-9, 2010

[177] T. Stützle. Iterated Local Search for the Quadratic Assignment Problem. *Techinal Report*, AIDA-99-03, Darmstadt University of Technology, 1999.

[178] T. Stützle and H.H. Hoos. Improvements on the Ant System: Introducing the MAX-MIN Ant System. *Artificial Neural Networks and Genetic Algorithms*, Berlin:Springer-Verlag, 245–249, 1998.

[179] T. Stützle and H.H. Hoos. Max-MIN Ant System. *Future Generation Computer Systems Journal*, 16:889–914, 2000.

[180] J.S. Swarts. The Complexity of Digraph Homomorphisms: Local Tournaments, Injective Homomorphisms and Polymorphisms. *PhD thesis*, University of Victoria, 2008.

[181] P.M. Thompson and J.B. Orlin. The Theory of Cyclic Transfers. *Working Paper No. OR*, 200-89, Operations Research Center, MIT, Cambridge, MA., 1989.

[182] C. Toregas, R. Swain, C. ReVelle, L. Bergman. The Location of Emergency Service Facilities. *Operational Research*, 19(6):1363–1373, 1971.

[183] M. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *CPAIOR*, 113–124, 2001.

[184] C. Unsworth and P. Prosser. An N-ary Constraint for the Stable Marriage Problem. *IJCAI*, 32–38, 2005.

[185] P. Van Hentenryck and L. Michel. Constraint-Based Local Search. *The MIT Press*, Cambridge, Mass. 2005.

[186] P. Van Hentenryck and L. Michel. Synthesis of Constraint-Based Local Search Algorithms From High-Level Models. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, AAAI Press, 2007.

[187] F.J. Vasko and G.R. Wilson. An Efficient Heuristic for Large Set Covering Problems. *Naval Research Logistics Quarterly*, 31:163–171, 1984.

[188] F.J. Vasko and F.E. Wolf. Optimal Selection of Ingot Sizes via Set Covering. *Operations Research*, 35:115–121, 1987.

[189] H.P. Williams. Logic Applied to Integer Programming and Integer Programming Applied to Logic. *European Journal of Operations Research* 81:605-616, 1995.

[190] D. Wojtaszek and J.W. Chinneck. Faster MIP Solutions via New Node Selection Rules. *INFORMS*, 2007.

[191] C.K. Wong and J. Nievergelt. Upper Bounds for the Total Path Length of Binary Trees. *Journal of the ACM*, 20(1):1–6,1973.

[192] A. Zanarini and G. Pesant. Solution Counting Algorithms for Constraint-Centered Search. *Constraints*, 4(3):392–413, 2009.

[193] H. Zhang. Specifying Latin Square Problems in Propositional Logic. *Automated Reasoning and Its Applications*, MIT Press, 1997.