Integrating Concepts from Constraint Programming and Operations Research Algorithms

Dissertation

von

Torsten Fahle

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn.

Paderborn, Dezember 2002

When you have eliminated the impossible,
whatever remains, however improbable, must be the truth.

Sir Arthur Conan Doyle (1859–1930)
Sherlock Holmes to Dr. Watson
in The Sign of Four *[68].*

But we all know the world is nonlinear.

Harold Hotelling, (1910–1975)
to George Dantzig, 1947 [59].

Jede Lösung eines Problems ist ein neues Problem.[1]

Johann Wolfgang von Goethe (1749–1832)
to chancellor von Müller, 1821 [95].

[1] Every solution of a problem is a new problem. (non-authorized translation)

Acknowledgment

Research is seldom a one-person project and many individuals have contributed to the results presented in this thesis. The first to mention here is my supervisor Prof. Dr. Burkhard Monien for his support during the last few years and for providing an excellent working environment in his group.

I am very indebted to my colleagues in the research group and in the PC^2 for providing a good working atmosphere. I would especially like to thank Rainer Feldmann, Silvia Götz, Sven Grothklags, Georg Kliewer, Jürgen Schulze, Norbert Sensen, Ulf-Peter Schroeder and Meinolf Sellmann for interesting discussions on optimization topics. Uli Ahlers, Sven Grothklags, Georg Kliewer, Tomas Plachetka, Stefan Schamberger, Norbert Sensen, Thomas Thissen and the LINUX-Gurus could always help me with the C++ standard, CPLEX's "special features", shell-programming, mysterious bugs in the code or extra hard- and software requirements.

Parts of this thesis were supported by the EU-Esprit projects PARALOR[2] and ALCOM-FT[3] and by the BMBF project PARPAP[4]. Michael Laska's management and administration of these projects was very much appreciated and helped us to concentrate on the research in question. Within the projects I had the pleasure of working in co-operation with other researchers. I would like to say: *Merci bien*, Ulrich, *Mange tak*, Stefan, Niklas and Bo, *Ευχαριστώ πολύ*, Takis and Kyriakos, *Danke schön*, Stefan, Sven, Georg, Stefan, Jürgen and Meinolf for the fruitful collaboration which resulted in some joint papers.

Geraldine Brehony found all those misspellings that only a non-native writer could make — *go raiḃ míle maiṫ agat*. In addition, Ruth Dietzel, Sandra Feisel, Sven Grothklags, Georg Kliewer and Manuel Rode read and commented on individual chapters — thanks a lot!

I am indebted to Prof. Dr. Dorothea Wagner (Konstanz) and Prof. Dr. Wilfried Hauenschild (Paderborn) for co-reviewing this thesis.

Finally, I'd like to thank Ruth, my parents and my friends for their support and patience during the last months when many evenings and weekends were taken up by this work.

Paderborn,
December 2002 *Torsten Fahle*

[2]partially funded by the ESPRIT programme of the Commission of the European Union as project number 24 960.

[3]partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

[4]partly supported by the German Ministry for Education and Research Bmb+f (PARPAP project 01 HR 9955)

Contents

1 Introduction **1**
- 1.1 Integrating Concepts from CP and OR 1
  - 1.1.1 Basic Concepts of Integer Linear Programming and Constraint Programming 2
  - 1.1.2 Strategic Considerations 4
  - 1.1.3 Tactical Considerations 4
- 1.2 Contribution of the Thesis 5
  - 1.2.1 Airline Crew Rostering, Shortest Path Constraint and CP based Column Generation 5
  - 1.2.2 Automatic Recording Problem, Knapsack Constraint and CP based Lagrangian Relaxation 6
  - 1.2.3 Home Health Care and Domain Filtering for Sequences 7
  - 1.2.4 Maximum Clique and Comparing OR Bounds to CP Domain Filtering 8

2 Basic Concepts **9**
- 2.1 Constraint Programming 9
  - 2.1.1 Consistency 11
  - 2.1.2 Tree Search 14
- 2.2 Operations Research Approaches 15
  - 2.2.1 LP-based Branch-and-Bound 16
  - 2.2.2 Column Generation 17
  - 2.2.3 Lagrangian Relaxation 19
- 2.3 Integrated Approaches 20
  - 2.3.1 Cost Based Filtering 21
  - 2.3.2 CP Based Column Generation 22
  - 2.3.3 CP Based Lagrangian Relaxation 23
- 2.4 Experimental Methodology 25
  - 2.4.1 Software Packages 26
  - 2.4.2 Benchmarks 26
  - 2.4.3 Measuring Runtime 27

3 Cost Based Filtering for Knapsack Constraints **29**
- 3.1 Constrained Knapsack Problems 30
- 3.2 Applications for Constrained Knapsack Problems 31
- 3.3 Constrained Knapsack vs. Pure Knapsack Problems 32
  - 3.3.1 Variable Fixing for the Constrained Knapsack Problem 32

3.3.2 Upper Bounds for Knapsack Problems 33
3.3.3 Reduction Techniques for Knapsack Problems 35
3.4 A Fast Filtering Algorithm based on Bound U_1 and U_2 36
3.5 More Effective Cost Based Filtering using Bound U_3 38
3.6 Cost Based Filtering for Special Constrained Knapsack Problems 38
3.6.1 Multi-Dimensional Knapsack Problems 39
3.6.2 Bounded Knapsack Problems 39
3.7 Numerical Evaluation 40
3.7.1 Test Environment 40
3.7.2 The Opponents 41
3.7.3 Experimental Results 41
3.8 Conclusions 45

4 Airline Crew Rostering 47
4.1 Solving Airline Crew Rostering Problems 47
4.1.1 The Airline Crew Assignment Problem 48
4.1.2 Current Solution Methods 49
4.1.3 The Airline Test Case 50
4.2 A Column Generation Model for the CAP 51
4.2.1 The Subproblem 51
4.2.2 Implementation 57
4.2.3 The Master Problem 60
4.3 Overview of the Entire Approach 61
4.4 Numerical Results 62
4.5 Conclusions 65

5 Hybrid Approaches for Airline Crew Rostering 67
5.1 The Airline Test Cases 67
5.2 Heuristic Tree Search Constraint Programming Approach 68
5.2.1 Tree Traversal 69
5.3 Constraint Programming based Column Generation Approach 69
5.3.1 Problems with Column Generation 70
5.4 Integrating the Approaches 71
5.4.1 First Way of Integration: Transforming a Set Covering into a Set Partitioning Solution 71
5.4.2 Second Way of Integration: Generating Combinable Columns and Exploiting Dual Values 74
5.5 Numerical Results 77
5.6 Combining Both Hybrid Methods 79
5.7 Conclusions 80

6 Home Health Care 83
6.1 Introduction 83
6.1.1 Specific Requirements 84
6.1.2 Literature Review 85
6.2 A Mathematical Model for the Home Health Care Problem 85

6.2.1 Parameters and Notation . . . 86
6.3 Solving Home Health Care Problems . . . 90
6.3.1 Preprocessing . . . 91
6.3.2 Sequencing a Roster . . . 91
6.3.3 Initial Solutions via Constraint Programming . . . 93
6.3.4 Improvement Heuristics . . . 94
6.4 A Hybrid Solution Approach . . . 97
6.4.1 The Solution Pool Concept . . . 97
6.4.2 Using a Good Solution to Improve the Search . . . 98
6.4.3 Using the Essence of All Solutions . . . 99
6.5 Numerical Evaluation . . . 100
6.5.1 Optimal Sequences . . . 101
6.5.2 Initial Solutions via Constraint Programming . . . 102
6.5.3 Comparing the Heuristics . . . 103
6.5.4 Combining Approaches . . . 103
6.6 Conclusions . . . 105

7 The Automatic Recording Problem 107
7.1 An Integer Linear Programming Formulation . . . 108
7.2 Solving the ARP via CP based Lagrangian Relaxation . . . 109
7.2.1 Substructures of the ARP . . . 110
7.2.2 CP based Lagrangian Relaxation for the ARP . . . 110
7.2.3 Implementation Details . . . 111
7.3 Numerical Results . . . 112
7.3.1 Test Instance Generation . . . 112
7.3.2 Numerical Results for Lagrangian Coupling . . . 113
7.4 Conclusions . . . 117

8 More Efficient Approaches for the Automatic Recording Problem 119
8.1 Solving the ARP via Branch-and-Cut . . . 119
8.1.1 Valid Inequalities for the Vertex Cover Polytope . . . 120
8.1.2 Valid Inequalities for the Knapsack Polytope . . . 122
8.2 Solving the ARP via Dynamic Programming . . . 124
8.2.1 A Simple Approach . . . 125
8.2.2 An Improved Dynamic Program . . . 125
8.3 Results . . . 127
8.3.1 Numerical Results for Branch-and-Cut and Dynamic Programming . . . 127
8.3.2 Comparing the Approaches . . . 131
8.4 Conclusion . . . 131
8.4.1 The Branch-and-Cut Approach . . . 132
8.4.2 The Dynamic Programming Approach . . . 132
8.5 Extending the Basic Model . . . 132
8.5.1 Avoiding Multiple Copies . . . 133
8.5.2 Using Multiple Recording Units . . . 133
8.5.3 Using Multiple Storage Units . . . 134

9 Domain Filtering for Maximum Clique 137
9.1 Introduction . . . 137
9.1.1 Literature Review . . . 138
9.1.2 Notation . . . 138
9.2 Branch-And-Bound Algorithms for Maximum Clique . . . 139
9.2.1 The Carraghan/Pardalos Algorithm . . . 139
9.2.2 The Östergård Algorithm . . . 139
9.3 Domain Filtering for Maximum Clique . . . 140
9.3.1 A Domain Filtering Algorithm . . . 141
9.4 Upper Bounds vs. Domain Filtering: Analytical Results . . . 143
9.4.1 Integer Programming Models for Maximum Clique . . . 143
9.4.2 Combinatorial Bounds . . . 145
9.4.3 Upper Bounds Dominated by Domain Filtering . . . 149
9.4.4 Upper Bounds Not Dominated by Domain Filtering . . . 149
9.4.5 Computational Complexity . . . 150
9.5 Conclusions . . . 151
9.5.1 Extensions to the Weighted Case . . . 151

10 Two Adapted Branch-and-Bound Algorithms for Maximum Clique 153
10.1 Upper and Lower Bounds by Coloring Heuristics . . . 153
10.1.1 Lower Bounds . . . 155
10.2 Improving the Algorithms . . . 155
10.2.1 Adapting the Carraghan and Pardalos Algorithm . . . 155
10.2.2 Adapting the Östergård Algorithm . . . 155
10.2.3 The Implementation . . . 155
10.3 Numerical Results on Benchmark Graphs . . . 156
10.3.1 Domain Filtering . . . 156
10.3.2 Coloring Bound $\mathcal{U}_8$ . . . 157
10.3.3 Primal Heuristics . . . 160
10.3.4 Comparison to other Algorithms . . . 160
10.3.5 Statistics on Large Sample Sets . . . 162
10.4 Numerical Results on Random Graphs . . . 163
10.5 Conclusions . . . 164

11 Conclusions and Open Questions 169
11.1 Conclusions . . . 169
11.2 Open Questions . . . 170

Appendix 175

A Numerical Results for the ARP 175
A.1 CP based Lagrangian Relaxation . . . 175
A.2 Dynamic Programming and Branch-and-Cut . . . 182

B Numerical Results for Maximum Clique **191**
B.1 Characteristics of the DIMACS Benchmark Set 191
B.2 Detailed Results for the DIMACS Benchmark Set 192
B.3 Single Statistics 195
B.4 Combined Statistics 216
B.5 Random Graphs 218

List of Figures **225**

List of Tables **229**

List of Algorithms **233**

Own Publications **235**

Bibliography **237**

1

Introduction

In recent years interest in the integration of techniques from Operations Research (OR) and Constraint Programming (CP) has been growing enormously. Researchers in both fields realize that developments made in the other field can be used beneficially in their own area of work. A prominent example in CP is the work of Régin [172] who showed that matching algorithms can efficiently and effectively filter domains in the AllDiff constraint. OR on the other hand has recognized CP's contributions to systematic tree search and to domain filtering.

Combined CP and OR techniques were shown to be successful in various fields. Scheduling problems, e.g., can be solved more efficiently when combining linear programming from OR and domain filtering from CP (see e.g. Baptiste et al. [18]). Other academic or real-life applications include vehicle routing and dispatching (e.g. in Caseau and Laburthe [49], Rousseau et al. [179]), staff rostering (e.g. Rousseau et al. [180], Fahle et al. [73]), or air traffic flow management (Barnier and Brisset [22]), respectively.

1.1 Integrating Concepts from CP and OR

In this thesis we aim to combine OR and CP concepts for solving NP-hard optimization problems. When speaking of OR approaches, we refer mainly to integer linear programming (ILP). The success story of (integer) linear programming was started after World War II by the pioneering work of Dantzig (primal simplex, 1947). Many landmark contributions followed his work. We refer to Dantzig [58] for an overview of the development up-to 1963, and to Maros and Mitra [139] for the more recent developments, including interior point methods (Karmarkar [126]). An overview of integer programming is provided by the books by Nemhauser and Wolsey [156] and Wolsey [211]. We also refer to Bixby [35] for the exciting computational progress of ILP during the last decade.

The work of Montanari [153] and Waltz [207] is usually considered as the root of constraint programming. Montanari studied properties of constraint networks and provided a formal treatment of path consistency. Waltz used constraint programming ideas for the scene

labeling problem, i.e. the visual interpretation of line drawings. Later, Mackworth [137] studied algorithmic aspects of node-, arc-, and path-consistency, respectively. He proposed algorithms AC-1, AC-2 and AC-3 to achieve arc-consistency, and some algorithm for node- and path-consistency. The book by Tsang [201] gives an introduction to the foundations. Some interesting and successful new techniques introduced recently include advanced consistency algorithms like AC-5 (Van Hentenryck et al. [202]) or AC-6 (Bessière [31]), and discrepancy-based tree traversal techniques like LDS (Harvey and Ginsberg [105]) or DDS (Walsh [206]).

1.1.1 Basic Concepts of Integer Linear Programming and Constraint Programming

ILP techniques require to model a problem P by linear inequalities bringing down the problem's structure to a description suitable to efficient numerical linear algebra methods.

$$\begin{array}{llll} P: & \text{minimize} & \sum_{j=1}^{n} c_j x_j & \\ & \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \geq b_i, & i = 1, \ldots, m \\ & & x_j \in \mathbb{N}, & j = 1, \ldots, n \end{array} \tag{1.1}$$

Choosing the right model can make the difference between solving or not solving the problem (see Williams [209]). We will touch on that topic in Section 2.2.2.2 when discussing column generation, and in Chapters 7 and 8 when developing a tight formulation for the automatic recording problem.

Efficiency of ILP approaches to (1.1) depends on the bounds found for the problem. Actually, improving bounds often corresponds to solving the problem more efficiently. Adding suitable cuts to (1.1) e.g. may improve the lower bound, using primal heuristics may improve the upper bound of a minimization problem. Given these bounds an ILP can be solved by an intelligent enumeration of the search space (e.g. by branch-and-bound).

Besides using LP-relaxations, Lagrangian relaxation is quite popular in ILP, since it is simple and yet provides lower bounds that are at least as tight as LP bounds. Column generation, to mention another technique used in this thesis, is used to solve huge models for which tight bounds are known, but which cannot be solved explicitly.

Constraint programming models a problem P as a relation of some (arbitrary) constraints $C_1, \ldots, C_m$ over variables $x_1, \ldots, x_n$ with corresponding domains $D(x_1), \ldots, D(x_n)$. Using a notation similar to (1.1) we can denote a constraint program as:

$$\begin{array}{llll} P: & \text{find} & x_1, \ldots, x_n & \\ & \text{subject to} & C_i(x_1, \ldots, x_n) = \text{true}, & i = 1, \ldots, m \\ & & x_j \in D(x_j), & j = 1, \ldots, n \end{array} \tag{1.2}$$

A solution of P is a consistent selection of values for the variables, such that all domains and all constraints are respected. In order to solve a constraint program, values which are not consistent with the current domain and constraint structure are removed. Whenever a domain is not unique[1] after eliminating inconsistent values, we branch on that domain within a tree search.

[1] If we use set variables, we branch whenever the domain contains values which have not been fixed or excluded yet. We will later use the notion of a *required set* and a *possible set* to define this in detail.

CP	ILP
Idea: *Branch-and-Prune*	Idea: *Branch-and-Bound*
Branch: Decompose problem. Use heuristics based on feasibility information	*Branch:* Decompose problem. Use information from relaxations
Prune: Examine constraints to eliminate infeasible configurations and to reduce possible variable values	*Bound:* Use (linear) relaxation of the problem (enhanced by additional cuts), eventually use primal heuristics
Main focus: constraints and feasibility	*Main focus:* objective function and optimality

Table 1.1: *A comparison of CP and ILP adapted from Trick [200].*

Therefore CP is interested in finding efficient algorithms to detect and remove inconsistent assignments for different constraint classes. In doing so, CP treats problems combinatorially, thus preserving much of the problem's original structure. Not surprisingly, efficient combinatorial algorithms were adapted for this particular use. We mentioned already matching techniques, and we will present domain filtering via a shortest path algorithm in Sec. 4.2.1. Mehlhorn [149] motivates further incorporation of graph algorithms into CP from an algorithmic perspective.

By construction, ILP methods view a problem globally, taking into account all variables and usually more than one or even most constraints at a time. By calculating upper and lower bounds on the costs, they show a good ability to identify promising parts of the search space. However, they often suffer from minor local conflicts, which might prevent a feasible solution from being found. On the other hand, CP methods can efficiently handle feasibility problems by resolving local conflicts using algorithms based on arc-consistency and advanced search techniques. Respectively, CP methods lack the ability to view the variables and constraints of a problem globally. Therefore, they often have problems when stuck in local optima.

Moreover, ILP methods are good for optimizing. Using the bounds they are guided to the promising parts of the search space. CP is good for finding feasible solutions. By removing inconsistent values from the variables' domain, they quickly find feasible regions of the search space.

Our discussion is summarized in Table 1.1 adapted from Trick [200]: Branching is a common idea used in both, CP and ILP. Whereas ILP methods use numerical information (bounds and estimations) to decide on the next branching step, CP applies heuristics based on feasibility information. ILP does not investigate parts of the search tree which cannot improve the incumbent solution and is thus focused on optimality. CP on the other hand discards inconsistent regions and enforces feasibility. A more detailed discussion of the concepts, similarities and differences of CP and OR is given by Heipcke [107].

Having seen the advantages and drawbacks of OR and CP concepts, we aim at combining ideas from both fields in order to improve on existing algorithms. In this context, two questions automatically occur, and we will answer them in the next two sections:

- Strategically: *What are the benefits of an integration?*
- Tactically: *What does an integration look like?*

1.1.2 Strategic Considerations

There are several motivations for combining CP and OR methods. The first to mention are:

- *Increase efficiency*, and thus increase performance of a solution approach.
- *Increase robustness*, and thus open the approach for a wider range of problem characteristics.

Both topics are driven by the idea that we may compensate for the weakness of the one method by the merits of the other. We may combine relaxations for finding bounds, and CP consistency techniques for detecting local conflicts (classical ILP preprocessing follows this path, see Andersen and Andersen [4], without using the full power of symbolic constraints). We may use LPs to handle linear constraints of a CP model rather than solving these constraints over continuous domains (see Brucker and Knust [41] for an example in scheduling). Or we may use CP for combinatorial structures which cannot be described well in ILP models or are known to be numerically instable (Padberg [162] gives examples for numerical instabilities when using cuts in ILPs).

Another motivation is important in many real-life applications where rules and regulations of a given problem are subject to changes. In crew rostering, e.g., companies have to adapt their planning tools to frequent changes in legislation and contractual agreements, whereas the basic planning scenario is not touched. A hybrid CP/OR approach can

- *Increase flexibility*, and thus reduce maintenance for a software solution.

Since CP supports various (non-linear) constraints and encapsulates them, it is often easier to change CP constraints than to re-model an ILP. This may imply loosing part of the efficiency compared to a 'hard-wired' solution. But we learned from industrial partners that companies tend to partly sacrifice efficiency for a gain in flexibility.

There are other reasons for combining CP and OR, but we do not consider them here (modeling and software engineering aspects, etc.). Throughout this thesis our major aim is to increase efficiency and robustness by combining CP and OR methods. Flexibility issues, in addition, are considered later on in the chapters on crew rostering.

1.1.3 Tactical Considerations

In order to improve on CP's performance for optimization problems, we can filter domains according to optimization considerations. I.e., values of a domain are removed if they are inconsistent with the current domain and constraint structure (classical CP), or if their use will lead to an objective value which does not improve the incumbent value. This idea is known as *cost based domain filtering*. It was introduced by Focacci et al. [81] and later extended to incorporate bounds as well in Fahle and Sellmann [77]. In this thesis the shortest path constraint, the knapsack constraint, and the filters for maximum clique follow this idea. Other approaches embed OR methods in a CP framework, thus using them as specialized constraints (Beringer and De Backer [28], Bockmayr and Kasper [36], Rodosek et al. [176] did this for a linear solver).

Column generation can be improved by using CP within the subproblem. Especially, in the presence of complex side-constraints, leading to a very sparse solution space, CP is a good

choice to find these few feasible solutions. We will discuss *CP based Column Generation* in detail for airline crew rostering problems. Cut generation is the dual counterpart of column generation. Focacci et al. [82] report on generating cutting planes via CP for an ILP to solve an asymmetric traveling salesman problem. In their approach cuts could be found faster using domain filtering on the corresponding separation problem.

Often a problem is composed by several different constraint classes. *CP based Lagrangian relaxation* allows for relaxing parts of these classes such that only one class remains for which an efficient (cost based) domain filtering algorithm is available. The technique is suitable for CP and ILP. The latter benefits from having more variables assigned a correct value which e.g. leaves less room for wrong branching decisions. We developed CP based Lagrangian relaxation and applied it to the automatic recording problem.

An integration on a macroscopic level is very often a simple, yet good approach to improve convergence of either CP or OR. Whenever an OR method requires a starting solution, CP can be used to deliver a feasible one. Whenever CP requires the improvement of a solution, OR methods can be used to improve the current solution (e.g. by local search). In Chapter 5 we will discuss how to improve a column generation approach to the airline crew rostering problem by these two principles.

Many more combinations are possible, especially if we do not insist on exact methods. E.g., combining metaheuristics and CP is a very promising area. The idea is to control the search by e.g. Tabu Search and to explore a given neighborhood systematically via CP. If the neighborhood contains only few feasible solutions, consistency checks typically find these much more efficiently than less intelligent mechanisms. We refer to Pesant and Gendreau [166, 167], Caseau and Laburthe [49], Shaw [190], Shaw et al. [191], Prestwich [171] for an insight into this topic, and to Hooker [112] and Milano [151] for a general overview.

1.2 Contribution of the Thesis

We develop integrated CP and OR approaches and apply them to four optimization problems. Two of them deal with complex staff rostering problems, one considers a multimedia application and the last one deals with the maximum clique problem. We study theoretical and numerical aspects of these. We will briefly describe the problems and the approaches in the next pages. Most of the work has been developed and published in co-operation with other researchers (see p. 235 in the appendix). Unless otherwise noted, all authors contributed equally to ideas, further development, implementation and tests in the published work.

1.2.1 Airline Crew Rostering, Shortest Path Constraint and CP based Column Generation

Within the EU ESPRIT Project PARROT[2] we dealt with the *Airline Crew Rostering Problem*. Airline crew rostering problems are large-scale optimization problems which can be adequately solved by column generation. The subproblem is typically a so-called constrained

[2] See [165]. Industrial partners in the project were: Carmen System (Sweden), Ilog SA (France), Olympic Airways (Greece) and Lufthansa Systems (Germany). Academic partner was the University of Athens.

shortest path problem and is solved by dynamic programming. However, complex airline regulations arising frequently in European airlines cannot be expressed entirely in this framework and limit the use of pure column generation. We formulate the subproblem as a constraint satisfaction problem, thus gaining high expressiveness. Each airline regulation is encoded by one or several constraints. An additional constraint which encapsulates a shortest path algorithm for generating columns with negative reduced costs is introduced. This constraint reduces the search space of the subproblem significantly. Resulting domain reductions are propagated to the other constraints which additionally reduces the search space. Numerical results based on data of a large European airline are presented and demonstrate the potential of our approach. Chapter 4 will describe this work in detail. The chapter follows our publication

> [73] T. Fahle, U. Junker, S. E. Karisch, N. Kohl, M. Sellmann, B. Vaaben. Constraint Programming Based Column Generation for Crew Assignment. *Journal of Heuristics*, 2002.

The basic idea of that work is to use CP for modeling complex airline crew planning rules and to apply column generation for the optimization. This idea was developed within the PARROT project by Ulrich Junker (Ilog SA) and Niklas Kohl (Carmen Systems). The idea was further studied by Bo Vaaben and Stefan Karisch (both Carmen Systems) for the CP part, and by the author of this thesis for the OR part. Later, the approach was combined with additional modules developed simultaneously in Paderborn. The concept itself was presented at the *CP'99 conference* as

> [125] U. Junker, S. E. Karisch, N. Kohl, B. Vaaben, T. Fahle, M. Sellmann. A Framework for Constraint Programming Based Column Generation. *CP'99*.

An introduction to the shortest path constraint developed [125] is given in Sec. 4.2.1.

Within the PARROT project, a different approach to the Airline Crew Assignment problem was developed by the University of Athens [194]. It performs a CP based heuristic tree search. In Chapter 5 we present their approach and show how the CP based column generation and the heuristic tree search approach can be coupled to overcome their inherent weaknesses. Numerical results show the superiority of the hybrid algorithm in comparison to a CP based tree search or column generation alone. Concepts for the hybrid approach were developed by all four authors, the implementation needed for coupling the existing main methods by the first and the second author of

> [189] M. Sellmann, K. Zervoudakis, P. Stamatopoulos, T. Fahle. Crew Assignment via Constraint Programming: Integrating Column Generation and Heuristic Tree Search. *Annals of Operations Research*, 2002.

1.2.2 Automatic Recording Problem, Knapsack Constraint and CP based Lagrangian Relaxation

Cost based domain filtering algorithms are only suitable for the problems for which they were designed. Models for new combinatorial optimization problems can frequently be composed by a combination of simpler structured problems. Domain filtering techniques for the substructure can be applied in such a case. In so doing, we partly ignore the interference of the

substructures and therefore a stronger coupling of domain filtering would be desirable. We show how cost based domain filtering for linear optimization problems can be coupled via Lagrangian relaxation in Sec. 2.3.3. This method is applied to a multimedia problem incorporating a knapsack and a maximum weighted stable set problem.

The corresponding domain filtering algorithms for the knapsack constraint are developed in Chapter 3. They are used as domain filtering routines for the Constraint Knapsack Problem with $\Theta(n \log n)$ preprocessing time and $\Theta(n)$ time per call. This sums up to an amortized time $\Theta(n)$ for $\Omega(\log n)$ calls when used within a tree search. These runtimes improve on other proposals in literature. Chapter 3 is essentially published as:

> [77] T. Fahle and M. Sellmann. Cost based filtering for the constrained knapsack problem. *Annals of Operations Research*, 2002.

A linear time algorithm for the second substructure of the multimedia problem was developed by M. Sellmann. It was published together with results of the multimedia application as

> [186] M. Sellmann and T. Fahle. Coupling variable fixing algorithms for the automatic recording problem. *ESA'01*.

The presentation in Chapter 7, however, gives more details on the coupling and a deeper experimental analysis. The corresponding publication is

> [187] M. Sellmann and T. Fahle. Constraint programming based Lagrangian relaxation for the automatic recording problem. *Annals of Operations Research*, 2003.

Our discussion of the automatic recording problem is finalized by Chapter 8, where we present alternative approaches to the problem. These very recent results show that dynamic programming and branch-and-cut, respectively, are substantially faster than CP based Lagrangian coupling approaches. We discuss this result and its implications. Furthermore, we show how to extend our mathematical model of the automatic recording problem to more realistic scenarios.

1.2.3 Home Health Care and Domain Filtering for Sequences

Home health care, i.e. visiting and nursing patients in their homes, is a growing sector in the medical service business. From a staff rostering point of view, the problem is to find a feasible working plan for all nurses that has to respect a variety of hard and soft constraints, and preferences. Additionally, home health care problems contain a routing component: A nurse must be able to visit his/her patients using a car or public transport. It is desired to design rosters that consider both, the staff rostering and the vehicle routing components while minimizing transportation costs and maximizing satisfaction of patients and nurses.

In Chapter 6 we present the Home Health Care Problem and a solution approach developed in the BMBF project PARPAP[3]. In principle, optimization techniques used in airline crew

[3] see [164]. Partners in the project are: For the computer based solution approach: Unilog Integrata (Bremen), SHokWare (Bremen) and the University of Paderborn. Ergonomics are studied by the University of Karlsruhe. Three home health care companies located in Bremen act as end-user: Arbeiter Samariter Bund, Bremer Pflegedienst and Ambulante Kranken- und Seniorenbetreuung Bremen.

rostering can be adapted to the scenario considered here. Real-world constraints, however, limit the successful application of column generation in practice. Whereas airline companies are willing to spend 20 hours of computation time (or more) to find a best possible solution, home health care companies prefer good solutions within, say, 10–15 minutes. Therefore, a change in the solution paradigm was necessary.

We develop a generic and flexible mathematical model for home health care problems. Based on this model an algorithm using CP and LP was built to solve the underlying sequencing — a constraint traveling salesman problem with time windows. A combination of CP and Tabu Search is used to efficiently explore the search space of an instance. The overall concept is able to adapt to various changes in the constraint structure, thus providing the flexibility needed in a generic tool for real-world settings. The model is more generic than models proposed in literature. Furthermore, much larger problem instances can be tackled by this approach than by alternative approaches.

The work presented in Chapter 6 was developed by Stefan Bertels and the author of this thesis. Stefan worked as a student programmer and recently finished his master thesis on this topic [29]. The chapter is part of a joint paper submitted to a special issue volume on "Staff Scheduling and Rostering":

> [30] S. Bertels, T. Fahle. A Hybrid Setup for a Hybrid Scenario: Combining Heuristics for the Home Health Care Problem. submitted to the *Annals of Operations Research*.

1.2.4 Maximum Clique and Comparing OR Bounds to CP Domain Filtering

The last two chapters are dedicated to the maximum clique problem. We develop cost based filtering techniques for the so-called *candidate set* (i.e. a set of nodes which can possibly extend the clique in the current choice point) and investigate it theoretically and numerically.

In Chapter 9 we propose a model in which upper bounds used in OR can be compared to cost based domain filtering techniques from CP. In particular, we can prove that our domain filtering is as least as tight as 7 out of 8 combinatorial bounds, and we discuss the connection between domain filtering and LP bounds. Furthermore, we present a taxonomy of 10 upper bounds used in OR — to our knowledge, such a taxonomy has not been published before.

In a numerical study in Chapter 10 we use two branch-and-bound algorithms from literature. We enhance them by domain filtering and some lower and upper bounds techniques. Numerical results demonstrate that the combination of the new cost based filtering and upper and lower bounds outperforms the original approach as well as approaches that only apply either of these techniques. Furthermore, the enhanced algorithm is competitive when compared to other recent algorithms for maximum clique. An earlier version of our results was presented at ESA'02.

> [71] T. Fahle. Simple and Fast: Improving a Branch-and-Bound Algorithm for Maximum Clique. *ESA'02*.

The theoretical classification of LP bounds, primal heuristics, and the second branch-and-bound algorithm was later added to the work which then was submitted as

> [72] T. Fahle. Domain Filtering for Maximum Clique. submitted to the *Annals of Operations Research*.

2

Basic Concepts

Throughout this thesis, our scenario is the following: We are given a finite number of constraints and we aim at finding a feasible solution to these constraints. If in addition, a linear objective function is provided, we aim at finding a best possible feasible solution. We are interested in NP-hard problems which can be modeled by integer linear or constraint programs.

This chapter briefly recalls the basic techniques from CP and OR used in later chapters. We have to refer to relevant literature for a more detailed overview. We start with presenting concepts of constraint programming and optimization. We define cost based filtering and CP based optimization technique thereafter and finally, we comment on the experimental methodology used in later chapters.

2.1 Constraint Programming

The building blocks of a constraint programming approach are *consistency techniques* which tighten the problem, *heuristics* which guide the search and some *tree search control*, that defines the way in which a systematic search is performed. We describe consistency and tree search ideas in the following pages. More details are given e.g. in the books by Tsang [201] or by Marriot and Stuckey [140]. Furthermore, we refer to the overview by Smith [192], Barták [23], [24], Kumar [132], Apt [8]. Our presentation here follows Barták [23], [24] and Mackworth [137].

Definition 1 *Let* $\mathbb{T} = \{\text{true}, \text{false}\}$. *A* constraint program *consists of a tuple of variables* $\mathcal{X} = (x_1, \ldots, x_n)$, *a set* $D(x_j)$ *denoting the possible set of values (the* domain*) of each variable* $x_j \in \mathcal{X}$, $\mathcal{D} = (D(x_1), \ldots, D(x_n))$. *Furthermore, we are given constraints* $\mathcal{C} = \{C_1, \ldots, C_m\}$, $C_i : (D(x_1) \times \cdots \times D(x_n)) \to \mathbb{T}$, $i = 1, \ldots, m$. *The* constraint satisfaction problem (CSP) $(\mathcal{C}, \mathcal{D}, \mathcal{X})$ *asks for a vector* $(v_1, \ldots, v_n) \in (D(x_1) \times \cdots \times D(x_n))$ *such that*

$$\forall i = 1, \ldots, m \quad C_i : (v_1, \ldots, v_n) = \text{true}.$$

Note that the definition above is not limited to special constraint classes (linear, convex, etc.) as often the case in OR. Values in domains, in addition, do not need to be consecutive numbers. Infinite domains can be handled, though throughout this thesis domains will be finite.

Variables can represent integer or float values as well as set or structured data types. CSPs are usually solved by applying domain reduction techniques. Therefore, each variable has a *current domain* $\mathrm{dom}(x)$ that is initialized by $D(x)$. If x is an integer or float variable, we denote the smallest element of $\mathrm{dom}(x)$ by $\min(x)$ and the largest element by $\max(x)$. We also call them the *lower* and *upper bounds* for x. The value of a set variable is a set of integers selected from an initially given domain. The current domain of a set variable is defined by a lower and an upper bound, which are also called required set $\mathrm{req}(Y)$ and possible set $\mathrm{pos}(Y)$. The value of the set variable has to be a superset of $\mathrm{req}(Y)$ and a subset of $\mathrm{pos}(Y)$. Set variables replace an array of boolean variables and lead to more compact constraint models. We will often use $\underline{\overline{x}}$ to denote some feasible value from $\mathrm{dom}(x)$ or $\mathrm{pos}(x)$, respectively.

We often neglect those variables from a constraint's definition that are not affected by the constraint. E.g. instead of writing

$$C : D(x_1) \times \cdots \times D(x_n) \to \mathbb{T}, \quad C(x_1, \ldots, x_n) \mapsto x_i < x_j \tag{2.1}$$

we write

$$C : D(x_i) \times D(x_j) \to \mathbb{T}, \quad C(x_i, x_j) \mapsto x_i < x_j \tag{2.2}$$

In the remaining of the thesis we will speak of an *unary constraint*, if the constraint involves only one variable, or a *binary constraint*, if it involves two variables after removing unnecessary variables. A unary constraint for variable i is denoted by $C^{(i)}$, and a binary constraint involving x_i, x_j, $i < j$, is denoted by $C^{(i,j)}$. This labeling is unique since any two constraints $C' : D(x_i) \times D(x_j) \to \mathbb{T}$, $C'' : D(x_i) \times D(x_j) \to \mathbb{T}$ operating on the same variables, can be replaced by a single constraint $C : D(x_i) \times D(x_j) \to \mathbb{T},\ C(x_i, x_j) \mapsto C'(x_i, x_j) \wedge C''(x_i, x_j)$.

CSPs can model various NP-hard problems and thus are NP-hard themselves. They can be solved via a *generate-and-test* approach, i.e. we enumerate all vectors. We generate a first vector $(v_1, \ldots, v_n) \in (D(x_1) \times \cdots \times D(x_n))$ and if all constraints are satisfied, we stop. Otherwise we generate the next vector. We can do better by a *test-and-generate* approach. I.e. after each assignment of a value to a variable we check whether this assignment still can be extended to a feasible solution. If not, we perform a *backtracking*, which means that we undo the last decision and continue. According to Barták [23] there are three major obstacles for pure backtracking (see also Mackworth [137]):

- thrashing, i.e. repeated failure due to the same reason,
- redundant work, i.e. conflicting values of variables are not remembered, and
- late detection of the conflict, i.e. a conflict is not detected before it really occurs.

Consistency techniques provide a means for coping with them all.

2.1.1 Consistency

A CSP $(\mathcal{C}, \mathcal{D}, \mathcal{X})$ only consisting of unary and binary constraints is often represented by a *constraint network* (Montanari [153]). A constraint network is a labeled, directed graph $N = (V, E)$. The variables in $\mathcal{X}$ are represented by the nodes in V, each with an associated set representing the variable's domain. Every unary constraint is represented by a self-loop. A binary constraint $C^{(i,j)}$, $i < j$, is represented by a labeled arc, directed from node i to node j and a labeled, directed arc from j to i. For convenience we assume that for each constraint $C^{(i,j)} : D(x_i) \times D(x_j) \to \mathbb{T}$, we have defined an artificial constraint $C^{(j,i)} : D(x_j) \times D(x_i) \to \mathbb{T}$, $C^{(j,i)}(x_j, x_i) \mapsto C^{(i,j)}(x_i, x_j)$. (The latter is needed for an easy notation of consistency algorithms). Using hyper-arc notation, constraints of higher arity can be treated accordingly.

Definition 2 (node / arc consistency, Mackworth, 1977) *Let $(\mathcal{C}, \mathcal{D}, \mathcal{X})$ denote a CSP, and $N = (V, E)$ be the corresponding constraint network.*

(i) A node $i \in N$ (i.e. variable $x_i \in \mathcal{X}$) is called node consistent *if for the unary constraint $C^{(i)} : D(x_i) \to \mathbb{T}$ it holds: $\forall \nu \in D(x_i) \;:\; C^{(i)}(\nu) = \text{true}$.*

(ii) An arc $(i, j) \in E$ is called arc consistent *if*

a) the corresponding nodes i and j are node consistent and

b) for the corresponding binary constraint $C^{(i,j)} : D(x_i) \times D(x_j) \to \mathbb{T}$ it holds:

$$\forall \nu \in D(x_i) \quad \exists \mu \in D(x_j) : \; C^{(i,j)}(\nu, \mu) = \text{true}.$$

(iii) A network is said to be node/arc consistent if every node/arc is node/arc consistent.

It is straightforward to extend the notion of arc consistency to constraints of higher arity which is often referred to as *hyper arc consistency*. Our definition follows Baptiste et al. [18]:

Definition 3 ((hyper) arc consistency for non-binary constraints) *Let $(\mathcal{C}, \mathcal{D}, \mathcal{X})$ denote a CSP. Let $C \in \mathcal{C}$ be a constraint, $C : D(x_1) \times \cdots \times D(x_n) \to \mathbb{T}$.*

(i) C is called (hyper) arc consistent *if*

a) all variables $x \in \mathcal{X}$ are node consistent and

b) for the constraint C it holds:

$$\forall i = 1, \ldots, n \quad \forall \nu \in D(x_i) \quad \forall j = 1, \ldots, n, \; j \neq i \quad \exists \mu_j \in D(x_j) : \\ C(\mu_1, \ldots, \mu_{i-1}, \nu, \mu_{i+1}, \ldots, \mu_n) = \text{true}.$$

(ii) A CSP is said to be (hyper) arc consistent if every variable $x \in \mathcal{X}$ is node consistent and every constraint $C \in \mathcal{C}$ is (hyper) arc consistent.

Other degrees of consistency can be defined via paths in the constraint network (path consistency, see Mackworth [137]) or by enforcing arc consistency for any subset of k variables (k-consistency). Barták [24] gives an overview of these and other notions. For the remaining part of this chapter we will mainly work with (binary) arc consistency, and we define a simple extension, which allows an easy handling of large domains:

Definition 4 (bound consistency) *Let $(\mathcal{C}, \mathcal{D}, \mathcal{X})$ denote a CSP, and $N = (V, E)$ be the corresponding constraint network. Furthermore, we require an ordering of values for each domain $D(x_i)$, $i = 1, \ldots, n$. We denote the smallest value of a domain by $\min(D(x_i))$, and the largest value by $\max(D(x_i))$.*

(i) A node $i \in N$ is called node bound consistent *if for the unary constraint $C^{(i)} : D(x_i) \to \mathbb{T}$ it holds:*

$$C^{(i)}(\min(D(x_i))) = \text{true} \quad \textit{and} \quad C^{(i)}(\max(D(x_i))) = \text{true}.$$

(ii) An arc $(i, j) \in E$ is called arc bound consistent *or* bound consistent *if*

a) the corresponding nodes i and j are node bound consistent and

b) for the corresponding binary constraint $C^{(i,j)} : D(x_i) \times D(x_j) \to \mathbb{T}$ it holds:

$$\forall \nu \in \{\min(D(x_i)), \max(D(x_i))\}\ \exists \mu \in D(x_j) : \ C^{(i,j)}(\nu, \mu) = \text{true}.$$

(iii) A network is said to be node (arc) bound consistent if every node (arc) is node (arc) bound consistent.

Bound consistency thus requires consistency of start and end of an interval rather than of any value in between. As done for arc consistency, bound consistency can be defined for non-binary constraints in a canonical way.

2.1.1.1 Consistency Algorithms

Node consistency can be achieved by a simple one-pass algorithm. For all variables $i = 1, \ldots, n$ we calculate $D(x_i)' = \{\nu \in D(x_i) \mid C^{(i)}(\nu) = \text{true}\}$ and replace the original domain by this set: $D(x_i) \leftarrow D(x_i)'$. Let $d = \max\{|D(x_i)| \mid i = 1, \ldots, n\}$ be the size of the largest domain. Then node consistency can be established by $O(d|V|)$ constraint checks.

We assume we have a node consistent network. In order to achieve arc consistency on this network, we have to ensure arc consistency for every arc.[1] We first give a procedure that makes an arc (i, j) consistent (procedure Revise(i, j) in Alg. 1). As long as it finds a value in $D(x_i)$ that does not have a counterpart in $D(x_j)$, such that the corresponding constraint is fulfilled, that value is removed. Afterwards, Def. 2*(ii)* holds for arc (i, j) (but not necessarily for the reverse arc (j, i)). Each call to Revise($\cdot, \cdot$) produces $O(d^2)$ constraint checks.

An arc (i, j) may not remain consistent as subsequent calls to Revise (for some arc (j, k)) may remove a value $\mu \in D(x_j)$ that was the only *support* for some $\nu \in D(x_i)$. Thus, removing μ requires a further check for arc (i, j).

Mackworth [137] proposed three different algorithms for establishing arc consistency, called AC-1, AC-2 and AC-3. All three start with a node consistent network $N = (V, E)$. In each round, AC-1 applies Revise(i, j) for all $(i, j) \in E$. Whenever a domain changes (i.e. Revise(i, j) returns true), Alg. 1 is called again for all arcs. We stop if we are in a steady state, i.e. no arc can be revised anymore. AC-1 thus requires $O(d^3|V|\,|E|)$ constraint checks.

[1] We present the consistency algorithms for binary arc consistency as done in the original papers. It is straightforward, to extend them to constraints of higher arity.

Algorithm 1 Revise an arc (i, j)

bool **Revise** (i,j)

1: *delete* ← false
2: **for all** $\nu \in D(x_i)$ **do**
3: **if** ($\nexists\, \mu \in D(x_j)$ such that $C^{(i,j)}(\nu, \mu) =$ true) **then**
4: $D(x_i) \leftarrow D(x_i) \setminus \{\nu\}$
5: *delete* ← true
6: **return** *delete*

Checking all arcs in each iteration is rarely necessary, especially, if only few domains have changed. AC-2 and AC-3 reduce the number of checks by testing only those arcs for which variables have been involved in a domain change in the previous round. AC-2 uses some special ordering here, whereas AC-3 works with queues. AC-2 is generalized by AC-3 and we present the latter in Alg. 2.

Algorithm 2 Arc consistency for a network of constraints: AC-3 (Mackworth [137])

AC-3

1: // *ensure that* $N = (V, E)$ *is node consistent*
2: $Q \leftarrow \{(i, j) \mid (i, j) \in E\}$
3: **while** $(Q \neq \emptyset)$ **do**
4: select $(i, j) \in Q$
5: $Q \leftarrow Q \setminus \{(i, j)\}$
6: **if** (Revise(i, j)) **then**
7: // $D(x_i)$ *was changed*
8: $Q \leftarrow Q \cup \{(k, i) \mid k = 1, \ldots, n \text{ and } (k, i) \in E\}$

AC-3 starts with filling the queue Q with all arcs in E (line 2). Then, each member of the queue is revised and deleted from Q. If a domain $D(x_i)$ changes, all arcs $(\cdot, i)$ might be affected by that change and are re-considered (line 8). If Q is empty, the algorithm terminates. In line 4 of Alg. 2 various strategies are possible, e.g. select constraints which can be checked quickly, select those which shrink the domain most, etc. The number of constraint checks for AC-3 is limited by $O(d^3|E|)$ (an analysis is given by Mackworth and Freuder [138]).

AC-3 still performs many unnecessary constraint checks, since often several values of $D(x_j)$ may support a certain value in $D(x_i)$. The first optimal algorithm for arc consistency, AC-4, was presented by Mohr and Henderson [152] and requires $\Theta(d^2|E|)$ constraint evaluations. By maintaining a list of *support values* AC-4 only revises arcs, if for some variable a value looses all its supports. As a drawback, AC-4 requires additional memory for the bookkeeping, and requires more complex initialization. Later developments like AC-6 (Bessière [31]) skip the idea of a complete support list. They only maintain one support value per domain. If that one is removed, a new support is searched. Thus, the initial effort of AC-4 is spread over the entire runtime and supporting values are only searched when needed.

Removing inconsistent values from a domain is often referred to as *domain filtering, domain reduction*, or (from an OR perspective) *tightening and variable fixing*. When establishing (arc) consistency constraints filter a domain, and thereby trigger other constraints that subse-

quently start filtering domains. The activity of transmitting these implications and carrying out their effects is called *(constraint) propagation*. Therefore, domain filtering methods are often referred to as *propagation algorithms*.

2.1.2 Tree Search

Arc consistency alone cannot guarantee finding a unique solution to an NP-hard problem. Therefore, the CP solution approach is typically embedded in a tree search. We start with the original problem and establish consistency there. If afterwards all domains contain exactly one value, we found a feasible solution and stop. If some domain is empty after establishing consistency, the current (sub)problem does not have a solution and we fail. Otherwise, we partition the current problem $(\mathcal{C}, \mathcal{D}, \mathcal{X})$ into k smaller subproblems

$$(\mathcal{C} \cup \mathcal{C}^{(1)}, \mathcal{D}, \mathcal{X}), \dots, (\mathcal{C} \cup \mathcal{C}^{(k)}, \mathcal{D}, \mathcal{X}) \tag{2.3}$$

such that no two subproblems overlap. More formally, let $S((\mathcal{C}, \mathcal{D}, \mathcal{X}))$ describe the solution space of $(\mathcal{C}, \mathcal{D}, \mathcal{X})$. Then we require

$$\bigcup_{i=0}^{k} S((\mathcal{C} \cup \mathcal{C}^{(i)}, \mathcal{D}, \mathcal{X})) = S((\mathcal{C}, \mathcal{D}, \mathcal{X})) \quad \text{and} \tag{2.4}$$

$$S((\mathcal{C} \cup \mathcal{C}^{(i)}, \mathcal{D}, \mathcal{X})) \;\cap\; S((\mathcal{C} \cup \mathcal{C}^{(j)}, \mathcal{D}, \mathcal{X})) \neq \emptyset \iff i = j \tag{2.5}$$

Using CP terminology such a subproblem is called a *choice point*[2]. Within a tree search all choice points are recursively treated as described above. A tree search in CP (as well as in OR) is defined by three parameters:

(i) Node ordering: Which subproblem do we explore next?

(ii) Variable ordering: On which variable (domain) do we branch next?

(iii) Value ordering: In which order do we assign the possible values?

We mentioned in the first chapter that tree search in CP is driven by heuristics. These heuristics are applied for *(ii)* and *(iii)* and try to find those assignments that lead to a feasible solution. For value ordering a typical strategy is to assign values that are more likely to be part of a feasible solution before assigning less promising values. If an objective is available, values improving the objective are often considered first.

By assigning more and more values to variables, and thus by diving deeper into the tree, more information of the problem's structure is discovered and more variables are fixed by domain filtering. Therefore a common strategy for variable ordering is to choose the domain with the smallest size first. In doing so, the upper part of the search tree will be small and intuitively, less wrong decisions have to be revised in the upper part of the tree.

[2] The corresponding term used in OR is a *branch-and-bound node*. We will use both terms synonymously.

2.1.2.1 Node Ordering

A variety of search methods for traversing the problem tree exists in literature. The oldest, most popular and, by far, most widely used search method is *Depth First Search (DFS)*. It explores the 'left' subproblem[3] of any problem first, thus performing a deep dive into the search tree. DFS is memory efficient as it stores $O(t)$ open problems at most in a search tree of depth t. The main drawback of DFS is that wrong decisions situated high in the search tree are revised late. If e.g. the first branching decision was wrong and we produce d branches per node, $O(d^{t-1})$ steps are required until the initial decision is revised.

The notion of *discrepancy* accounts for this phenomena. At a given node, a heuristic function decides which branch is most likely to contain a feasible solution (or a solution of good quality in case of an optimization problem). Always following the heuristic advice defines a unique path that is said to contain no discrepancies. Following the heuristic advice except for one case defines paths of one discrepancy. Within a tree search the discrepancy of a path from the root node to a leaf is defined as the number of right branches taken in that path.

Limited Discrepancy Search (LDS) is an iterative search method proposed by Harvey and Ginsberg [105]. It defines node orderings via discrepancy. In the i-th iteration, it explores all paths from the root node of the tree to a leaf that have i or less discrepancies. Such a tree traversal is built on the assumption that the heuristic method choosing the next variable and assigning the next value is good and produces few wrong decisions compared to an optimal variable selection. Hence, if we increase i and traverse the tree using LDS, intuitively, we should find a good or a best solution early.

Depth-Bounded Discrepancy Search (DDS) (Walsh [206]) refines the idea of LDS. In the i-th iteration, it explores all paths where discrepancies occur before depth i. I.e. a path with many discrepancies high in the tree is explored before a path with very few discrepancies low in the tree. This is justified by the assumption that heuristics tend to fail more likely at the top of the tree where less information of the problem structure is known.

2.2 Operations Research Approaches

Operations Research has developed several approaches for integer linear programs. We refer to some standard textbooks for the basics of linear programming and branch & bound (e.g. Papadimitriou and Stieglitz [163], Chvátal [55], Nemhauser and Wolsey [156], Wolsey [211]). Here, we only introduce some notation needed in succeeding chapters.

Let $A \in \mathbb{Q}^{n\times m}$ be the coefficient matrix, $b \in \mathbb{Q}^m$ the right hand side, $c \in \mathbb{Q}^n$ be the cost vector and x the solution vector. An integer linear program is given as

$$\begin{array}{rl} \min & \sum_{i=1}^{n} c_i \cdot x_i \\ \text{s.t.} & Ax \geq b \\ & x \in \mathbb{N}_0^n \end{array} \tag{2.6}$$

We obtain the linear programming (LP) relaxation of (2.6) by relaxing $x \in \mathbb{N}_0^n$ to $x_i \geq 0$, $i = 1, \ldots, n$. Whereas (2.6) is NP-hard, the LP relaxation of (2.6) can be solved in polynomial time e.g. by interior point methods (Karmarkar [126]).

[3]It is common practice to regard the branches under a node as ordered according to a heuristic function. Following the advice of a heuristic function means to go "left" down the search tree.

2.2.1 LP-based Branch-and-Bound

The term *branch-and-bound* was coined by Little et al. in their paper on the traveling salesman problem [136]. The method itself was already proposed in 1960 by Land and Doig [133]. They proposed an LP-based general purpose method for solving IPs like (2.6). LP-based branch-and-bound solves the LP relaxation of (2.6) and branches on non-integral variables. The optimal LP-values found are used for pruning. An outline of the method is given in Alg. 3. Q is the set of active subproblems (called *branch-and-bound nodes*[4]) together with their lower bound, x^* denotes the best feasible solution and z_{IP} is the corresponding global upper bound.

Algorithm 3 LP-based Branch-and-Bound for a minimization IP P

branch-and-bound

1: $Q \leftarrow (P, \infty)$, $z_{IP} \leftarrow \infty$, $x^* \leftarrow \emptyset$
2: **while** $(Q \neq \emptyset)$ **do**
3: select $(p, \cdot) \in Q$; $Q \leftarrow Q \setminus \{(p, \cdot)\}$
4: **if** (LP on p is feasible) **then**
5: $z_{LP} \leftarrow$ LP-value; $x \leftarrow$ LP-solution
6: **if** $(z_{LP} < z_{IP})$ **then**
7: **if** (x is integral) **then**
8: $x^* \leftarrow x$; $z_{IP} \leftarrow z_{LP}$
9: $Q \leftarrow \{(p', z') \in Q \mid z' < z_{LP}\}$ // *Bounding*
10: **else**
11: Let $S(p)$ denote the solution space of p.
12: partition p in subproblems $p_1, \ldots, p_k$ such that $S(p) = \bigcup_{i=1}^{k} S(p_i)$.
13: $Q \leftarrow Q \cup \{(p_1, z_{LP})\} \cup \cdots \cup \{(p_k, z_{LP})\}$ // *Branching*
14: **return** x^*

In the branching step quite often a variable x_i with a "most fractional" current value $\overline{x}_i \notin \mathbb{N}_0$ is chosen, i.e. $i = \arg\min_{i=1,\ldots,n}\{|\lfloor \overline{x}_i \rfloor + 1/2 - \overline{x}_i|\}$. Two branches P+"$x_i \leq \lfloor \overline{x}_i \rfloor$" and P+"$x_i \geq \lceil \overline{x}_i \rceil$", respectively, are created and stored in Q. Using depth-first-search in line 3 is very common in LP-based branch-and-bound. It requires only moderate memory resources, and can often find a good upper bound deep in the search tree. Furthermore, it allows to use the dual basis of the previous node for computing the LP bound of the current node, thus speeding up convergence by incremental calculations. *Best-first-search (BFS)* is another common node ordering strategy. It selects a node from Q that has a best lower bound. BFS often converges much faster than DFS. As a drawback, up-to $O(2^n)$ active nodes have to be stored in Q. More details of the basic LP-based branch-and-bound method are given e.g. in Nemhauser and Wolsey [156, pp. 355–367].

Many techniques were introduced to accelerate the convergence of LP-based branch-and-bound approaches. We mention three techniques that we refer to later on.

The first one is *reduced cost fixing*: Suppose that the LP relaxation of p is solved and the optimal value is z_{LP}. Furthermore, we know an integer solution value $z_{IP} > z_{LP}$. Let $\overline{c}_i$ denote the reduced cost of column $i = 1, \ldots, n$. If a non-basic variable x_i enters the basis, the

[4] playing the role of *choice points* in CP. We will use both terms synonymously.

objective value increases by at least $\overline{c}_i$. Thus, if $z_{LP} + \overline{c}_i > z_{IP}$, we can fix x_i to zero. Reduced cost fixing is used before partitioning the problem into smaller ones (line 11 of Alg. 3).

Probing follows a similar path. It considers all variables x_i not fixed so far and tentatively fixes them to some possible value $\alpha \in \mathbb{N}_0$. The resulting model is relaxed and solved as an LP. If no solution exists, or if the resulting objective value is worse than z_{IP}, we can exclude α from the domain of x_i. Probing is very time consuming since it requires to solve many LPs. Variants of probing consider only some promising candidates, or only perform some iteration of the LP solver in order to get an estimation of the change in costs. Probing can be applied only to the initial problem (before line 1 of Alg. 3) or to selected problems (line 12 of Alg. 3).

The third idea used to tighten bounds in branch-and-bound are *cuts*. Cuts are inequalities that are valid for the IP formulation, but not for the LP relaxation. Adding cuts thus forces the LP relaxation towards integral solutions. *Branch-and-cut* methods (see e.g. Bixby [35], Elf et al. [69], Johnson et al. [123], Martin [147], Padberg [162]) combine branch-and-bound with systematic cut generation. In a branch-and-cut approach new cuts are added to the current LP before line 4 of Alg. 3.

A detailed description of further techniques used in LP-based branch-and-bound is e.g. provided by Linderoth and Savelsbergh [135] and Johnson et al. [123].

2.2.2 Column Generation

Column generation is a well-known technique for handling linear programs with a huge amount of variables:

$$\min \sum_{j=1}^{n} c_j x_j \quad \text{s.t.} \quad A\,x = b, \quad x_i \geq 0, \; i = 1, \ldots, n, \qquad \text{and } n \text{ is huge} \tag{2.7}$$

Its origins date back to the works of Dantzig and Wolfe [60] and Gilmore and Gomory [88]. The latter paper applies column generation to the classical cutting stock problem where the subproblem is a knapsack problem. More recent applications include specially structured integer programs such as the generalized assignment problem and time constrained vehicle routing, crew pairing, crew assignment and related problems. We refer to Desrosiers et al. [65] for a survey.

Due to its size it is often impossible to solve the large system (2.7) directly. Column generation provides a way for obtaining the solution of this system indirectly. Therefore, a much smaller system $A'\,x = b$ is considered where A' contains only few columns of A. An optimal basic solution of the *restricted master problem* $A'\,x = b$ provides dual values λ_i of each constraint i of the linear system.

Now we pose the question: Which columns must be added to A' yielding a linear system with the same solution value as the original problem (2.7)? Linear programming duality theory tells us that only columns with *negative reduced cost* can be candidates for entering the basis (see Papadimitriou and Stieglitz [163], Theorem 2.8). This is the way the simplex algorithm chooses columns for its basis internally. But this fact can also be applied for the external generation of columns. Similar to the simplex algorithm, column generation can be stopped as soon as no further columns with negative reduced costs exist. Therefore, in the *subproblem* of column generation it is sufficient to search for a column $\alpha = (\alpha_1, \ldots, \alpha_m)^T$ that obeys the

negative reduced cost inequality

$$c_\alpha - \sum_{i=1}^{m} \lambda_i \cdot \alpha_i < 0 \tag{2.8}$$

where c_α denotes the cost coefficient for column α. α then is added to A'.

Theoretically, it may still be necessary to generate all possible columns before terminating the generation phase but in practice this rarely happens. Typically, only a small subset of all possible columns will be needed. Algorithm 4 describes the column generation idea. There, we assume that function `solveSubproblem()` returns new columns $\{\alpha^{(1)}, \ldots, \alpha^{(k)}\}$ which are valid solutions to the subproblem, or an empty set, if no more solutions that respect (2.8) exist. An initial matrix A' is returned by `getInitialColumns()`. `solveLP()` solves the LP given by $A'\,x = b$ and returns the corresponding dual values λ. Function `addColumn-sToMatrix()` extends A' by the columns generated.

Algorithm 4 Outline of the Column Generation

A' ←`getInitialColumns()`
repeat
  $\lambda \leftarrow$ `solveLP(`A'`)` // *get new duals*
  $\{\alpha^{(1)}, \ldots, \alpha^{(k)}\} \leftarrow$ `solveSubproblem(`λ`)` // *solve subproblem, respecting Equ. (2.8)*
  $A' \leftarrow$ `addColumnsToMatrix(`$A', \alpha^{(1)}, \ldots, \alpha^{(k)}$`)`
until $(\{\alpha^{(1)}, \ldots, \alpha^{(k)}\} = \emptyset)$

2.2.2.1 Column Generation and Integer Programming

Many problems can be modeled by an IP if we use (2.7) and add integrality requirements for some variables:

$$\min \sum_{j=1}^{n} c_j x_j \quad \text{s.t.} \quad A\,x = b, \quad x_i \in \mathbb{N}_0, \; i = 1, \ldots, n, \qquad \text{and } n \text{ is huge} \tag{2.9}$$

Unfortunately, we cannot apply column generation directly to (2.9) as linear programming duality theory is not valid for IPs. In this case, two general approaches are possible:

The first approach is to ignore that fact: One solves the continuous relaxation of the problem first, and then applies branch-and-bound to obtain an integer solution. For a range of IP problems it is known that the remaining gap between the solution values of LP and IP is small enough to be neglected. We use a variant of this approach for the airline crew rostering problem in Chapter 4.

Another possibility is to use *branch-and-price* where columns are generated in every nodes of a branch-and-bound tree and optimality can be proven. We refer to Barnhart et al. [19] for further information of this topic.

2.2.2.2 Why Column Generation Models?

Barnhart et al. [19] list four reasons for preferring a potential exponential number of variables to compact IP models:

- A compact formulation of a MIP may have a weak LP relaxation. Frequently the relaxation can be tightened by a reformulation that involves a huge number of variables.
- A compact formulation of a MIP may have a symmetric structure that causes branch-and-bound to perform poorly because the problem barely changes after branching. A reformulation with a huge number of variables can eliminate these symmetries.
- Column generation provides a decomposition of the problem into master and subproblem. This decomposition may have a natural interpretation in the contextual setting allowing for the incorporation of additional important constraints.
- A formulation with a huge number of variables may be the only choice [as no compact linear model is known].

2.2.3 Lagrangian Relaxation

Lagrangian Relaxation provides a means for calculating lower bounds on an optimization problem. Whereas linear relaxation drops the integrality requirement from an IP, Lagrangian relaxation relaxes parts of the constraints. These relaxed constraints are used as a penalty term in the objective function. Geoffrion [87], Held and Karp [108, 109] are classical references on Lagrangian relaxation and IPs. A good introduction into Lagrangian Relaxation in an IP context is given by Fischer [79]. Lagrangian Relaxation can be applied, however, to more general constraint classes. We refer to Lemaréchal [134] for details.

We start our introduction by a refinement of (2.6). Suppose, (2.6) consists of two constraint families, the one describing a simple structure ($\boldsymbol{Dx} \geq \boldsymbol{d}$), the other one describing a difficult structure ($\boldsymbol{Ex} \geq \boldsymbol{e}$):

$$\begin{array}{rl} \min & \sum_{i=1}^{n} c_i \cdot x_i \\ \text{s.t.} & \boldsymbol{Dx} \geq \boldsymbol{d} \\ & \boldsymbol{Ex} \geq \boldsymbol{e} \\ & \boldsymbol{x} \in \mathbb{N}^n \end{array} \tag{2.10}$$

We remove $\boldsymbol{Ex} \geq \boldsymbol{e}$ from the constraint set and introduce it into the objective as a penalty term. A *Lagrangian multiplier* $\lambda = (\lambda_1, \ldots, \lambda_m)^T \in \mathbb{R}^m_{\geq 0}$ is used to adjust the impact of the penalty term.

$$\begin{array}{rl} z(\lambda) = \min & \sum_{i=1}^{n} c_i \cdot x_i + \lambda(\boldsymbol{e} - \boldsymbol{Ex}) \\ \text{s.t.} & \boldsymbol{Dx} \geq \boldsymbol{d} \\ & \boldsymbol{x} \in \mathbb{N}^n \end{array} \tag{2.11}$$

Given some λ, we can solve (2.11) by some suitable algorithm and obtain a solution $\overline{\boldsymbol{x}} = (\overline{x}_1, \ldots, \overline{x}_n)$. It is easy to see that (2.11) is in fact a relaxation of (2.10) and that each choice of λ provides a lower bound on the original problem. We are interested in some λ^* that provides the tightest lower bound. Thus, we have to solve the *Lagrangian dual problem*:

$$\max_{\lambda \geq 0^m} z(\lambda) \tag{2.12}$$

Since $z(\lambda)$ is a convex function, *subgradient methods* can be used to solve (2.12). The idea of subgradient methods is to start with some λ and to move that λ into the direction of the steepest

ascent. Subgradients are used to guide the search since they point to the right direction. A subgradient $g = (g_1, \ldots, g_m)^T \in \mathbb{R}^m$ is defined as

$$g_i = e_i - \sum_{j=1}^{n} E_{ij}\bar{x}_j, \quad i = 1, \ldots, m. \tag{2.13}$$

Held and Karp [109] proposed the following update step:

$$\lambda_i^{(k+1)} \leftarrow \max\left\{0,\ \lambda_i^{(k)} + \alpha\frac{(ub - lb)g_i}{||g||^2}\right\}, \quad i = 1, \ldots, m \tag{2.14}$$

Using (2.14) it is easy to calculate a sequence $\lambda^{(1)}, \lambda^{(2)}, \ldots$ of Lagrangian multipliers that converges towards an optimal solution to (2.12). The initial multiplier $\lambda^{(0)}$ is chosen arbitrarily. The gap $(ub - lb)$ between upper and lower bound of the original problem adapts the update step to the progress of the optimization algorithm. The parameter α is halved if ub and lb have not changed for longer time. Whenever $||g|| = 0$, or α is smaller than some predefined constant, the iterative generation of new multipliers (2.14) stops and the best solution to (2.12) is returned as a lower bound.

Detailed studies on subgradient methods are given in Held et al. [110], Goffin [92] and Fischer [79] .

2.2.3.1 Why Lagrangian Relaxation?

The popularity of Lagrangian relaxation stems from several facts. Firstly, the bound found by Lagrangian relaxation is at least as good as the bound found by LP relaxation. This result is due to Geoffrion [87] who also showed that the Lagrangian bound equals the LP bound if (2.11) has the integrality property, i.e. if the LP relaxation of (2.11) is always integral. Secondly, Lagrangian relaxation methods are simpler than LP methods. They can be easily implemented and do not suffer as much from numerical problems as LP methods often do. Finally, Lagrangian relaxation methods are fast if the relaxed system (2.11) can be solved fast. This is in particular the case for 0-1 IPs that allow to relaxing all constraints (i.e. the constraint set "$Dx \geq d$" is empty). Then the remaining minimization problem

$$\begin{array}{rl} z(\lambda) = \min & \sum_{i=1}^{n} c_i \cdot x_i + \lambda(e - Ex) \\ \text{s.t.} & x \in \{0,1\}^n \end{array} \tag{2.15}$$

can be solved via simply checking the sign of x_i's coefficient. The best currently available solvers for set covering problems are based on this idea (e.g. Beasley [25], Caprara et al. [44], see also Beasley [26] for the basics).

2.3 Integrated Approaches

Domain filtering in CP is derived from feasibility considerations and removes values from domains that cannot lead to a feasible solution. *Cost based filtering* extends this idea using optimality criteria. It removes values that cannot lead to an improving solution.

2.3.1 Cost Based Filtering

Cost based filtering was introduced in a CP context by Focacci et al. [81]. The basic idea has been knowing in OR for much longer time. We mentioned already probing and reduced cost fixing in a previous section. Some LP/IP preprocessing techniques (see Andersen and Andersen [4]) follow this path as well. The novelty in the work by Focacci et al. [81] lies in the fact, that (a) it combines traditional OR ideas with CP and proves them to be superior to pure CP and pure OR techniques, and (b) it allows for using specific optimization algorithms rather then generic LP-tools and thus can offer much tighter pruning than e.g. reduced cost fixing. As defined in [81], the costs or bounds for a cost based filtering algorithm are derived from exact methods for the underlying constraint structure P.

Obviously, if obtaining a solution to P is difficult (e.g. if P is NP-hard, or if a polynomial running time algorithm for P is too slow) a fast variable fixing algorithm is desired. In the next definition we formalize the concept of using *relaxations* for cost based domain filtering. This concept was first mentioned in Fahle and Sellmann [77].

2.3.1.1 Optimization Constraints

We start with a formal concept of optimization constraints. To our knowledge, this has not been done before in literature, though optimization constraints were recently used by many authors, see e.g. Focacci et al. [81, 82], Junker et al. [125], Ottosson and Thorsteinsson [159].

Given $n \in \mathbb{N}$, let $x_1, \dots, x_n$ denote some variables with finite domains $D(x_1), \dots, D(x_n)$. Furthermore, given a constraint $C : D(x_1) \times \dots \times D(x_n) \to \mathbb{T}$, and an objective function $Z : D(x_1) \times \dots \times D(x_n) \to \mathbb{Q}$.

Definition 5 *Let $B \in \mathbb{Q}$ denote an upper/lower bound on the objective Z to be optimized.*

(i) $\vartheta_{C,Z}[B] : D(x_1) \times \dots \times D(x_n) \to \mathbb{T}$ *is called* minimization constraint *if*

$$\forall i = 1, \dots, n \quad \forall v_i \in D(x_i) :$$

$$\vartheta_{C,Z}[B](v_1, \dots, v_n) = \text{true} \iff C(v_1, \dots, v_n) = \text{true} \quad \textit{and} \quad Z(v_1, \dots, v_n) < B$$

(ii) $\vartheta_{C,Z}[B] : D(x_1) \times \dots \times D(x_n) \to \mathbb{T}$ *is called* maximization constraint *if*

$$\forall i = 1, \dots, n \quad \forall v_i \in D(x_i) :$$

$$\vartheta_{C,Z}[B](v_1, \dots, v_n) = \text{true} \iff C(v_1, \dots, v_n) = \text{true} \quad \textit{and} \quad Z(v_1, \dots, v_n) > B$$

(iii) A minimization or maximization constraint is also called an optimization constraint.

The next definition couples optimization constraints and relaxations.

Definition 6 *Given an optimization constraint $\vartheta_{C,Z}[B] : D(x_1) \times \dots \times D(x_n) \to \mathbb{T}$, let $\Delta := 2^{D(x_1)} \times \dots \times 2^{D(x_n)}$, where 2^D denotes the set of all subsets of D.*

(i) We say that an optimization constraint $\vartheta_{C,Z}[B]$ is consistent, *iff for any given $1 \leq i \leq n$ and $v_i \in D(x_i)$, there exists $v_j \in D(x_j)$, $j \neq i$, such that* $\vartheta_{C,Z}[B](v_1, \dots, v_n) = \text{true}$.

(ii) Let $\vartheta_{C,Z}[B]$ *be a minimization constraint, and let* $L : \Delta \to \mathbb{Q}$ *such that for all* $M_i \subseteq D(x_i),\ 1 \leq i \leq n$*,*

$$L(M_1 \times \cdots \times M_n) \leq \min\{Z(v_1,\ldots,v_n) \mid C(v_1,\ldots,v_n) = \text{true},\ v_i \in M_i,\ 1 \leq i \leq n\},$$

where $\min\emptyset = \infty$*. We say that* $\vartheta_{C,Z}[B]$ *is* relaxed L-consistent*, iff for any given* $1 \leq i \leq n$ *and* $v_i \in D(x_i)$*,* $L(D(x_1) \times \cdots \times \{v_i\} \times \cdots \times D(x_n)) < B$*.*

(iii) Analogously, let $\vartheta_{C,Z}[B]$ *be a maximization constraint, and let* $U : \Delta \to \mathbb{Q}$ *such that for all* $M_i \subseteq D(x_i),\ i = 1 \ldots n$*,*

$$U(M_1 \times \cdots \times M_n) \geq \max\{Z(v_1,\ldots,v_n) \mid C(v_1,\ldots,v_n) = \text{true},\ v_i \in M_i,\ 1 \leq i \leq n\},$$

where $\max\emptyset = -\infty$*. We say that* $\vartheta_{C,Z}[B]$ *is* relaxed U-consistent*, iff for any given* $1 \leq i \leq n$ *and* $v_i \in D(x_i)$*,* $U(D(x_1) \times \cdots \times \{v_i\} \times \cdots \times D(x_n)) > B$*.*

When solving an optimization problem, B is used as a no-good and is usually determined as the value of the incumbent solution. As the quality of B is crucial for the effectiveness of the domain filtering algorithm, in practice a primal heuristic is often applied to determine a fairly good solution prior to the tree search.

From the definition, relaxed L-consistency (relaxed U-consistency follows analogously) can the easier be achieved the weaker L is. For $L \equiv -\infty$, any domain filtering algorithm has nothing to do, whereas the tightest "relaxation" is achieved when $L(M_1 \times \cdots \times M_n) = \min\{Z(v_1,\ldots,v_n) \mid C(v_1,\ldots,v_n) = \text{true},\ v_i \in M_i,\ 1 \leq i \leq n\}$. That is, the choice of L determines the degree of domain filtering and thus the degree of consistency. Usually, L is chosen as a tight bound that can be computed quickly.

Clearly, optimization constraints are closely related to global constraints and generalized arc-consistency (e.g. Régin [172]) as they link a (global) constraint together with the restriction to improve on the objective function. The main contribution here consists of the definition of relaxed consistency that has been widely used in the OR community before to prune in the search tree. The idea is similar to the definition of bound consistency that can also be achieved more easily than general arc consistency, and that was proved valuable for many applications.

2.3.2 CP Based Column Generation

Column generation divides a given problem into a master problem and a subproblem. Especially in real-world applications the subproblem contains a large number of non-linear constraints and it is therefore not so well suited for traditional OR algorithms. We propose to apply a CP approach instead to solve the subproblem.

Using an arbitrary CSP as the subproblem has two major advantages. Firstly, it generalizes the class of subproblems and thus allows to use column generation even if the subproblem does not reduce to a MIP. Secondly, it allows to exploit constraint satisfaction techniques to solve the subproblem. CP-based column generation is particularly well-suited for subproblems that can partially, but not entirely be solved by a polynomial OR method M. In this case, some

constraints do not fit and have to be treated separately. In the CSP approach, the optimization algorithm M, as well as the algorithms of the other constraints will be used in a uniform way, namely to reduce the domains of variables. We can also say that the CSP-approach allows different algorithms to communicate and to co-operate via domain reduction.

Compared to traditional approaches of generating columns, the CP framework combines a high expressiveness with the domain reduction capabilities of CP algorithms. In particular, the CP framework allows to encapsulate traditional generation techniques (e.g. M) inside a constraint and to exploit them for domain reduction. We can also say that additional constraints will cut illegal choices as early as possible when searching for best columns. Thus, constraint programming based column generation can be seen as a way of enhancing or extending traditional approaches to column generation.

In Chapter 4 we will apply this framework to complex airline crew rostering problems where the core constraint is a shortest path problem. Before that, we will present a knapsack constraint in Chapter 3. In many applications of column generation the subproblem is either a constraint knapsack problem or a constraint shortest path problem. Thus, we have the ingredients to model and solve a large class of real-world problems by a column generation approach.

2.3.3 CP Based Lagrangian Relaxation

We consider an integer linear optimization problem P consisting of the two constraint families $\mathcal{A}$: $\boldsymbol{Ax} \leq \boldsymbol{b}$, $\boldsymbol{x} \in \mathbb{N}_0^n$, and $\mathcal{B}$: $\boldsymbol{Bx} \leq \boldsymbol{d}$, $\boldsymbol{x} \in \mathbb{N}_0^n$:

$$\begin{array}{rl} P: \max & \boldsymbol{p}^T \boldsymbol{x} \\ \text{s.t.} & \boldsymbol{Ax} \leq \boldsymbol{b} \\ & \boldsymbol{Bx} \leq \boldsymbol{d} \\ & \boldsymbol{x} \in \mathbb{N}_0^n \end{array} \tag{2.16}$$

Further, we assume that domain filtering algorithms DF($\mathcal{A}$) and DF($\mathcal{B}$) exist for each of the two families. Then, an obvious approach to solve P exactly is to apply an LP based branch-and-bound algorithm. Linear relaxation bounds can easily be obtained by applying a standard LP solver. That bound often yields a good estimate on the objective value that can (still) be reached. However, it is not straightforward to see how this value could be exploited for efficient domain filtering. Applying conventional reduced cost propagation only indirectly exploits the structure of the problem and therefore appears ineffective, whereas to perform probing via full re-optimization can be very costly and thus inefficient.

We can do better if we use the existing domain filtering algorithms DF($\mathcal{A}$) and DF($\mathcal{B}$) to tighten the problem formulation in every choice point. Even though DF($\mathcal{A}$) and DF($\mathcal{B}$) may be efficient and effective for the substructures they have been designed for, their application for the combined problem is usually not. Due to the fact that tight bounds on the objective cannot be obtained by taking only a subset of the restrictions into account. An accurate bound on the overall problem can only be computed by looking at the entire problem, i.e., it cannot be achieved by only looking at either one constraint family alone.

Lagrangian relaxation allows us to bring together the advantages of a tight continuous global bound and the existing domain filtering algorithms that exploit the special structure of

their respective constraint families. We introduce non-negative Lagrangian multipliers $\lambda_i \geq \mathbf{0}$ and define the Lagrangian subproblem

$$\begin{aligned} L_{\mathcal{B}}(\lambda) : \max \quad & z(\lambda) := \boldsymbol{p}^T \boldsymbol{x} - \lambda^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) \\ \text{s.t.} \quad & \boldsymbol{B}\boldsymbol{x} \leq \boldsymbol{d} \\ & \boldsymbol{x} \in \mathbb{N}_0^n \end{aligned} \tag{2.17}$$

Then, the Lagrangian multiplier problem is to solve

$$\begin{aligned} \min \quad & z(\lambda) \\ \text{s.t.} \quad & \lambda \geq \mathbf{0} \end{aligned} \tag{2.18}$$

For every $\lambda \geq \mathbf{0}$, $z(\lambda)$ is a valid upper bound on the objective. Therefore, we can apply DF($\mathcal{B}$) on the constraint family $\mathcal{B}$ each time we solve the Lagrangian subproblem $L_{\mathcal{B}}(\lambda)$. After we have found optimal Lagrangian multipliers λ^*, i.e. $z(\lambda^*) \leq z(\lambda)\ \forall\ \lambda \geq \mathbf{0}$, we use the (optimal) dual information π on the constraint family $\mathcal{B}$ to perform cost based filtering with respect to substructure $\mathcal{A}$. By relaxing the second constraint family and using multipliers $\pi \geq \mathbf{0}$, we obtain:

$$\begin{aligned} L_{\mathcal{A}}(\pi) : \max \quad & \boldsymbol{p}^T \boldsymbol{x} - \pi^T(\boldsymbol{B}\boldsymbol{x} - \boldsymbol{d}) \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{N}_0^n \end{aligned} \tag{2.19}$$

Analogously, we can apply DF($\mathcal{A}$) now.

To summarize: two linear optimization constraint families $\mathcal{A}$ and $\mathcal{B}$ for which efficient domain filtering algorithms DF($\mathcal{A}$) and DF($\mathcal{B}$) are known can be combined effectively by computing Lagrangian multipliers for $\mathcal{A}$, using DF($\mathcal{B}$) for filtering in each Lagrangian subproblem $L_{\mathcal{B}}(\lambda)$, and then handing back dual information of the optimal $L_{\mathcal{B}}(\lambda^*)$ on $\mathcal{B}$ to propagate $\mathcal{A}$ with the corresponding (optimal) reduced cost objective, i.e. we apply DF($\mathcal{A}$) on $L_{\mathcal{A}}(\pi)$. This procedure even strengthens the bound on the objective, as domain reduction is also performed during bound computation. However, one problem arises: If domains are reduced during the process of finding optimal Lagrangian multipliers, the algorithm that solves the Lagrangian dual must be aware of this. It is subject of further research, how e.g. subgradient methods must be adapted to be able to cope with that situation.

Later in this thesis we will apply CP based Lagrangian relaxation to a multimedia application. Before, we present some extensions and generalizations of the concept presented above.

2.3.3.1 Coupling more than Two Optimization Constraints

The procedure sketched can easily be generalized if the coupling of more than two constraints is desired. All we need to do is to select the substructure that determines the Lagrangian subproblem, i.e., the one that has to be solved several times with changing objective functions. After we have computed (nearly) optimal Lagrangian multipliers, we apply the domain filtering algorithm for the other substructures with a modified objective function. That modification is determined by the dual values of the family of constraints in the Lagrangian subproblem and the Lagrangian multipliers for the remaining substructures.

2.3.3.2 Linear Relaxations

If continuous bounds are preferred to bounds based on Lagrangian relaxations, it is also possible to use dual values instead of Lagrangian multipliers to modify the objective functions for the respective subproblems we want to apply a domain filtering algorithm to. We still use the terminology of a coupling method based on Lagrangian relaxation, as we use Lagrangian objectives for cost based filtering.

Notice that this method can also be used in combination with tightening algorithms such as cut generators. We simply incorporate all additional cuts as a new family of constraints for which we have to find Lagrangian multipliers (or dual values). This method is known as *Relax-and-Cut* (see e.g. Guignard [102], Porto et al. [170]).

2.3.3.3 Binary IPs

Interestingly, we achieve a domain filtering algorithm for binary IPs as a special case. Given $A \in \mathbb{Q}^{m\times n}$, $b \in \mathbb{Q}^m$, and $p \in \mathbb{Q}^n$, we consider the following binary program:

$$\begin{array}{lrcl} \max & p^T x & & \\ \text{s.t.} & Ax & \leq & b \\ & x & \in & \{0,1\}^n \end{array} \tag{2.20}$$

The problem can be viewed as a combination of m knapsack problems. Assuming that we solve the continuous relaxation to compute an upper bound, let $\pi \in \mathbb{Q}^m$ and $\mu \in \mathbb{Q}^n$ denote the optimal solution to the dual problem, i.e., π and μ solve the following linear problem:

$$\begin{array}{lrcl} \min & b^T\pi + 1^T\mu & & \\ \text{s.t.} & A^T\pi + \mu & \geq & p \\ & \pi, \mu & \geq & 0 \end{array} \tag{2.21}$$

Let $0 \leq i < m$, and let ${}^iA \in \mathbb{Q}^{(m-1)\times n}$ denote the matrix that evolves from A by erasing row i, and ${}^ib$, ${}^i\pi \in \mathbb{Q}^{m-1}$ are the vectors that evolve by erasing component i from b and π, respectively. Furthermore, let a_i denote the ith row of matrix A. Then, for every $0 \leq i < m$ we perform domain reduction with respect to the following knapsack problem:

$$\begin{array}{lrcl} \max & (p^T - {}^i\pi^{T\,i}A)x + {}^i\pi^{T\,i}b & & \\ \text{s.t.} & a_i x & \leq & b_i \\ & x & \in & \{0,1\}^n \end{array} \tag{2.22}$$

Thus, as a special application of CP based Lagrangian relaxation, we achieve an effective filtering algorithm for binary IPs that runs in $\Theta(mn \log n)$ (using the domain filtering algorithm for knapsack problems presented in Chapter 3) after we have found some optimal dual values of the continuous relaxation.

A different way of using cost based filtering for binary IPs will be presented in Section 3.2.

2.4 Experimental Methodology

Several results in this thesis are supported by numerical evaluation. The underlying algorithms were coded in C++ as well as commercial software packages for CP and IP/LP

problems. Whenever applicable, we used a hardware environment best suited to the experiments. E.g. for crew rostering (Chapters 4 and 5), we used large Sun Sparc compute-servers. This kind of computer is widely used in production systems in airline companies. In the real-world set-up for nurse scheduling (Chapter 6), and the automatic recording problem (Chapters 7, 8), on the other hand, low-end architectures are in use. Hence, our experiments were performed on standard PCs.

2.4.1 Software Packages

For the numerical tests in this thesis we used ILOG SOLVER whenever CP functionality (consistency, advanced tree search) was needed.[5] And we applied ILOG CPLEX to obtain LP solution or LP-based IP solutions.

ILOG SOLVER is a commercial software package that provides data structures for variables and domains, several basic constraints, an arc-consistency method, a tree search module (e.g. DFS, LDS, DDS) and some global constraints (see [120]). SOLVER thus offers all the primitives needed to implement efficient CP approaches. Furthermore, it is based on many years of experience and "know-how" in CP. This allows the user to concentrate on the problems to be solved rather than on details of the solution technology. A drawback is the fact, that it is sometimes difficult to analyze what's going on "behind the scenes". E.g. it is not documented which arc-consistency algorithm is internally used.

ILOG CPLEX is a well-established commercial software package which is used for solving LPs and IPs. It provides simplex and interior point methods and a tree search mechanism. Furthermore, primal heuristics and cutting plane methods are integrated (see [119]). This sums up to a very efficient and fast software solution (see Bixby [34] for examples and figures). On the other hand CPLEX does not provide a full documentation of the internals which hampers the analyzation of some phenomena.

2.4.2 Benchmarks

For knapsack (Chapter 3) or maximum clique (Chapter 10) standard benchmarks exist and are widely used. In the area of airline crew scheduling (Chapters 4, 5) there is no general benchmark set. Huge variations in the requirements of different airline companies are the one, confidentiality the other reason for this situation. Fortunately, we were provided with real-world data by our industrial partners.

The home health care problem (Chapter 6), as well as the automatic recording problem (Chapters 7, 8) were barely considered before and no established benchmark data is available. We designed some synthetic benchmarks in both cases, closely following the requirements from the respective real-world scenarios.

---

[5] The only exception is the domain filtering for the clique problem. We can prove (Lemma 6) that one pass through our constraints is sufficient to ensure consistency and therefore we did not apply any generic consistency method.

2.4.3 Measuring Runtime

Our runtime is measured as original process time. Theoretically, this provides the correct runtime, independent of any other process on the machine. In practice, two problems remain and should be mentioned here:

(i) Solaris as well as Linux systems provide a clock resolution of $\frac{1}{100}$ sec for process time. Thus, we cannot safely distinguish two runtimes t_1, t_2 that differ in less than $\frac{2}{100}$ sec. For the same reason, short run times may differ by large relative gaps. For long running processes, the relative deviation in runtime can be neglected.

(ii) Other processes still indirectly affect our runs, since each task-switch invalidates the caches. Thus, the more processes compete for the processor, the more often a program has to request data from slow memory instead of using prefetched information. Since the operating system (OS) itself starts actions at some arbitrary points in time, we cannot completely discard these effects.

To cope with situations like these a typical work around would be to repeat experiments for a certain number of times and use some appropriate measure for the mean value. Unfortunately, this was not possible here: The experiments presented in Chapter 10 were run on 9 computers and each of them took 400h (≈ 17 days) for the DIMACS benchmark and additional 2 month for the tests on random graphs. Evidently, a reasonable number of repetition would require to bind the same computers for more than a year.[6] The experiments in Chapter 7 used accumulated runtimes in the same order of magnitude.

On the other hand, we are only focusing some small deviations. Furthermore, these deviations seem to be bounded: We never produced deviations between best and worst runtime of more than 1.5% if some benchmark was running for more than 60 sec. Therefore, when interpreting experimental data, we take into account that runtime may be up to 1.5% faster than the time measured, and rank results accordingly. Regarding *(i)* we also consider a $\frac{1}{100}$sec difference. That is, in order to compare two times t_1, t_2, we say "t_1 is better than t_2" ("$t_1 < t_2$"), or "t_1 is equal to t_2" ("$t_1 = t_2$") if

$$\begin{aligned} \text{``}t_1 < t_2\text{''} \quad &:\Longleftrightarrow \quad t_1 + \tfrac{1}{100} < \tfrac{t_2 - \frac{1}{100}}{1.015} \\ \text{``}t_1 = t_2\text{''} \quad &:\Longleftrightarrow \quad \left[\tfrac{t_1 - \frac{1}{100}}{1.015}, t_1 + \tfrac{1}{100}\right] \cap \left[\tfrac{t_2 - \frac{1}{100}}{1.015}, t_2 + \tfrac{1}{100}\right] \neq \emptyset \end{aligned} \tag{2.23}$$

The number of iterations, or branch-and-bound nodes is treated analogously when being analyzed.

[6] It is more than likely that we will *not* get identical results for identical runs over a time period of more than a year since kernel updates and changes in the infrastructure will virtually produce "different" computers.

3

Cost Based Filtering for Knapsack Constraints

An effective way of combining the advantages of Constraint Programming (CP) and Operations Research (OR) techniques is the development of optimization constraints that perform cost based filtering (Focacci et al. [81]). Optimization constraints are used for pruning and to include (exclude) items that must (cannot) be part of any improving solution. We introduce propagation algorithms to perform pruning and cost based filtering for Constrained Knapsack Problems (CKPs).

In every tree search, there is a trade-off between the quality of the bounds (i.e. the time saved due to effective pruning) and the time needed for their computation. When solving pure KPs, a big effort to tighten the problem in every search node does not usually pay off. However, in the presence of additional constraints that have to be propagated in addition to the optimization constraint, the total cost per choice point is usually much bigger. Thus, the gain due to effective bounding and tightening is higher, and better bounds can be used profitably for pruning and domain reduction. On the other hand, fast KP reduction algorithms using weak bounds, such as the algorithm developed by Dembo and Hammer [62], are not effective enough for more complex CKPs.

Based on reduction techniques for KP, we develop propagation routines for knapsack constraints. We present several new propagation algorithms using bounds of different quality. The method considered the most interesting — both theoretically and practically — is based on a bound proposed by Martello and Toth [142]. By reusing information gained in an initial preprocessing step taking time $\Theta(n \log n)$, the actual reduction per choice point only requires linear time. We experimentally compare two of the new methods with two other reduction algorithms which have been proposed earlier in the KP literature.

Trick recently proposed a dynamic programming approach for propagation of knapsack constraints [199]. We compare this approach to the one presented here in section 3.3.

CP based column generation describes a generic way of how to treat arbitrary constraints for the constrained subproblem in the column generation phase. This approach has been

successfully applied to the Crew Assignment Problem, where the subproblem is a Constrained Shortest Path Problem (Fahle et al. [73], see Chapter 4).

Another important class of subproblems that evolves when following a column generation approach are (Constrained) Knapsack Problems. They evolve for example when solving a (Constrained) Cutting Stock problem. To motivate the work presented, we show exemplary how CKPs can be used when generating columns for this problem.

3.1 Constrained Knapsack Problems

The CKP is a knapsack problem with additional constraints. We do not require these additional constraints to be linear. Nevertheless, the objective function and the knapsack constraint itself have to be linear. Formally, the CKP is defined as follows:

Definition 7 *Let $C, n, w_1, \dots, w_n \in \mathbb{N}$; $p_1, \dots, p_n \in \mathbb{Z}$. C is the capacity of the knapsack, n the number of items, and w_i the weight of item i with profit p_i $\forall\ 1 \le i \le n$. Moreover, let $w := (w_1, \dots, w_n)^T$, and $p := (p_1, \dots, p_n)^T$.*

1. *Let $\mathbb{B} := \{0, 1\}$, and $G := \{x \in \mathbb{B}^n \mid w^T x \le C\}$.*
2. *Let $k \in \mathbb{N}$, and $R := \{r_1, \dots, r_k \mid r_j : \mathbb{B}^n \to \mathbb{B} \quad \forall\ 1 \le j \le k\}$. Every $r \in R$ is called a* (knapsack) rule *and R is called a* (knapsack) rule set.
3. *Every $x \in G$ is called* feasible (with respect to a given rule set R), *iff $r(x) = 1\ \forall\ r \in R$. $F(R) := \{x \in G \mid x$ is feasible$\}$ is called the* set of feasible constrained knapsacks (with respect to rule set R). *To simplify the notation, we often write F instead of $F(R)$ if R is known from the context.*
4. *The* Constrained Knapsack Problem *is then to*

$$\textit{maximize } p^T x, \quad x \in F.$$

Notice, that for the unconstrained KP it holds, $F = G$. Here, we investigate the general case of $F \subseteq G$. Generally, algorithms for the unconstrained KP are not able to solve the CKP, because they do not allow the incorporation of additional constraints. Moreover, algorithms designed to solve pure KPs make certain assumptions that do not hold for CKPs. E.g., it is not clear with the CKP that we can require the profits to be non-negative (as is the case for KP), because the strategy to omit items with positive weight and negative profit (see Martello and Toth [145]) may not yield feasible knapsacks at all.

In the following, with identifiers $\mathbb{B}$, C, n, w, p, G, R, and F we refer to the above definition. We will sometimes need to refer to reduced CKPs where an item $i \in \{1, \dots, n\}$ is either included or excluded in any feasible solution. We refer to those problems with CKP$[x_i = 1]$ or CKP$[x_i = 0]$, respectively.

3.2 Applications for Constrained Knapsack Problems

Constrained Knapsack Problems appear in various application areas. In the following, we sketch four examples.

When applying the Constraint Based Column Generation paradigm to appropriate optimization problems, CKPs occur as subproblems. For example, when applying Column Generation to the *Constrained Cutting Stock Problem* – a Cutting Stock Problem with additional constraints on the cutting patterns – results in a CKP subproblem. Additional constraints usually stem from real-world applications (an example of real-world constraints is given by Chu and Antonio [53]) and may be non-linear.

In the context of the CP-based Column Generation, we search for feasible knapsacks with negative reduced costs. In the Constrained Cutting Stock Problem, for instance, each cutting pattern has cost 1 since we try to minimize the number of rolls needed to cover the specified demand. Thus, the objective in the subproblem is to minimize $1 - \pi^T \boldsymbol{x}$ (i.e. to minimize the reduced costs of the cutting pattern), where π is the vector of dual values corresponding to the current optimal solution of the continuous relaxation of the master problem. Our objective in the CKP then is to maximize $\pi^T \boldsymbol{x}$ with an initial lower bound of 1.

The *Quadratic Knapsack Problem (QKP)* calls for maximizing a quadratic boolean objective function subject to a linear knapsack constraint. The relax and cut algorithm of Porto et al. [170] computes bounds of the QKP by linearizing the problem to KP, then tightening the problem by adding three families of valid inequalities, and finally solving the resulting linear program (LP) by Lagrangian relaxation. Thus, a series of KPs has to be solved in every search node. The authors note that fixing variables was vital to their approach. Filtering algorithms for KP are typically used to reduce the size of the given QKP (see e.g. Caprara et al. [46]). The algorithms proposed in Sections 3.4 and 3.5 may help to increase the performance of the overall approach by Porto et al. [170].

In section 2.3.3.3 we have seen how to formulate a domain filtering algorithm for binary IPs based on Lagrangian coupling. An alternative approach is to use *surrogate relaxation* and knapsack constraints instead. As before, let $A \in \mathbb{Q}^{m \times n}$, $\boldsymbol{b} \in \mathbb{Q}^m$, and $\boldsymbol{p} \in \mathbb{Q}^n$, and we consider the following binary program:

$$\begin{array}{lrcl} \text{Maximize} & \boldsymbol{p}^T\boldsymbol{x} & & \\ \text{subject to} & A\boldsymbol{x} & \leq & \boldsymbol{b} \\ & \boldsymbol{x} & \in & \{0,1\}^n \end{array} \tag{3.1}$$

Let $\mu \in \mathbb{R}^m_{\geq 0}$ be a vector, and $\boldsymbol{x} \in \mathbb{B}^n$ be a feasible solution to (3.1). Then $\boldsymbol{x}$ is also a solution to

$$\begin{array}{lrcl} \text{Maximize} & \boldsymbol{p}^T\boldsymbol{x} & & \\ \text{subject to} & (\mu^T A)\boldsymbol{x} & \leq & (\mu^T \boldsymbol{b}) \\ & \boldsymbol{x} & \in & \{0,1\}^n \end{array} \tag{3.2}$$

Surrogate relaxation has been proposed by Glover [89]. It is known to be at least as tight as Lagrangian relaxation (see Greenberg and Pierskalla [97]) and thus as least as tight as LP bounds (Geoffrion [87]). There is no general theory on how to find optimal surrogate multipliers, though for some models corresponding methods are known (see e.g. Martello and Toth [143]). Still, computing the surrogate bound (3.2) exactly, is NP-hard in general as it remains to solve a knapsack problem. Using the methods developed in this chapter, it

is however possible to perform domain filtering based on (3.2). The resulting runtime for a given μ is $O(mn)$ for transforming (3.1) to (3.2), and $O(n \log n)$ per domain filtering. Whether to apply Lagrangian coupling or surrogate relaxation and knapsack constraints for domain filtering depends on the problem's characteristic and cannot be decided in general.

Finally, the *automatic recording problem*, that we will focus on in Chapters 7 and 8, can be solved via a combination of knapsack and shortest path or weighted maximum stable set constraints.

3.3 Constrained Knapsack vs. Pure Knapsack Problems

For pure knapsack problems without additional constraints, the state-of-the-art solving techniques would focus on a so called *core problem*, which may be extended during the optimization process (Martello et al. [141], Pisinger [168]). For these algorithms it is difficult to foresee how the reduction algorithm we present in the following could be efficiently integrated. However, in the context of this chapter we focus on Constrained Knapsack Problems, where a branch-and-bound tree search framework is needed to find feasible solutions with respect to additional constraints, and where algorithms tailored for the pure KP are likely to fail. This motivates the decision to make use of efficiency orderings of the knapsack items in an algorithm for CKP, although it is already known that the calculation of those orderings does not often pay off when solving pure KPs.

Trick [199] derives a hyper-arc-consistency approach from some dynamic programming method designed for pure KPs. The problems considered in that paper differ from the ones described in this chapter: the input is a *two sided knapsack*, i.e. a linear constraint of the form $L \leq \sum_i W_i x_i \leq C$. The running time depends heavily on C and L in practice, and is theoretically bounded in $O(nC^2)$ only. That is, the algorithm is pseudo-polynomial in C. Also, the space requirement of $O(nC)$ depends on C. The advantage of this approach is that it works independently of the type of objective function.

In contrast, the approach presented here considers *one sided knapsacks* with $\sum_i W_i x_i \leq C$, runs in amortized linear time, uses $O(n)$ space, and is tailored to linear cost functions only.

3.3.1 Variable Fixing for the Constrained Knapsack Problem

In a canonical IP formulation of the knapsack problem, there is one variable x_i for each item $i \in \{1, \ldots, n\}$. The domain of each variable is defined as $D(x_i) := \mathbb{B}$. Furthermore, the knapsack constraint is modeled by a function $\omega : \mathbb{B}^n \to \mathbb{B}$ with $\omega(x) = \omega(x_1, \ldots, x_n) = 1$ iff $w^T x \leq C$. Finally, the objective function is $P : \mathbb{B}^n \to \mathbb{Q}$ with $P(x) = P(x_1, \ldots, x_n) := p^T x$. Given any lower bound $B \geq 0$, we consider the maximization constraint $\vartheta_{\omega,P}[B]$. Items of the CKP fall into either one of the following classes:

- items i that can be excluded from further investigation as they cannot be part of any improving solution, i.e.

$$P(x) \leq B \quad \forall\, x \in \{y \in G \mid y_i = 1\} \tag{3.3}$$

- items i that can be included in the knapsack as they must be part of any improving solution, i.e.

$$P(x) \leq B \quad \forall\, x \in \{y \in G \mid y_i = 0\} \tag{3.4}$$

- items that cannot be decided at the moment.

Propagation is expected to include or remove items that do not fall into the last class. Since showing that either (3.3) or (3.4) holds for an item i (i.e. to check the satisfiability of $\vartheta_{\omega,P}[B]$) generally requires solving a KP itself, complete propagation here is an NP-hard task. Therefore, we only check if the inequality holds for an upper bound U on KP$[x_i = b]$, $b = 0$ or $b = 1$, i.e., if $U(\mathbb{B} \times \cdots \times \{b\} \times \cdots \times \mathbb{B}) \leq B$. Then, we write $U(\text{KP}[x_i = b]) \leq B$.[1]

In the KP literature, this idea has been used to reduce problem sizes initially or in selected nodes of a branch-and-bound tree. Variable fixing is of great importance especially when solving complex problems such as quadratic knapsack problems [170]. In the next section, we give the definitions of some bounds that have been developed for the KP.

3.3.2 Upper Bounds for Knapsack Problems

The effectiveness of a knapsack constraint propagation algorithm relies heavily on the quality of the bounds calculated. Following the presentation given in Chapter 2 by Martello and Toth [145], we present some upper bounds that have been originally developed for the maximization problem KP. They also apply to the CKP by relaxing it to a KP first. Obviously, ignoring all additional constraints does not often yield tight bounds on the objective. However, if the additional constraints satisfy certain properties, they can be incorporated into the objective function of a pure KP using Lagrangian relaxation. For additional linear constraints, there are ways of doing this effectively (see Focacci et al. [82], Sellmann and Fahle [185]). Notice, that dropping all additional constraints allows to set $x_i := 0\ \forall\ p_i \leq 0$ and $1 \leq i \leq n$. We therefore require all items to have positive profits.

Without loss of generality, we may assume that the items are ordered according to decreasing efficiency, i.e. $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$. We define the *critical item* s of a knapsack problem as the first item that overloads the knapsack, that is $s = \min_j\{\sum_{i=1}^{j} w_i > C\}$ (we omit the trivial case here where no such s exists). Dantzig [57] showed that the linear relaxation of the 0-1 knapsack has the optimal value $\sum_{j=1}^{s-1} p_j + \bar{c}\frac{p_s}{w_s}$, where $\bar{c}$ is defined as the remaining capacity of the knapsack after filling in the first $s-1$ items: $\bar{c} = C - \sum_{j=1}^{s-1} w_j$.

Let $\emptyset \neq M_1, \ldots, M_n \subseteq \mathbb{B}$. Denote with $l_i := \min(M_i)$ the minimum, and with $r_i := \max(M_i)$ the maximum of M_i, $1 \leq i \leq n$. The first upper bound on KP is defined as $U_1 : 2^{\mathbb{B}^n} \to \mathbb{Q}$ with

$$U_1(M_1 \times \cdots \times M_n) := \max\{P(x_1, \ldots, x_n) \mid \omega(x_1, \ldots, x_n) = 1,\ x_i \in [l_i, r_i],\ 1 \leq i \leq n)\}.$$

It holds,

$$U_1 := U_1(KP) = \sum_{j=1}^{s-1} p_j + \left\lfloor \bar{c}\frac{p_s}{w_s} \right\rfloor . \tag{3.5}$$

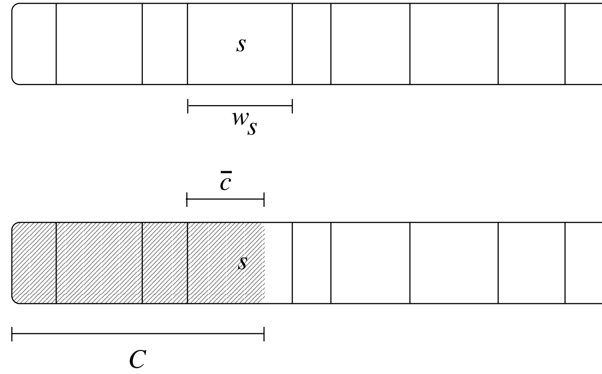


Figure 3.1: *The width of each element is proportional to its weight. The elements are ordered with respect to the efficiencies p_i/w_i. The leftmost element has the biggest efficiency, and the rightmost the smallest one. s marks the critical item in U_1.*

A second bound U_2 was introduced by Martello and Toth [142]. It imposes the integrality of the critical item s. Item s either belongs to the optimal solution (leading to a value U^1) or not (leading to a value U^0):

$$U^0 = \sum_{j=1}^{s-1} p_j + \left\lfloor \bar{c}\frac{p_{s+1}}{w_{s+1}} \right\rfloor. \tag{3.6}$$

$$U^1 = \sum_{j=1}^{s-1} p_j + \left\lfloor p_s - (w_s - \bar{c})\frac{p_{s-1}}{w_{s-1}} \right\rfloor. \tag{3.7}$$

Defining U_2 as the maximum of U^0 and U^1 results in a bound dominating U_1. Formally, let $\emptyset \neq M_1, \ldots, M_n \subseteq \mathbb{B}$, and denote with s the critical item with respect to necessarily included and excluded items implicitly defined by the M_i. We set $U_2 : 2^{\mathbb{B}^n} \to \mathbb{Q}$ with $U_2(\emptyset) := -\infty$, and

$$U_2(M_1 \times \cdots \times M_n) := \max(U^0, U^1) - \textstyle\sum_{i<s, M_i=\{0\}} p_i + \sum_{i>s, M_i=\{1\}} p_i.$$

It holds,

$$U_2 := U_2(KP) = \max(U^0, U^1) \leq U_1. \tag{3.8}$$

Instead of estimating the loss caused by the integrality of item s by the efficiency of the neighboring items of s, an even tighter bound can be obtained by calculating bounds U_1 on KP$[x_s = 0]$, and KP$[x_s = 1]$ (Fayard and Plateau [78], Hudson [113], Villela and Bornstein [205]). Let $\overline{U}^0 := U_1(\text{KP}[x_s = 0])$, and $\overline{U}^1 := U_1(\text{KP}[x_s = 1])$. Then, $U_3 := \max(\overline{U}^0, \overline{U}^1)$ dominates U_1 and U_2. An even tighter bound could be obtained by using U_2 instead of U_1 in the definition of $\overline{U}^0$ and $\overline{U}^1$ and so on.

Fig. 3.1 and 3.2 give graphical interpretations of the bounds U_1 and U_3. Obviously, all three bounds U_1, U_2, U_3 can be computed in time $O(n)$ after a preprocessing step of sorting the items according to decreasing efficiencies. This requires time $\Theta(n \log n)$. Balas and Zemel [17] developed an algorithm for the calculation of s using linear time without any

[1] To improve the readability, here and in the following we write CKP or KP instead of $\mathbb{B}^n$, and identify CKP$[x_i = b]$ as well as KP$[x_i = b]$ with $\mathbb{B} \times \cdots \times \{b\} \times \cdots \times \mathbb{B}$, where $\{b\}$ is the ith factor.

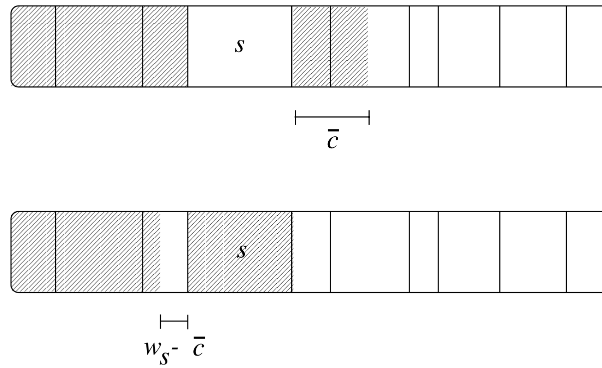


Figure 3.2: *U_3 requires $s \in \{0, 1\}$. The figures show $U_1(KP[x_s = 0])$, and $U_1(KP[x_s = 1])$.*

preprocessing. But for the reduction algorithm that we present in the following – just as in former reduction algorithms for the KP – the efficiency ordering is needed anyway. In addition, we use an ordering of the items with respect to increasing weights.

In a tree search, both orderings can be calculated in an initial preprocessing step. After that, they can be reused in every search node. Within a column generation context, the weight ordering only has to be calculated once, but the efficiency ordering has to be recomputed every time new dual values of the master problem lead to a change of the objective in the successive CKPs.

3.3.3 Reduction Techniques for Knapsack Problems

A first reduction algorithm for KPs based on upper bound U_1 has been proposed by Ingargiola and Korsh [121]. In a loop over all items $i = 1, \ldots, n$, the algorithm determines $U_1(KP[x_i = b]),\ b \in \{0, 1\}$. Since each bound calculation takes linear time, the worst case complexity of this algorithm is $\Theta(n^2)$.

If bound U_2 is used instead of U_1, more effective pruning can be achieved in the same asymptotic running time. Martello and Toth [144], showed that the running time can be reduced to $O(n \log n)$ while keeping the solution quality of bound U_2. The key idea of their algorithm is to compute the critical item s by binary search. We refer to the methods of Ingargiola and Korsh, and Martello and Toth as IKR, and MTR, respectively.

Dembo and Hammer [62] proposed a reduction algorithm (DHR) that runs in linear time $\Theta(n)$. They calculate the critical item s only once for the original problem. Within a loop they estimate the loss when removing/including item $i = 1, \ldots, n$ by extrapolating the efficiency of item s, which allows them to perform this step in constant time. As this extrapolation is less accurate than U_1, their method is not as effective as IKR or MTR.

Though they were developed a decade or more ago, DHR or MTR are still vital ingredients in state-of-the-art solvers for pure KP or quadratic KP (see e.g. Pisinger [168],[169], Porto et al. [170]).

The algorithm we present in the following does not improve on the running time of reduction techniques based on the more efficient bounds U_1, U_2, if the reduction algorithm is only called once. For such an application, the new method presented and the one developed by Martello and Toth both use the same asymptotic time $\Theta(n \log n)$.

However, the situation changes if a reduction method is called many times for similar knapsack instances, as is the case when applying a tree search. Within a tree search, we try to prune the search or to apply domain filtering after every branching step. The subsequent instances only differ with respect to the sets of variables that have already been fixed. As we will see in the next section, such a situation allows to hide parts of the work in a preprocessing step that takes time $\Theta(n \log n)$. When provided with the information gathered in this preprocessing, every call to the reduction routine requires linear time only.

3.4 A Fast Filtering Algorithm based on Bound U_1 and U_2

We will now show how to reduce the running time of IKR and MTR to $\Theta(n)$ by making use of information generated in a preprocessing step requiring time $\Theta(n \log n)$. The bounds obtained are of the same quality as in the original algorithms. Again, let KP$[x_j = b]$ denote $\mathbb{B} \times \cdots \times \{b\} \times \cdots \times \mathbb{B}$, $b \in \{0,1\}$, and $s(M_1 \times \cdots \times M_n) = \min_j\{\sum_{i \leq j \mid M_i = \mathbb{B}} w_i > C - \sum_{i \mid M_i = \{1\}} w_i\}$ denote the critical item of KP$[x_j = b]$. The key idea of the routine is to calculate the bounds of the reduced problems $U(\text{KP}[x_j = b])$ in an order of increasing weight of the items j. Thereby, we obtain a sequence of critical items that are monotonically increasing. Thus, the critical item and the upper bound for the jth item (with respect to the weight ordering) can be transformed into the critical item and upper bound for the $(j+1)$th item by starting the calculation of $s(\text{KP}[x_{j+1} = b])$ at $s(\text{KP}[x_j = b])$.

The time consuming step in reduction algorithms using bound U_1, U_2, resp., is to determine the critical items $s(\text{KP}[x_i = b]) \ \forall\ 1 \leq i \leq n$, and $b \in \{0,1\}$. Once these values are known, the calculation of the upper bounds and the reduction itself only requires linear time. (In fact, in the following algorithm the bounds can be computed at the same time as the critical items. To clarify the argumentation, however, we just show how to calculate the latter.) By omitting the fractional parts, it is also possible to calculate lower bounds for the KP. Reduction should only take place, after all lower bounds have been calculated [144]. For the CKP, however, the necessary checking of feasibility with respect to additional constraints makes the generation of lower bounds more complicated.

Although calculating $s(\text{KP}[x_i = b])$ for each single $i \in \{1,\ldots,n\}$, $b \in \{0,1\}$, generally takes linear time, the calculation of *all* these values also only requires time $\Theta(n)$ once we know an ordering $\sigma = (\sigma_1,\ldots,\sigma_n)$ of the items according to their weight, i.e. $w_{\sigma_i} \leq w_{\sigma_j}$ iff $i \leq j$. The efficiency ordering of the items as well as the the permutation σ can be obtained in a sorting step prior to any reduction and requiring time $\Theta(n \log n)$.

Given $s = s(\text{KP})$, we know that $U(\text{KP}[x_i = 1]) = U(\text{KP}) \ \forall\ i < s$, and $U(\text{KP}[x_i = 0]) = U(\text{KP}) \ \forall\ i > s$. Thus, we only need to calculate the arrays $S^1 := [s(\text{KP}[x_i = 1]) \mid i \geq s]$, and $S^0 := [s(\text{KP}[x_i = 0]) \mid i \leq s]$. We describe how to determine S^0 in the following. The calculation of S^1 is done analogously.

We iterate over all items $i < s$ in increasing order of weight. In doing so, we can be sure that $s(\text{KP}[x_i = 0])$ increases monotonically with growing $i \in \{0,\ldots,s-1\}$. Thus, we can start the search for the next critical item at the position of the previous one.

The following book keeping argument shows that this procedure only takes linear time. We estimate the computational effort of the reduction algorithm by assigning a unit cost (say, 1 €) to the items causing it:

- Every item $j \geq s$ that is being passed is charged 1 €. By "passed" we mean, that the item is being included entirely when iterating from one critical item to the other.
- Every item is charged 1 € each time it is included fractionally.

The first group of items causes at most n € costs as the critical items are monotonically increasing: every item is being passed once at most. It remains to estimate the effort for all items that are being included fractionally. Obviously, there are as many fractionally included items as critical items at most. Therefore, this group of items also costs not more than n €. Thus, the costs for the entire computation are in $O(n)$.

Finally, the calculation of $s(\text{KP}[x_s = 0])$ can be performed in a time that is linear in the number of items also. Another possibility of calculating this value is to insert item s at the position corresponding to $\overline{c}$ in the weight ordering of items and to calculate $s(\text{KP}[x_s = 0])$ just like the critical items for the exclusion of the other items.

Obviously, the above algorithm can be applied with bounds U_1 and U_2. As a consequence, we have shown the following

Theorem 1 *After a $\Theta(n \log n)$ preprocessing step,* relaxed U_2-consistency *for a knapsack constraint can be obtained in time $O(n)$ per choice point.*

It is easy to see that, for a constant number of choice points, MTR and the algorithm given above need the same running time of $\Theta(n \log n)$. If $\Omega(\log n)$ choice points have to be investigated, however, the time spent in preprocessing is dominated by the accumulated time needed in the choice points. In that case, Theorem 1 implies

Corollary 1 *If propagation is triggered in $\Omega(\log n)$ search nodes,* relaxed U_2-consistency *for a knapsack constraint can be obtained in amortized time $O(n)$ per choice point.*

Thus, in a typical CP search tree with $\Omega(\log n)$ search nodes, the method presented here is asymptotically optimal and superior to the algorithms proposed previously.

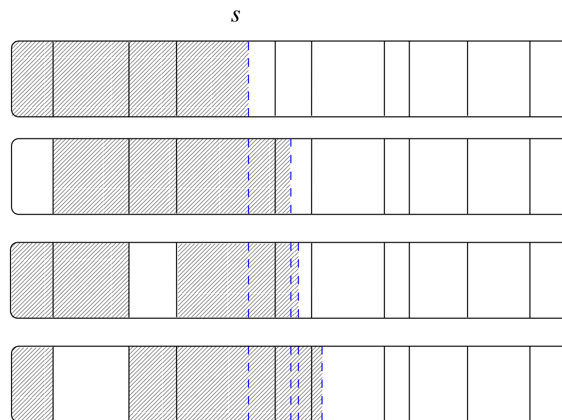


Figure 3.3: *The figure illustrates the proceeding of the reduction algorithm presented for KP$[x_i = 0]$. The weight ordering in which the items are tested ensures that the critical item moves monotonically to the right.*

3.5 More Effective Cost Based Filtering using Bound U_3

To strengthen the propagation abilities of the optimization constraint, we can use the stronger bound U_3:

$U_3(KP)$ is obtained by calculating bound U_1 on KP$[x_s = 0]$, and KP$[x_s = 1]$. For propagation based on that bound, we need to compute s_i^b, $b \in \{0,1\}$, the critical items of those restricted KPs for which $x_i = b$: Let $1 \le i \le n$, $b \in \{0,1\}$. Then, $s_i^0 := s(\text{KP}[x_i = b, x_{s(\text{KP}[x_i=b])} = 0])$, and $s_i^1 := s(\text{KP}[x_i = b, x_{s(\text{KP}[x_i=b])} = 1])$.

To do so efficiently, we first determine the values $s(\text{KP}[x_i = b])$ using the algorithm in Section 3.4. Then, we apply a binary search to determine s_i^0 and s_i^1 for all $1 \le i \le n$. This leads to a running time of $\Theta(n \log n)$. A similar idea has been introduced by Martello and Toth [144].

Corollary 2 *Relaxed U_3-consistency for a knapsack constraint can be obtained in $O(n \log n)$ time per choice point.*

For real-world instances, using a binary search to determine the critical item of KP$[x_s = b_2, x_i = b_1]$ for $b_1, b_2 \in \{0,1\}$, usually does not pay off as it is likely to be "close" to s. Thus, we consider this result to be of theoretical interest only. However, the algorithm above leads to another propagation algorithm that is asymptotically as efficient as the one presented in Section 3.4 (that runs in amortized linear time), but is even more effective. In fact, the bound it uses to perform cost based filtering is at least as good as U_2, but for some items it is even U_3:

Let $1 \le i \le n$, $b \in \{0,1\}$, $s := s(KP)$, $s_i^0 := s(\text{KP}[x_i = b, x_s = 0])$, and $s_i^1 := s(\text{KP}[x_i = b, x_s = 1])$. In contrast to the sequence of critical items that is computed for U_3, the second variable x_s that is fixed remains the same for all s_i^0, and s_i^1. Again by using the algorithm in Section 3.4, we determine $U_2(\text{KP}[x_s = 0, x_i = b]) \ \forall\ 1 \le i \le n$, and then $U_2(\text{KP}[x_s = 1, x_i = b]) \ \forall\ 1 \le i \le n$. For any given $1 \le i \le n$, we check whether $\max\{U_2(\text{KP}[x_s = 0, x_i = b]), U_2(\text{KP}[x_s = 1, x_i = b])\} \le B$. If so, we fix the value of x_i to $1 - b$.

It is easy to see that the bound calculated is at least as good as U_2. For items $i < s$ with $s(\text{KP}[x_i = 0]) = s$ and items $i > s$ with $s(\text{KP}[x_i = 1]) = s$, however, propagation is even as effective as for bound U_3. Hence, we achieve an amortized linear time algorithm based on a 'mix' of U_2 and U_3 bounds.

3.6 Cost Based Filtering for Special Constrained Knapsack Problems

Before we evaluate the propagation algorithms empirically, we would like to discuss their applicability to two special variants of the (constrained) knapsack problem that have been introduced in the literature.

3.6.1 Multi-Dimensional Knapsack Problems

The multi-dimensional knapsack problem consists of the maximization of a given profit function with respect to two or more given capacity constraints. The problem can be viewed as a collection of m pure knapsack problems sharing one objective:

$$\begin{array}{lll} \max & \sum_j p_j x_j & \\ s.t. & \sum_j w_{i,j} x_j \leq C_i, & i = 1, \ldots, m \\ & x_j \in \{0,1\} & \end{array} \tag{3.9}$$

Thus, for each of the capacity constraints we can define an optimization constraint and perform cost based filtering using the propagation algorithms we have just presented. This approach, however, suffers a setback from the fact, that the bounds computed in each optimization constraint ignore all constraints except one. Therefore, the bounds are not tight, and filtering is less effective than it could and should be.

In Sellmann and Fahle [185], we developed a generic method for the coupling of linear optimization constraints to one global optimization constraint, the *CP-based Lagrangian Relaxation*. When applied to multi-dimensional knapsack problems, the filtering algorithm for the composite constraint uses the propagation routines of the individual knapsack constraints incorporating the other constraints in a Lagrangian objective. We have shown that this approach is clearly favorable compared to the loose connection of optimization constraints that interact by domain reduction only.

Note, however, that the asymptotic complexity improvements we introduce in this chapter are lost when applying the knapsack filtering algorithm in the context of CP-based Lagrangian relaxation, because for each Lagrangian subproblem the objective changes. Thus, the efficiency ordering has to be recomputed which then dominates the algorithmic complexity. It should be mentioned that this problem does not occur when the filtering algorithms presented here are applied to column generation subproblems (as in CP-based column generation), because the objective remains fixed for the entire tree search that is applied to compute a new column. Thus, the efficiency ordering of the knapsack items has to be recomputed only when a new subproblem is set up.

3.6.2 Bounded Knapsack Problems

Bounded knapsack problems generalize the 0-1 KP by defining individual bounds on the solution vector:

$$\begin{array}{lll} \max & \sum_j p_j x_j & \\ s.t. & \sum_j w_j x_j \leq C & \\ & x_j \in \{0,1,2,\ldots,u_j\} & \end{array} \tag{3.10}$$

Obviously, (3.10) can be transformed into a CKP by replacing one original x_j by u_j new variables $x'_{j,k} \in \{0,1\}, k = 1,\ldots,u_j$. (Notice, that a finite u_j always exists, as $x_j \leq \lfloor \frac{C}{w_j} \rfloor$.). Then the algorithms presented before could be applied. This approach, however, artificially enlarges the number of variables and completely ignores the additional structure of (3.10).

We can do better by extending U_1 and U_2 to general integer bounds for KP. That is, we chose the critical item as $s := \min_j \{\sum_{i=1}^{j} u_i \cdot p_i > C\}$. Then U_1 can be re-written as

$U_1(KP) = \sum_{j=1}^{s-1} u_j \cdot p_j + \left\lfloor \overline{c}\frac{p_s}{w_s} \right\rfloor$, where $\overline{c} = C - \sum_{j=1}^{s-1} u_j \cdot w_j$. For a detailed discussion of such generalizations, and an extension of U_2, we refer to Martello and Toth [145, pp. 84ff.]. Taking these extended bounds, efficient propagation for the bounded knapsack problem is then easily achieved by the algorithms proposed in Sec. 3.4, 3.5.

3.7 Numerical Evaluation

After we theoretically analyzed the new algorithm in Section 3.4, we now compare it numerically to different methods that were derived from KP reduction techniques presented in the literature. All experiments were run on a Sun Enterprise 450 Model 4300 (296 MHz) with 1 GB RAM, under Solaris 2.6. The reduction algorithms were implemented in C++ on top of Ilog Solver 5.0 [117].

3.7.1 Test Environment

To show the potential of the new propagation techniques, and to avoid cross talking with other constraints, we decided to base the experiments on pure knapsack problems only. In doing so, we get a clear view of the performance of each filtering algorithm without disturbing interferences that can evoke easily when using more complex settings incorporating additional constraints. (see also Chapter 7). Accordingly, we also omitted specially tailored tree search or branching strategies for pure KPs. Instead, we used the default settings of the underlying CP library.

A word of warning is necessary here: even though our experiments are based on pure KP data, the filtering algorithms we developed are not suited for state-of-the-art KP solvers. Also, we do not claim that the solvers we implemented are in competition with the best KP solvers (see Section 3.3). Our focus here is clearly on constrained knapsack problems.

A weak propagation algorithm, if started from scratch, will obviously need more choice points to find an optimal or near optimal solution of the problem than a strong one. Therefore, to make the comparison fair, we initialize the lower bound with the optimal objective value $B \in \mathbb{Q}$ and just measure the time and the number of choice points that each approach requires to prove optimality.

The generator code of David Pisinger [168] was used to produce random instances of two different classes of knapsack problems where the weights w_j are randomly distributed in [1,1000], and the profits p_j are chosen as given below:

- *uncorrelated:* p_j randomly distributed in [1,1000],
- *weakly correlated:* p_j randomly distributed in $[w_j - 100, w_j + 100] \cap [1, 1100]$

In all cases, the knapsack capacity is chosen as $C = \frac{1}{2}\sum_{j=1}^{n} w_j$. The problem sizes range from 10 to 20 000 items, and 100 knapsack problems were generated for each size and class.

We omit the classes of *strongly correlated* data ($p_j = w_j + 10$) and *subset-sum* data ($p_j = w_j$). It is known that the bounds described in Section 3.3.2 are not suited to these classes (which is easy to see as $\forall\ k: \ p_k/w_k \approx 1$). For them, bounds based on cardinality constraints have shown to be effective(Martello et al. [141], Martello and Toth [146]).[2]. In the application

[2] Cardinality constraints are a restricted form of cover inequalities which we will briefly discuss in section 8.1.1

Name	see	Bound	preproc time	time per node
DHR	Sect. 3.3.3,	$D/H - bound$	–	$\Theta(n)$
MTR	Sect. 3.3.3,	U_2	$\Theta(n \log n)$	$\Theta(n \log n)$
linU_1	Sect. 3.4	U_1	$\Theta(n \log n)$	$\Theta(n)$
linU_2	Sect. 3.4	U_2	$\Theta(n \log n)$	$\Theta(n)$

Table 3.1: *Characteristics of the four algorithms used in the experiments.*

area that we focus on (see Sec. 3.2), however, it is justified to assume that the evolving KPs are more likely to fall into one of the classes that we used for our tests.

3.7.2 The Opponents

The algorithms referred to as linU_1 and linU_2 are based on the amortized linear time reduction method described in Section 3.4, and they use bounds U_1 and U_2, respectively. Methods DHR, and MTR have been described already in Section 3.3.3. We implemented all algorithms in the same CP environment. Table 3.1 summarizes the major characteristics of the candidates used in the experiments. All methods need $O(n)$ memory for the propagation stack and for the different orderings used. Only $O(1)$ additional memory is required within a choice point.

Notice, that in our experiments we do not evaluate the filtering algorithm based on a mixture of bound U_2 and U_3 which was sketched in Section 3.5. The propagation algorithm based on this mixed bound visits only slightly fewer choice points than linU_2, but therefore requires a much higher computation time. Remember from Section 3.4 that the amount of work required to perform propagation using bound U_2 is almost the same as using bound U_1. But when using the mixed bound, the workload is twice as large as that for bound U_1.

As we will show in this Section, we are facing a trade-off between the time needed per choice point and the reduction in choice points that can be achieved by using tighter bounds. Within the test environment that we have chosen for our experiments, a slight reduction in choice points does not justify a much higher effort undertaken in each choice point. Therefore, the propagation algorithm based on the mixed bound is only of interest in the context of a more complex CKP incorporating additional and possibly hard side constraints that would make even small reductions in the number of choice points more favorable. However, the algorithm developed in Section 3.5 is not competitive in the KP setting we use here where we try to avoid cross talking with additional constraints and to evaluate the pure performance of the different propagation algorithms.

3.7.3 Experimental Results

The simple approach for solving a CKP in a CP context would be to introduce a sum-constraint (i.e. $\sum_j w_j x_j \leq C$) plus a constraint stating that we are only looking for improving solutions (i.e. $\sum_j p_j x_j > B$). However, as shown in Table 3.2, this approach cannot compete with the other propagation methods at all. Both the number of choice points and the CPU time grow exponentially as the problem size increases. A dash means that the average calculation for a test instance takes more than two hours. Only small problems with not more than 40 items can be solved within that time limit for both classes. The poor performance of the pure CP

Size	uncorrelated		weakly correlated	
n	cp	$time$	cp	$time$
10	37.77	0.01	73.74	0.01
20	1 455.80	0.16	28 736.07	2.91
30	141 338.82	15.50	16 771 406.92	1641.94
40	10 311 820.44	1410.07	—	—

Table 3.2: *The pure CP approach for both problem classes.* cp *is the average number of choice points,* $time$ *the average time in seconds for 100 instances of the given size.*

Size	DHR		linU_1		linU_2		MTR	
n	cp	$time$	cp	$time$	cp	$time$	cp	$time$
10	2.43	0.00	0.87	0.00	0.67	0.00	0.67	0.00
20	5.47	0.00	2.68	0.00	2.35	0.00	2.35	0.00
40	7.20	0.00	3.61	0.00	3.22	0.00	3.22	0.00
60	10.18	0.00	6.07	0.00	5.26	0.00	5.26	0.00
80	13.96	0.01	8.43	0.00	7.04	0.00	7.04	0.00
100	14.21	0.01	8.20	0.00	6.75	0.00	6.75	0.00
200	24.85	0.02	17.16	0.02	14.47	0.01	14.47	0.01
300	32.47	0.04	22.57	0.03	18.76	0.02	18.76	0.02
400	38.19	0.05	27.69	0.04	23.28	0.04	23.28	0.04
500	46.50	0.08	33.64	0.06	28.68	0.05	28.68	0.05
600	63.61	0.11	48.67	0.09	40.95	0.08	40.95	0.08
700	54.67	0.11	41.16	0.09	34.53	0.08	34.53	0.08
800	69.92	0.16	51.76	0.13	42.38	0.11	42.38	0.11
900	68.89	0.17	51.76	0.14	42.35	0.13	42.35	0.12
1000	97.83	0.26	72.38	0.21	59.73	0.17	59.73	0.18

Table 3.3: *Uncorrelated data instances. We give the average numbers for 100 test sets per size.* $time$ *is the time in seconds,* cp *the number of choice points.*

approach highlights the need for sophisticated filtering techniques when knapsack constraints occur in a CP model. As will be shown in the following, more elaborate techniques are able to tackle problems of several 1000 items in a few seconds, generating only relatively few choice points.

3.7.3.1 Small Instances

Tables 3.3 and 3.4 show the average results of 100 different instances of the same data size n. We present the running time in seconds, and the number of choice points cp which the method visits. Table 3.5 shows a comparison of the different methods regarding the time per choice point for uncorrelated and weakly correlated data.

The Dembo/Hammer based propagation method needs to visit the largest amount of choice points among the four propagation algorithms tested. This matches the expected behavior

Size	DHR		$\text{lin}U_1$		$\text{lin}U_2$		MTR	
n	cp	$time$	cp	$time$	cp	$time$	cp	$time$
10	10.42	0.00	6.31	0.00	5.42	0.00	5.42	0.00
20	20.41	0.00	13.82	0.00	11.35	0.00	11.35	0.00
40	33.26	0.01	23.42	0.01	19.87	0.01	19.87	0.00
60	37.69	0.01	26.69	0.01	22.52	0.01	22.52	0.01
80	56.07	0.02	40.10	0.01	33.21	0.01	33.21	0.01
100	61.60	0.02	45.49	0.02	37.94	0.02	37.94	0.02
200	103.85	0.06	77.05	0.05	64.33	0.05	64.33	0.04
300	162.20	0.13	123.11	0.11	99.67	0.10	99.67	0.09
400	202.23	0.21	151.50	0.17	118.71	0.15	118.71	0.14
500	226.36	0.29	161.80	0.23	122.57	0.19	122.57	0.18
600	286.40	0.42	207.56	0.33	158.92	0.27	158.92	0.26
700	345.28	0.58	252.25	0.45	185.42	0.36	185.42	0.35
800	314.00	0.61	214.64	0.44	151.34	0.34	151.34	0.33
900	428.16	0.89	300.34	0.67	210.06	0.51	210.06	0.49
1000	451.74	1.04	313.50	0.78	220.33	0.60	220.33	0.57

Table 3.4: *Weakly correlated data instances. We give the average numbers for 100 test sets per size.* $time$ *is the time in seconds,* cp *the number of choice points.*

of a method that prunes with respect to weaker bounds. Due to the short time per choice point, though, it is only slightly slower than the other methods on uncorrelated data. Thus, the results obtained reflect the expected trade-off between an effective domain filtering and the time needed for it. In the presence of additional constraints (causing higher times spent per choice point than is needed for propagation), it is likely that a smaller number of choice points will result in a faster overall computation. $\text{lin}U_1$ uses fewer choice points than DHR, but is not as effective as the U_2 based algorithms, MTR and $\text{lin}U_2$. For the big instances, these two algorithms only visit between 50% and 65.6% of the choice points needed by DHR.

For weakly correlated data, $\text{lin}U_2$ only visits 69.7% of the choice points of the DHR routine at most. Moreover, $\text{lin}U_2$ slightly outperforms DHR with respect to the total running time. Notice that the time spent per choice point by $\text{lin}U_2$ for weakly correlated instances is less than for uncorrelated data. The reason for this is, that the preprocessing time taken for initializing the more complex data structures for $\text{lin}U_2$, and for sorting the items according to weight and efficiency is spread over a much higher amount of choice points.

3.7.3.2 Big Instances

To get a clearer insight into the characteristics of the different algorithms, we performed some tests on bigger instances. Going up to 10 000 items, the disadvantages of the poor bounds used by $\text{lin}U_1$ and especially DHR become obvious. Due to a much larger amount of choice points, the total running times exceed those of $\text{lin}U_2$ and MTR (see Table 3.6).

Still, MTR and $\text{lin}U_2$ need on average about the same time. We assume that for smaller test instances, the binary search performed by MTR is faster than the more complex book keeping of $\text{lin}U_2$. As the problem size increases, however, the difference in efficiency becomes more

Size n	**Type**	**DHR** time/cp	**linU_1** time/cp	**linU_2** time/cp	**MTR** time/cp
500	uncorrelated	1.72	1.78	1.74	1.74
500	correlated	1.28	1.42	1.55	1.47
1000	uncorrelated	2.66	2.90	2.85	3.01
1000	correlated	2.30	2.49	2.72	2.59

Table 3.5: *Uncorrelated and weakly correlated data instances. We give the average time per choice point in milliseconds for 100 test sets per size.*

Size	**DHR**		**linU_1**		**linU_2**		**MTR**	
n	cp	$time$	cp	$time$	cp	$time$	cp	$time$
1000	97.83	0.26	72.38	0.21	59.73	0.17	59.73	0.18
2000	161.48	0.79	120.64	0.65	100.38	0.51	100.38	0.56
3000	202.34	1.59	148.43	1.31	118.90	1.00	118.90	1.06
4000	291.00	3.17	205.16	2.43	146.58	1.73	146.58	1.82
5000	360.47	4.82	245.32	3.79	184.83	2.65	184.83	2.98
6000	534.61	9.46	376.69	7.81	197.43	3.84	197.43	4.30
7000	620.48	12.90	431.55	10.11	294.18	6.78	294.18	7.57
8000	823.34	21.08	567.43	16.47	285.22	8.19	285.22	9.23
9000	1051.72	31.76	712.51	23.74	435.65	14.50	435.65	15.46
10000	1143.54	38.39	797.58	30.21	620.35	22.71	620.35	24.99

Table 3.6: *Uncorrelated data. Comparison of running times for the new amortized linear time propagation algorithms and implementations of DHR, and MTR. We give the average time in seconds as well as the number of choice points for 100 test sets per size.*

Size n	linU_2 (time per cp)	MTR (time per cp)
500	1.74	1.74
1000	2.85	3.01
2000	5.08	5.58
4000	11.80	12.42
8000	28.71	32.36
16000	71.71	75.42

Table 3.7: *Uncorrelated data. Comparison of running times per choice point for the new amortized linear time propagation algorithm based on bound U_2 and the implementation of MTR. We give the average time per choice point in milliseconds for 100 test sets per size.*

Size	uncorrelated			weakly correlated		
	linU_2		MTR	linU_2		MTR
n	cp	$time$	$time$	cp	$time$	$time$
10 000	620.35	22.71	24.99	1626.78	60.98	66.58
11 000	629.43	26.38	28.76	2572.45	110.47	121.08
12 000	604.87	28.04	32.31	2590.45	125.40	137.21
13 000	1341.42	69.30	77.31	2694.07	142.13	156.26
14 000	875.71	50.42	56.96	3520.18	206.68	228.54
15 000	1041.80	64.60	70.74	2818.97	185.33	204.80
16 000	1256.73	90.12	94.78	2164.99	154.56	172.14
17 000	1670.81	124.53	139.63	3145.36	250.59	276.93
18 000	2580.28	205.81	227.81	2980.91	251.43	279.63
19 000	2870.68	243.05	274.93	4871.67	435.33	476.97
20 000	2750.36	256.88	288.15	4319.27	405.56	452.50

Table 3.8: *Comparison of running times of linU_2 and MTR on uncorrelated and weakly correlated data. cp is the number of choice points, $time$ the running time in seconds.*

apparent, and linU_2 outperforms MTR (see Tables 3.7 and 3.8).

One drawback of the new methods is the need for an initial sorting step in the preprocessing in which a profit and a weight ordering of all items are calculated. However, timing experiments show, that this initial step costs about 0.06 seconds for 10 000 items and takes less than 0.01 seconds for 1000 items. According to Table 3.6, the total running time for these problem sizes is much higher. Hence, the preprocessing time can be neglected in practice.

3.8 Conclusions

We proposed a formal definition of optimization constraints and relaxed L/U-consistency. Propagation based on these concepts has proved to be quite successful in recent years. Based on relaxation bounds for KP, we introduced a new reduction algorithm that runs in amortized time $\Theta(n)$ for $\Omega(\log n)$ calls. This algorithm can be used efficiently as a propagation routine when solving the CKP in a CP context.

In a CP search, the efficiency of the algorithm developed depends on the number of choice points and the time needed per choice point: The more choice points are investigated during the search, the less dominant the preprocessing times for initialization and sorting. If other routines that propagate additional constraints of the CKP or calculate expensive bounds on the objective require more time per choice point, an effective pruning behavior that justifies a higher effort spent per choice point becomes more important.

Experiments show that in a tree search the algorithm is as effective as another method based on a reduction technique previously proposed by Martello and Toth for KP. However, theoretical analysis and experimental comparisons show, that the new method is asymptotically more efficient. Finally, the routines presented have already been used successfully in combination with other constraints in a Lagrangian relaxation based approach for a multimedia application, see Chapter 7.

4

Airline Crew Rostering

Routing and scheduling of crews and equipment in large public transportation networks such as airlines, railways, and bus companies has been a major field for optimization for a long time. Different planning problems such as fleet assignment, aircraft routing, and crew scheduling have been addressed by numerous research papers. A recent book by Yu [213] and the article by Rushmeier et al. [181] give examples of problems and solution approaches in the airline industry.

Optimizing crew scheduling problems is highly motivated by the potential cost savings in this area. According to Anbil et al. [3], published in 1991, *"a one percent increase in [American Airlines'] crew utilization translates into US-$ 13 million savings each year"*.[1] The *"total crew cost [...] exceeded US-$ 1.3 billion every year and was second only to fuel cost"*. For United Airlines Graves et al. [96] report *"savings of US-$ 16 million annually in crew scheduling costs"* after switching to some crew planning system. Deutsche Lufthansa reported crew costs of €922 million for the passenger segment for the first half of 2002 (see [66]), which is almost one fifth of all expenses in that segment (€5.2 billion).

Nowadays most large airline companies use optimization in crew planning. Nevertheless, improving the underlying algorithms or models can still result in savings of some million Euro per year.

4.1 Solving Airline Crew Rostering Problems

In the *airline crew scheduling problem* a set of basic activities or tasks[2] – typically, to fly an aircraft from A to B – has to be assigned to a set of crew members such that all basic activities are covered. Additionally, complex rules and regulations coming from legislation and contractual agreements have to be met by the solutions. Different objectives like minimizing overall costs and maximizing crew satisfaction are of interest. The latter objective is chosen in

[1] At that time, American Airlines employed more than 8 300 pilots and 16 200 flight attendants [3].

[2] We will use *activity* and *task* synonymously.

so-called preferential bidding systems where crew members are allowed to express their likes and dislikes for certain activities (see, e.g., Gamache et al. [84]).

In larger airlines the crew scheduling problem is decomposed into a crew pairing and a crew assignment (or rostering) problem. In the crew pairing problem the basic activities, *flight legs* (flight without stopover), are grouped into so-called *pairings*. A pairing is a sequence of flight legs usually starting and ending at the same home base and is considered one single "piece" of work. A pairing corresponds to one or more days of work and will be assigned as a whole to one or more named individuals. However, in the crew pairing problem the pairings remain anonymous. Pairings have to respect certain rules, e.g., connection time between flight legs and rest time between duty days. The crew pairing problem has been studied by several authors (see, e.g., Andersson et al. [5], Barnhart and Shenoi [20], Chu et al. [54], Desaulniers et al. [63]).

The topic of this chapter is the *crew assignment problem* (CAP). In the CAP the pairings together with other activities such as ground duties, reserve duties, and off-duty blocks must be assigned to named individuals. The outcome of this is a so-called *roster* for each crew member. Rosters must respect rules and regulations additionally to those considered in the pairing problem. These rules can be classified into *single crew member rules* and *multiple crew member rules*. The rules of the first class only influence the legality of a single roster, whereas those of the second class express regulations on combinations of rosters or crew members. Crew assignment is considered to be a large scale optimization problem, as in practice several thousand pairings and other activities have to be assigned to hundreds or possibly even thousands of crew members. The number of rules and regulations which must be respected can be up to one hundred and the number of possible rosters is gigantic.

4.1.1 The Airline Crew Assignment Problem

Given a set of crew members, a set of pairings, a set of rules and a cost function, a *roster* is an assignment of a subset of pairings to one specific crew member. A *schedule* is a set of rosters such that all rules are obeyed and every pairing is assigned to exactly one crew member. Rules may concern a *single crew member* or *multiple crew members*. Single crew member rules regard each individual crew member's roster, stating for example that no two temporally overlapping pairings can be assigned to the same crew member. Multiple crew member rules aim at more than one crew member, stating for example that two given pairings must be assigned to two crew members out of which at least one must have a certain level of experience. The cost function associates a cost with every legal schedule, and its minimization is desired.

In our case, every rule in the rule set only deals with just one single crew member, and the objective function is linear over the rosters. That means that only single crew member rules can be modeled and that the cost of the entire solution to the CAP is defined as the sum of the costs of the selected rosters. More formally:

Definition 8 *Let* $k, m, n \in \mathbb{N}$ *and let* $C := \{1, \dots, m\}$ *the* set of crew members *and* $T := \{1, \dots, n\}$ *the* set of pairings.

1. *Let* $R := C \times 2^T$. *Every* $r \in R$ *is called a* roster *and* R *is called the* set of all possible rosters.

2. *Let* $B := \{0,1\}$ *and* $H := \{h_1,\dots,h_k \mid h_i : R \to B\ \forall\, 1 \le i \le k\}$. *Every* $h \in H$ *is called a* (single crew member) rule *and* H *is called a* rule set.

3. *A roster* $r \in R$ *is called* legal (with respect to a rule set H) *iff* $h(r) = 1\ \forall\, h \in H$. $L(H) := \{r \in R;\ r \text{ is legal}\}$ *is the* set of legal rosters (with respect to the rule set H).

4. $f : R \to \mathbb{Q}^+$ *is called a* cost function.

5. *The* Crew Assignment Problem (CAP) *is to minimize* $\sum_{1\le i\le m} f((c_i, t_i))$, *where* $(c_i, t_i) \in L(H)\ \forall\, 1 \le i \le m$ *s.t.*

   a) $\{c_1,\dots,c_m\} = C$

   b) $\bigcup_{1\le i\le m} t_i = T$ *where* $t_i \cap t_j \ne \emptyset \Rightarrow i = j\ \forall\, 1 \le i,j \le m$

The model as stated above neither allows non-linear objectives when combining rosters, nor permits to restrict the combination of rosters by additional multiple crew member rules one might be interested in when tackling real life applications. Nevertheless, the methods we present to solve the above problem allow to treat linear multiple crew member rules as well.

4.1.2 Current Solution Methods

To our knowledge, all published work on the airline CAP follows the idea of generate-and-optimize or column generation, see Gamache and Soumis [84], Day and Ryan [61], and Ryan [182] for examples. The problem is divided into a subproblem where a subset of all legal rosters is generated, and a master problem where some of these rosters, one for each crew member, are selected. The master problem is a linear integer program whereas the structure of the subproblem is dependent on the rules, regulations, and objectives of the underlying CAP. Usually, one iterates between the master problem and the subproblem, thus gradually improving the solution quality.

The generation of legal rosters in the subproblem is either done by partial enumeration based on propagation and pruning techniques [131] or by solving a resource constrained shortest path problem where the constraints ensure that only legal rosters are generated. The objective function measures the reduced costs of the roster with respect to the solution of the continuous relaxation of the master problem defined on the previously generated rosters [84]. The latter approach is known as resource constrained shortest path column generation. In this approach, the subproblem can be solved optimally and one can prove that it is possible to obtain the optimal solution to the entire (relaxed) problem without explicit enumeration of all possible rosters.

Resource constrained shortest path column generation is a very powerful technique but it is only useful if rules, regulations and objectives satisfy certain properties (see Desrosiers et al. [65] for an overview). Especially for European airlines though, there are very strict rules enforced by legislation, unions, etc. that define the feasibility of schedules. Propagation and pruning techniques do not depend on particular assumptions on the problem, and thus seem to be a good choice for highly constrained rostering problems.

Several alternative approaches have been proposed for solving the railway CAP, which differs slightly from the airline crew assignment, but also has to model rules and regulations.

Caprara et al. [47] gave an algorithm that uses information from a Lagrangian lower bound based on the solution of an assignment problem to guide a heuristic. Their algorithm won the first prize in a competition organized by the Italian Railway Company. For the Lisbon Underground, Cavique et al. [50] used a Tabu Search procedure working on the entire solution. The contractual and operational rules in this case consist mainly of time limits for minimum/maximum work time and for meal breaks.

For the approach presented here, the work of Caprara et al. [45] is of certain interest. Their approach is mainly based on the constraint logic programming (CLP) paradigm enhanced with a lower bounding procedure taken from the operations research field. This is the natural choice since good and fast bounding procedures were available from [47]. The efficiency of the approach is supposed to be due to the existence of these bounding procedures for the case considered. Thus, a generalization to the generic airline CAP considered here is not straightforward.

4.1.3 The Airline Test Case

For our experiments, we use data for several 4 week planning periods of Spring and Summer 1998 from a major European airline (company A). The problem instances consider cockpit crew at one crew base and consist of around 400 crew members and around 1000 activities which have to be assigned.

For the crew assignment, the airline is interested in fair rosters which minimize overall costs. The costs are represented by a linear objective function on the rosters and the unassigned activities. The main costs of each roster depend on the total flight time (or block time) and costs on extra days off.

In the production system of that airline, around 90 single crew member rules are implemented. Representative subsets of rules and regulations have been chosen for our experiments. However, certain issues such as individual rule relaxations or restrictions are not considered.

The following lists seven rules which have been implemented for the test case presented here. The formulations of some rules use the term *airline day* which is a 24 hours period not corresponding to a calendar day. It starts at time τ on one day and ends one minute before τ on the next day.

(R1) *Minimum Rest at Home Base:* A crew member gets at least an η_1-hour rest period at his/her home base between two activities.

(R2) *Minimum Rest for One Day Off:* If a crew member has one complete airline day off, i.e., the rest between two activities contains one airline day, the rest period must be at least η_2 hours.

(R3) *Minimum Rest for Two Days Off:* If a crew member has two complete airline days off, i.e., the rest between two activities contains two airline days, the rest period must be at least η_3 hours.

(R4) *Minimum Rest for Three Days Off:* If a crew member has three complete airline days off, i.e., the rest between two activities contains three airline days, the rest period must be at least η_4 hours.

(R5) *Latest Check-out for Two Days Off:* If an activity is followed by a rest period containing two airline days, the activity is not allowed to end in a *night period*, i.e., the period defined by an interval $[\tau_1, \tau_2]$.

(R6) *No Early Briefing After Five/Two:* After a 5-day working period followed by two airline days off, a crew member cannot be assigned an activity starting before time τ_3.

(R7) *No two trips on the same day:* Except for preassignments, it is not allowed to have two flight trips on the same day.

Additionally, crew members are required to have special activities like simulator training, medical checks, etc. during the planing period under consideration. These activities usually require interaction with resources that are not necessarily known in the planning system (e.g., slots for the simulator, medical personal, etc.). Therefore, in the present work they are not handled as special rules but simply by assigning these activities to a crew member in advance. We refer to these activities as *preassignments*.

Note, that due to the generic approach based on CP, the rule set can easily be extended and modified without significantly changing the algorithms described in the following sections.

For the column generation approach used in this work, we decompose the CAP into a subproblem, where legal rosters that obey (2.8) are generated, and a set partitioning master problem ensuring that all activities are assigned exactly once.

4.2 A Column Generation Model for the CAP

The definition of the CAP as stated above allows to apply the column generation principle, i.e., it can naturally be decomposed into the *subproblem* of generating legal rosters and the set partitioning *master problem*. The latter one is an integer program (IP) that ensures 5a and 5b of Definition 8.

4.2.1 The Subproblem

For each crew member we generate a set of rosters. A roster for crew member c is a set of activities Y that contains the chosen and possibly preassigned activities P of the crew member and that satisfies all single crew member rules. Each generated roster leads to the introduction of a new column in the master problem. For this purpose, we need the cost coefficient c_Y that is defined by this roster, as well as the coefficients $a_{i,Y}$ for each constraint of the master problem. These coefficients are uniquely defined by a roster and will be calculated when generating the roster. They also allow to check (2.8) and to generate only rosters with negative reduced costs.

The subproblem for each crew member consists of generating a set of rosters. For each roster we have to find a set Y of activities, a cost coefficient c_Y, and the coefficient $a_{i,Y}$ for the master constraint such that the following conditions are satisfied:

1. $P \subseteq Y \subseteq T$, where T is the set of all tasks that have to be assigned,
2. Y satisfies the single crew member rules, and
3. obeys the negative reduced cost constraint $c_Y - \sum_i \lambda_i \cdot a_{i,Y} < 0$.

In the following, we introduce a constraint programming approach in Sect. 4.2.1.1 and show how costs and reduced costs can be determined by finding a path in a graph (Sect. 4.2.1.2). We introduce and show how to encode the reduced cost constraint in this approach (Sect. 4.2.1.3). In order to obtain a powerful propagation, we additionally exploit the path-finding algorithm and encapsulate it in a path constraint (Sect. 4.2.1.4). Section 4.2.2 shows how the propagations of this constraint can be computed efficiently in linear time. Since the path finding approach alone does not take into account all airline rules, we introduce these additional constraints into the constraint model in Sect. 4.2.2.1. Finally, Sect. 4.2.2.2 presents some search heuristics for solving the subproblem.

4.2.1.1 Formulation as a CSP

In this section, we formulate the subproblem as a constraint satisfaction problem allowing us to express complex airline rules and regulations.

For the roster generation problem of crew member c, we introduce a single set variable Y denoting the (unknown) set of tasks of c. The value of Y has to be a subset of the set of tasks T. Initially, the possible set $\text{pos}(Y)$ contains all tasks and the required set $\text{req}(Y)$ contains all preassigned tasks of the crew member. During roster generation, the required set contains all tasks that have already been assigned to the considered crew member, whereas the possible set contains activities that are still candidates to be assigned. Constraint propagation and search decisions will then remove tasks from the possible set or assign new tasks by adding them to the required set. If the required set is equal to the possible set, the set variable is bound and a solution has been found.

Most rules and regulations require the introduction of further constrained variables. For example, for ensuring a minimal rest time after a given number of days off we introduce a constrained integer variable d_i for each task t_i that represents the number of days off directly after this task.

We now give a small example illustrating effects of domain reduction during roster generation. Consider the set variable Y of a crew member c and suppose that its possible set $\text{pos}(Y)$ initially contains the four tasks t_1, t_2, t_3, t_4 satisfying following properties:

$$\begin{array}{ccccc} 0 & \leq & \text{start}(t_2) - \text{end}(t_1) & < & \eta_1 \\ 1\text{day} & \leq & \text{start}(t_3) - \text{end}(t_1) & < & \eta_2 \\ \eta_2 & \leq & \text{start}(t_4) - \text{end}(t_1) & & \end{array} \tag{4.1}$$

If crew member c has no preassigned tasks then the required set $\text{req}(Y)$ is empty initially. Now we start the roster generation procedure which explores different search decisions. For example, it assigns task t_1 to crew member c by adding it to $\text{req}(Y)$. This search decision leads to following domain reductions, which are all executed before another search decision is made.

1. Since t_1 is in $\text{req}(Y)$ and the rest between t_1 and t_2 is smaller than the minimal rest time η_1, the minimal rest time rule removes t_2 from $\text{pos}(Y)$.

2. The earliest task that can follow t_1 is now t_3. Since the rest time between these tasks is greater than 1 day the lower bound $\min(d_1)$ of the variable d_1 representing the number of days off after t_1 is set to 1.

3. Since the rest time between t_1 and t_3 is smaller than the minimal rest time η_2 after 1 day off, the minimal rest time rule for 1 day off removes t_3 from $\text{pos}(Y)$.

Table 4.1 summarizes these domain reductions.

	Initial domain	**Add t_1 to Y**	**Reduced domains**
$\text{req}(Y)$	$\emptyset$	$\{t_1\}$	$\{t_1\}$
$\text{pos}(Y)$	$\{t_1, t_2, t_3, t_4\}$	$\{t_1, t_2, t_3, t_4\}$	$\{t_1, t_4\}$

Table 4.1: *Domain reductions after an assignment*

4.2.1.2 Shortest Path Problems

So far, we have not discussed how to calculate the coefficients c_Y and $a_{i,Y}$ (as introduced in Sect. 4.2.1). In CAPs, the cost c_Y of a roster is often determined by an additive cost function, and in cases where the cost is not completely additive, an additive function will usually be a good approximation. Furthermore, the dominating terms in (2.8) tend to be those associated with the λ coefficients, as there is a strong feasibility component in the problem. These terms are always additive. For each task t selected in Y, we obtain a cost c_t and the resulting cost of the roster is the sum of the c_t's of all selected tasks. The cost c_t does not only depend on task t, but also on the next task t'. For example, c_t can depend on the rest time after t which depends on the arrival time of t and the departure time of t'. In the following, we assume that roster costs are in fact defined by a sum of task costs c_t which depend on the selected tasks t and their successors t' (but not on any other tasks).

Due to this assumption, we can model the problem of finding a cheapest set of tasks by the problem of finding a shortest path in a weighted graph $G = (V, E)$. The nodes of the graph are the tasks plus an additional source s and an additional sink s'.

$$V = T \cup \{s, s'\} \tag{4.2}$$

If task t' has a start time $\mathit{start}(t')$ that is greater than the end time $\mathit{end}(t)$ of task t we introduce an edge (t, t'). Furthermore, we introduce an edge from the source s to any task t and an edge from any task t to the sink s':

$$E = \{(t, t') \in T \times T \mid \mathit{end}(t) \leq \mathit{start}(t')\} \cup \{(s, t),\ (t, s') \mid t \in T\} \tag{4.3}$$

Hence, the graph G is a directed acyclic graph and if we order the tasks in T by increasing start time we obtain a topological order $s, t_1, \ldots, t_n, s'$ on the nodes (see Fig. 4.1).

Each legal roster corresponds to a path from source to sink. The roster costs are obtained as the path costs if the edge weights are given as follows:

1. An edge from source s to node t is labeled with costs $\omega_{s,t} := 0$.
2. An edge from node t to node t' is labeled with the costs $\omega_{t,t'}$ that present the costs for executing t' directly after t.
3. An edge from node t to the sink node s' is labeled with the costs that t will have if it is the last task in the roster.

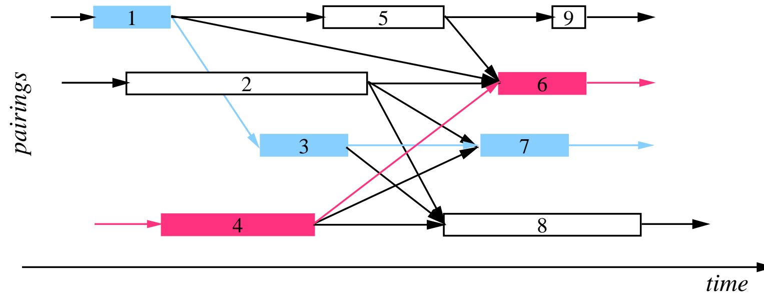


Figure 4.1: *An optimal and legal roster is equivalent to a constrained shortest path in a weighted DAG.*

We can also represent the reduced costs of a roster in this way. We distinguish two kinds of constraints in the master problem. The first kind of constraint ensures that exactly one roster is selected for each crew member. We know a-priori that the coefficient a_i is 1, if constraint i concerns the considered crew member c and 0 otherwise. The second kind of constraint ensures that each task is covered by the required number of crew members. The coefficient a_i of the latter constraint will be 1, if the constraint concerns task t and if task t belongs to the selected path. Otherwise it will be 0. We can therefore simplify (2.8) as

$$-\lambda_c - \sum_{t \in Y} (\omega_{t,t'} - \lambda_t) < 0 \tag{4.4}$$

where λ_t is the dual value of master constraint for task t and λ_c is the dual value of master constraint for crew member c. Edge weights are then adapted correspondingly.

1. For edges from the source s to any task t we only add the negative dual of the crew member, i.e.,

$$c'_{s,t} = -\lambda_c. \tag{4.5}$$

2. Edges from node t to node t' (or to sink s') are labeled with the difference of the initial costs $\omega_{t,t'}$ and the dual value of task t:

$$c'_{t,t'} = \omega_{t,t'} - \lambda_t \tag{4.6}$$

These new costs $c'_{t,t'}$ reflect the reduced costs for performing activity t' after t. We call them also reduced costs of activity t and we distinguish them from the initial costs $c_{t,t'}$ of task t. Each roster with negative reduced costs then corresponds to a path from source to sink with negative costs in the graph with weights $c'_{t,t'}$. However, not every path from source to sink is a legal roster. We can already reduce a large number of illegal paths in a preprocessing step by removing edges. If task t' cannot follow task t since this violates some rule (e.g., on minimal rest time) we suppress the edge (t, t') in the graph. Rules that depend on more than two tasks cannot be treated in this way. For example rule (R6) requiring a day off after 5 days of work depends on more than two successive tasks. In the next sections, we show how constraint satisfaction techniques can be exploited to obtain dynamic reductions of the graph.

4.2.1.3 A Negative Reduced Cost Constraint

We show how to encode the negative reduced cost condition in the CSP-framework. The first idea is to use (4.4) and to directly encode it by a sum-over-set constraint. For the sake of

readability we do not specify the relation of the constraints, but just give the condition that the constraint is imposing on the value of Y.

$$
\begin{aligned}
-\lambda_c - \sum_{t \in \overline{Y}} \overline{\underline{c'}}_t &< 0 \\
\text{where} \quad \overline{\underline{c'}}_t &= \overline{\underline{c}}_t - \lambda_t
\end{aligned}
\tag{4.7}
$$

The initial costs c_t of task t and its 'reduced costs' c'_t are represented by constrained variables. The variable c_t can have a more complex definition involving next-in-set constraints (which will be explained in Sect. 4.2.2.1).

Although this is a straightforward way to express the negative reduced cost constraint, we have to analyze how effective it is. The negative reduced cost constraint is a very tight constraint. If it does not reduce the domains sufficiently early during the search, much search effort will be spent in subtrees of the search space which will not contain any roster of negative reduced cost.

We only have an upper bound on the reduced costs of a roster, namely 0. The sum-over-set constraint uses this upper bound as follows. First, it determines a necessary lower bound lb by summing up the lower bounds $\min(c'_t)$ of the tasks. If lb is strictly greater than 0 an inconsistent search state is reached and backtracking occurs. Otherwise, if the bound lb is smaller than 0 the constraint checks for each element t in $\text{pos}(Y) \setminus \text{req}(Y)$ whether its selection would make the lower bound greater than 0. If so, the task t will be removed from the possible set.

This method is only efficient, if a good lower bound is computed. The lower bound is defined as follows:

$$
lb := -\lambda_c - \sum_{t \in req(Y), min(c'_t) \geq 0} min(c'_t) + \sum_{t \in pos(Y), min(c'_t) < 0} min(c'_t) \tag{4.8}
$$

Suppose that almost all tasks could have negative reduced costs c'_t. The sum-over-set constraint does not know any incompatibilities between tasks and assumes that all tasks in $\text{pos}(Y)$ could be selected. In this case, we obtain a very small negative lower bound, which probably will not lead to any domain reductions. Domain reductions will only occur deep in the search tree when the size of the possible set comes close to the size of the required set.

The graph introduced in Sect. 4.2.1.2 avoids a large number of incompatible tasks. If task t' cannot follow task t there is no edge from t to t'. A path of the graph never contains two successive tasks that are incompatible. As a consequence, a shortest path in the graph leads to a much tighter lower bound and to better propagation.

In the next section, we introduce a path constraint that implements this approach and that provides a significant improvement over the sum-over-set constraint. We nevertheless keep the sum-over-set constraint in order to be able to measure the improvements. In our experiments we use a simplified implementation of this sum-over-set constraint that avoids all calculations for propagations which will probably not be effective anyway. It just provides the basic pruning rule that compares the calculated lower bound lb with the given upper bound 0. Hence, no domain reduction is caused by this constraint. It thus provides an interesting basis for comparisons, because it is fast and does not lead to any computational overhead.

4.2.1.4 Cost based Filtering for Shortest Paths on DAGs

In this section, we introduce a path constraint for acyclic graphs that provides a powerful propagation for the negative reduced cost condition.

The path constraint is defined for a directed acyclic graph $G = (V, E)$, the edge costs $c_{i,j}$, a set variable Y, and a variable z. The graph G has the same form as in Sect. 4.2.1.2 except that we use numbers for representing the nodes. If the tasks are ordered by increasing start time as follows $t_1, \dots, t_n$ we replace t_i by i, s by 0, and s' by $n+1$. Since there is no edge from i to j if $j < i$, the node numbers represent a topological order.

To simplify the notation we define $t_0 := s$ and $t_{n+1} := s'$. The path constraint then has the variables Y and z and is defined by two conditions:

1. Y represents a path in the graph from source s to sink s', i.e., $\overline{Y} \subseteq T$ and

$$\{(i,j) \mid t_i \in \overline{\underline{Y}} \cup \{s\},\ t_j = \text{next}(t_i, Y)\} \subseteq E \tag{4.9}$$

2. z is the sum of the edge costs

$$\overline{\underline{z}} = \sum_{\substack{t_i \in \overline{\underline{Y}} \cup \{s\}, \\ t_j = \text{next}(t_i, Y)}} c_{i,j} \tag{4.10}$$

where $\text{next}(t, Y)$ denotes the successor of t on the path Y. Since we have a topological order on G, this successor is uniquely defined as

$$\text{next}(t, Y) = \min\{t' \in \overline{\underline{Y}} \cup \{s'\} \mid t' > t\} \tag{4.11}$$

$$\text{where} \quad t_j > t_i \quad \text{iff} \quad j > i.$$

In the following, we introduce propagations of the path constraint that are important for roster generation. In this case, we just have the upper bound 0 on the path costs z. The propagations require the calculations of paths from source to sink that contain all required tasks and only possible tasks (i.e., the set of nodes Y of such a path has to satisfy $\text{req}(Y) \subseteq Y \subseteq \textit{pos}(Y)$). We call such a path *admissible*. The propagation rules are:

1. If the bound $\min(z)$ is smaller than the costs lb of the shortest admissible path (or $+\infty$ if there is no admissible path) we replace it by lb. If the new bound is strictly greater than the upper bound $\max(z)$, a failure is obtained as a consequence.

2. If the costs of the shortest admissible path through node i for $i \in \text{pos}(Y)$ are strictly greater than the upper bound $\max(z)$ we can remove i from $\text{pos}(Y)$.

In the next section, we show how shortest admissible paths can be computed efficiently. Furthermore, we discuss how to maintain these paths when incremental updates occur.

4.2.2 Implementation

To compute a lower bound on the reduced costs, we just have to solve the *single source shortest path problem* (SSSP) starting with node $\boldsymbol{s}$. We denote the single source shortest path distance from $\boldsymbol{s}$ to $\boldsymbol{s'}$ by $y^s_{s'}$ and call it the *SSSP-distance*. The SSSP-distance is the desired lower bound on the reduced costs of all legal rosters, as each of them can be identified with a $(\boldsymbol{s}, \boldsymbol{s'})$-path.

To solve the SSSP, we apply an algorithm that makes use of the acyclic structure of the graph. It visits nodes in topological order and thus runs in linear time $O(|V| + |E|)$. Furthermore, it does not require that edge costs are positive (see Cormen et al. [56] for details).

However, we have to ensure that the algorithm determines only nodes which are subsets of $\boldsymbol{pos}(Y)$ and supersets of $\boldsymbol{req}(Y)$. For this purpose, we consider only nodes in $\boldsymbol{pos}(Y)$ and we ignore all edges $(\boldsymbol{i}, \boldsymbol{j})$ that go around a node of $\boldsymbol{req}(Y)$. That means, if there is a $\boldsymbol{k} \in \boldsymbol{req}(Y)$ s.t. $\boldsymbol{i} < \boldsymbol{k} < \boldsymbol{j}$ we do not consider $(\boldsymbol{i}, \boldsymbol{j})$. This can be done efficiently by determining the smallest element $\boldsymbol{k} \in \boldsymbol{req}(Y)$ that is strictly greater than $\boldsymbol{i}$ and ignoring all edges $(\boldsymbol{i}, \boldsymbol{j}),\ \forall \boldsymbol{j} > \boldsymbol{k}$.

During the search for a legal roster, the domain of Y changes frequently, which means that many and often similar SSSPs have to be solved. We therefore developed an incremental version of the algorithm that computes the differences in the domains of the current and the last iteration. The required set may have grown and the possible set may have shrunken. If a new node became required, we need to restart the SSSP-algorithm with this node and can stop when we first run over an already formerly required node; its difference in distance then applies to all following nodes as well. The algorithm follows the support idea in the style of AC-6 (Bessière [31]). If a node $\boldsymbol{i'}$ left the possible set, we check if any adjacent node $\boldsymbol{i}$ is affected by that change. We call $\boldsymbol{i'}$ the *support node* for $\boldsymbol{i}$.

If the node $\boldsymbol{i}$ is affected, its support may be replaceable by another node without a change in the distance y^s_i to the start node $\boldsymbol{s}$. If this is not the case, we need to do the distance update and mark the successors of node $\boldsymbol{i}$ for a continuing update on all affected nodes. Notice, that the updates for all nodes can be done in one pass.

Of course, if the lower bound becomes greater than zero, the search for a legal roster with negative reduced costs fails and we backtrack. If not, we can use the distance information just computed to reduce the domain of Y further.

To eliminate activities from $\boldsymbol{pos}(Y)$, we must determine the cost of a shortest admissible path going through node $\boldsymbol{i} \in \boldsymbol{pos}(Y)$. For directed acyclic graphs, this cost is simply the sum of the costs y^s_i of the shortest path from the source $\boldsymbol{s}$ to node $\boldsymbol{i}$ and the costs $y^{s'}_i$ of the shortest path from $\boldsymbol{i}$ to the sink $\boldsymbol{s'}$. The latter can be computed with the same algorithm by just inverting all the edges and by applying it to the sink $\boldsymbol{s'}$. If $y^s_i + y^{s'}_i$ is non-negative, we remove $\boldsymbol{i}$ from the possible set. This idea was originally proposed by Aneja et al. [6].

We summarize our discussion in Alg. 5. To have a compact representation we omit the technical details of the incremental variant.

4.2.2.1 Single Crew Member Rules

The rules and regulations coming from legislation and contractual agreements are formulated by constraints over set variables, such as cardinality constraints, sum-over-set constraints, next- and previous-in-set constraints, or set-of constraints. Most of them are provided by the constraint library ILOG SOLVER, others have been implemented as extensions of SOLVER.

Algorithm 5 (Non-Incremental) Shortest Path Constraint

for all $i \in T$ **do**
  $y_i^s \leftarrow \infty;\ y_i^{s'} \leftarrow \infty;$
$y_s^s \leftarrow 0;\ y_{s'}^{s'} \leftarrow 0$ // *Init source and sink*
$k \leftarrow 0;$
// *determining shortest path from source* s *to all nodes*
for all $i \in V$ taken in increasing topological order **do**
  if $(k \leq i)$ **then**
    $k \leftarrow \min\{l \in \mathrm{req}(Y) \mid l > i\}$; // *get next required node in the top. ordering*
  for all $j \in pos(Y)$ s.t. $(i,j) \in E$ and $j \leq k$ **do**
    if $(y_j^s > y_i^s + c_{i,j})$ **then**
      $y_j^s \leftarrow y_i^s + c_{i,j}$
// *determining reverse shortest path from sink* s' *to all nodes*
$k \leftarrow 0;$
for all $i \in V$ taken in decreasing topological order **do**
  if $(k \geq i)$ **then**
    $k \leftarrow \max\{l \in \mathrm{req}(Y) \mid l < i\}$; // *get next required node in the top. ordering*
  for all $j \in pos(Y)$ s.t. $(j,i) \in E$ and $j \geq k$ **do**
    if $(y_j^{s'} > y_i^{s'} + c_{j,i})$ **then**
      $y_j^{s'} \leftarrow y_i^{s'} + c_{j,i}$
if $(y_{s'}^s > min(z))$ **then**
  $min(z) \leftarrow y_{s'}^s$ // *propagate new lower bound on costs*
for all $i \in pos(Y)$ **do**
  if $(y_i^s + y_i^{s'} > max(z))$ **then**
    $pos(Y) \leftarrow pos(Y) \setminus \{i\}$ // *exclude nodes that are too expensive*

Given the flight time f_t (block hours) of each task t and a maximal flight time F, we can express a maximal flight time rule by a sum-over-set constraint

$$\sum_{t \in \underline{\overline{Y}}} f_t \leq F. \tag{4.12}$$

In order to express a minimal rest time rule between two successive tasks, we need the arrival time $end(t)$ of a task t and the departure time $start(t')$ of the next task t'. Given a minimal rest time η between two tasks, the minimal rest time rule can then be expressed by the next-in-set constraint (4.11) as follows:

$$start(next(t,Y)) - end(t) \geq \eta \tag{4.13}$$

where $\mathrm{next}(t,Y)$ is defined as in (4.11).

Regulations on days off, maximal duration of working periods, and so on, can be expressed as well, but usually lead to more complex constraint models. In order to facilitate this modeling task, a specific ROSTER LIBRARY has been developed in the scope of the PARROT project [165]. Complex crew regulations that have been formulated by expressions of the ROSTER LIBRARY are automatically translated into compact constraint models of SOLVER.

In real-world CAPs the rules and regulations will vary from crew member to crew member. Quite often individuals only differ with respect to parameters (e.g. reduced flying time). This

does not affect the number of rules or the structure of these. It may also be, that there has to be some kind of compatibility between attributes (e.g. qualifications) of the crew member and attributes (requirements) of the tasks. This is also modeled with generic rules, where the results clearly depend on the crew member in question. A common rule requires that the aircraft type flown must be in the set of aircraft types, for which the crew member is qualified. We may have cases where different crew members work under completely different agreements. This increase the number of rules, but not necessarily the number of constraints, as some rules will not apply to all crew members.

Since the resulting models become quite complex, we illustrate the possible propagations just for two rules, namely (R1) and (R2). For (R1) (Minimum Rest at Home Base) we get a constraint model that excludes all those activities from the possible set of Y that cannot follow legally any already required activity. Rule (R2) (Minimum Rest for One Day Off) is modeled such that any possible activity that follows a required one with one airline day in between is excluded from further search, if the rest time is less than η_2. The corresponding propagation rules are given in Alg. 6.

Algorithm 6 Propagation Algorithm for Rules (R1) and (R2)

for all $t \in req(Y)$ **do**
  for all $t' \in pos(Y) \setminus req(Y)$ **do**
    if $(start(t') - end(t) < \eta_1)$ **then**
      $pos(Y) \leftarrow pos(Y) \setminus \{t'\}$ // *Rule R1*

for all $t \in req(Y)$ **do**
  for all $t' \in pos(Y) \setminus req(Y)$ **do**
    if $(one_airline_day(t,t')$ **AND** $(start(t') - end(t) < \eta_2))$ **then**
      $pos(Y) \leftarrow pos(Y) \setminus \{t'\}$ // *Rule R2*

4.2.2.2 Search Heuristics

The shortest path constraint and the negative reduced cost constraint help to reduce the search effort in pruning parts of the search tree without using the special structure of the underlying problem. However, for moderate to large size real-world problems this pruning alone is not sufficient for finding promising rosters. Heuristics designed for the special structure of the CAP are needed to prune further and/or guide the search to promising regions of the search space. The search space is explored by Alg. 7.

Algorithm 7 A Non-Deterministic Algorithm Describing the Search Tree

while $(pos(Y) \neq req(Y))$ **do**
  1: select a task $t \in pos(Y) \setminus req(Y)$ s.t. t is on a shortest admissible path
  2: chose $(req(Y) \leftarrow req(Y) \cup \{t\})$ **OR** $(pos(Y) \leftarrow pos(Y) \setminus \{t\})$
  3: apply constraint propagation algorithm

The task selection (Step 1 in Alg. 7) should take care of first selecting *the best* activity from the possible set. It is often possible to sort activities during preprocessing (*static ordering*). In

case the order of activities depends on the search history, the ordering can also be carried out during the search (*dynamic ordering*).

As static ordering method one can choose the first possible successor of the last activity that is already required. This first activity first approach is believed to be good at generating productive rosters with only little idle time between activities. In general, these rosters have a low cost, due to the fact that airlines typically prefer such rosters to those with much unproductive time between two activities. This static ordering is also used by Ryan [182].

We get a dynamic ordering when using the shortest path constraint and choosing the pairing on the path that contributes the lowest value to the path costs. The *Lowest Reduced First (LRF)* method selects a task with the lowest reduced cost and proves to be superior in our experiments. The advantage of LRF is that the costs contributed by the pairing to the resulting costs of the roster are also taken into account.

For the branching order (Step 2 in Alg. 7) we try to add the selected task to the required set of Y in the first branch and to remove it from the possible set of Y on the second branch.

4.2.3 The Master Problem

To combine the rosters generated by solving the subproblem to one entire work plan, the master problem has to be solved. It consists of minimizing the total cost (the sum of all chosen rosters) by choosing exactly one line of work for each crew member, such that all pairings are assigned exactly once. This problem can be stated as *set partitioning problem (SPP)*.

$$\text{SPP:} \qquad \min c\,x, \quad \text{s.t. } A\,x = 1, \quad x \text{ binary} \tag{4.14}$$

We are also interested in the *set covering problem (SCP)* as we use it as a relaxation of the SPP:

$$\text{SCP:} \qquad \min c\,x, \quad \text{s.t. } A\,x \geq 1, \quad x \text{ binary} \tag{4.15}$$

where A is a 0-1 matrix, and c is the cost vector for the columns. Both problems are well studied and good algorithms solving (4.14), (4.15) can be found in the literature (see e.g., Caprara et al. [43, 44], Hoffman and Padberg [111], Wedelin [208]).

A legal roster Y for a crew member is translated into a column of matrix A by inserting 1's into the row corresponding to the crew member and into all rows that correspond to activities used in Y. Especially in the beginning, however, it is unlikely and hard to guarantee that the columns generated so far will fit together exactly. Having n crew members and m activities, we extend A by unit-vectors e_i, $i = 1, \ldots, n+m$, ensuring that the resulting set partitioning problem always has a feasible solution. The so-called *dummy rosters* can be interpreted as crew members without work ($i = 1, \ldots, n$), and uncovered activities ($i = n+1, \ldots, n+m$), respectively. To find meaningful solutions, we have to penalize uncovered pairings by assigning high costs to the columns representing them.

Obviously, solutions consisting of many dummy rosters are not of interest. Therefore we relax the last n constraints to SCP constraints, i.e., every activity must be covered *at least* once. Due to the high costs of the dummy rosters, a solution will contain few or even no dummy rosters. But in the solution some of the chosen columns might collide in the sense that there are activities that are assigned more than once. Then, a solutions respecting SPP

constraints is heuristically obtained by adding new legal subrosters extracted from colliding ones.

However, in some cases it is not possible to simply fix collisions without generating new rosters in the subproblem. E.g., a rule limiting the minimum working time for a roster might not allow to build subrosters of a colliding roster. In such a case, the solution of the SPP will contain dummy rosters and the subproblem has to generate different rosters from scratch using new dual information. In Chapter 5 we present a systematic way to master this problem.

4.2.3.1 Multiple Crew Member Rules

In real world applications of crew assignment, there is also a class of multiple crew member rules that involve properties of *several* rosters. E.g., for cabin crew it might be necessary to have at least n_1 people speaking a certain language on a certain flight. Another typical example is to have at most n_2 unexperienced persons in the cockpit. As the subproblem only handles single rosters, these rules have to be handled in the master problem.

A simple way of integrating linear multiple crew member constraints (as the ones sketched above) is to extend the SPP by additional constraints. These constraints are usually expressed by a weighted sum over some attributes of the crew members. The only complication arising then is to compute the reduced costs when generating rosters in the subproblem. To do this, the column generator just needs to have knowledge about the coefficients of the multiple crew member constraint that will arise when adding an activity to a roster. In essence (4.8) is adapted by adding dual values of all those multiple crew member rules in which the current crew member provides a nonzero attribute.

Therefore, multiple crew member rules do not change the behavior of the negative reduced cost constraint in principal. Thus to keep the setting simple we will not use multiple crew member rules in the experiments (Sect. 4.4).

4.3 Overview of the Entire Approach

With the different components discussed before, we are now able to describe their interaction. First, we set up the master problem and add dummy rosters (i.e., crew members without work and unassigned activities) to ensure the existence of a solution. We additionally append a certain number of initial rosters for each crew member. Then we initialize the column generator and define the search strategies to be used.

Entering the (outer) loop of *master iterations* (see Fig. 4.2), we solve the current SPP-IP and get a first current solution and new dual values of the LP-relaxation. The column generator then defines the next subproblem to be solved by picking a crew member and maybe additionally applying other search limitations as, e.g., fixing some assignments according to a time window focus and the current SPP solution. We start generating a specified number of rosters, then add the corresponding columns to the master problem, solve its continuous relaxation, and update the current dual values. This inner loop corresponds to Alg. 4.

After rosters have been generated for all crew members, we solve the set covering relaxation of the SPP master problem and generate subrosters of the rosters used in the SCP optimal solution to resolve collisions in the SPP. Afterwards, the SPP itself is being solved and a new loop begins.

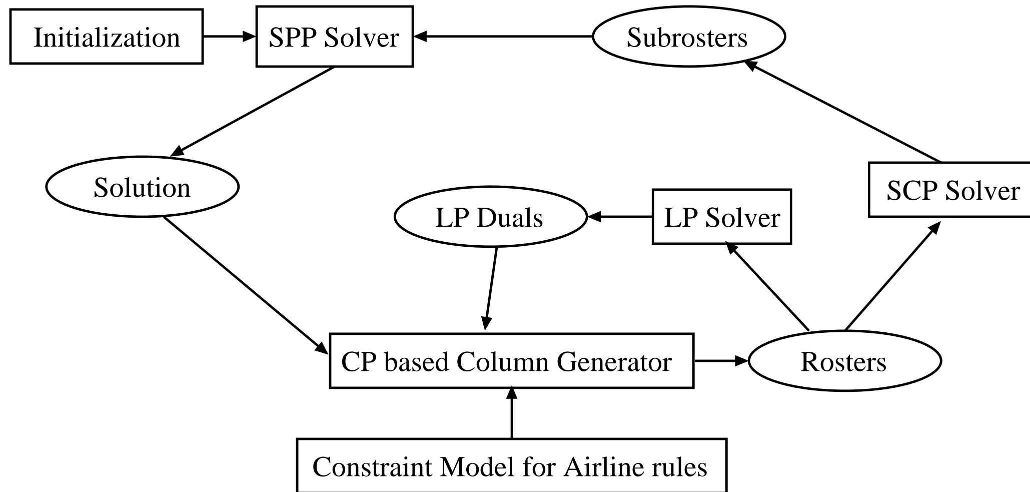


Figure 4.2: *The entire approach: The inner loop generates columns using dual information, the outer loop solves the master problem.*

This process is either interrupted after a given time limit has been exceeded or when no more rosters have been generated that yield to an improvement of the LP-relaxation in any of the subproblems.

4.4 Numerical Results

In this section, we show that constraint programming based column generation is able to solve non-trivial crew assignment problems. In particular, we demonstrate the effect obtained by the propagation of the path constraint.

As mentioned before, the problem instances used for the experiments stem from a major European airline. The rules, regulations, and objective function have directly been abstracted from the real-world case and preserve the essential characteristics of this case. The data sets are sufficiently large to measure the effects of constraint propagation, but they are small enough to run experiments in a reasonable time frame.

To characterize an instance, we specify the number of crew members, the number of preassigned activities, and the number of activities to be assigned. For example, an instance of type 67-165-280 consists of 67 crew members, 165 preassignments, and 280 tasks. All experiments were run on a SUN Ultra 4 with 296 MHz CPU and 1024 MB main memory. For the constraint model of the CAP, the ROSTER LIBRARY [165] based on ILOG SOLVER 4.4 [116] was used. The LP and IP problems were solved with ILOG PLANNER 3.3 [115].

It is important to note that the size of a problem instance is not only determined by the number of crew members and tasks, but also by the number of subtasks (e.g. flight legs and ground duties) and the number of attributes per tasks. In the example above, the 280 tasks consist of 1422 sub-tasks. The given rules and regulations are formulated with the help of 66 attributes per task, 32 per sub-task, and 7 per crew member. A part of these attributes is translated into constrained variables.

In the experiment for Fig. 4.3 we show the effects of negative reduced cost constraint (NRC)

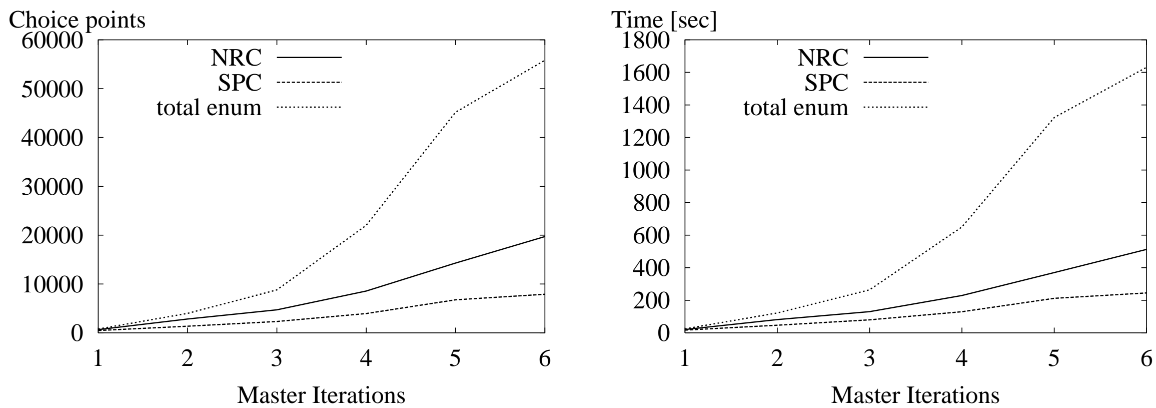


Figure 4.3: *Number of choice points versus master iteration (left), and running time versus master iteration (right) for SPC, NRC, and total enumeration. The tests were run with a data instance of type 10-0-20 that was solved to optimality. (Lines connecting measuring points are given for easier readability and do not mark intermediate values.)*

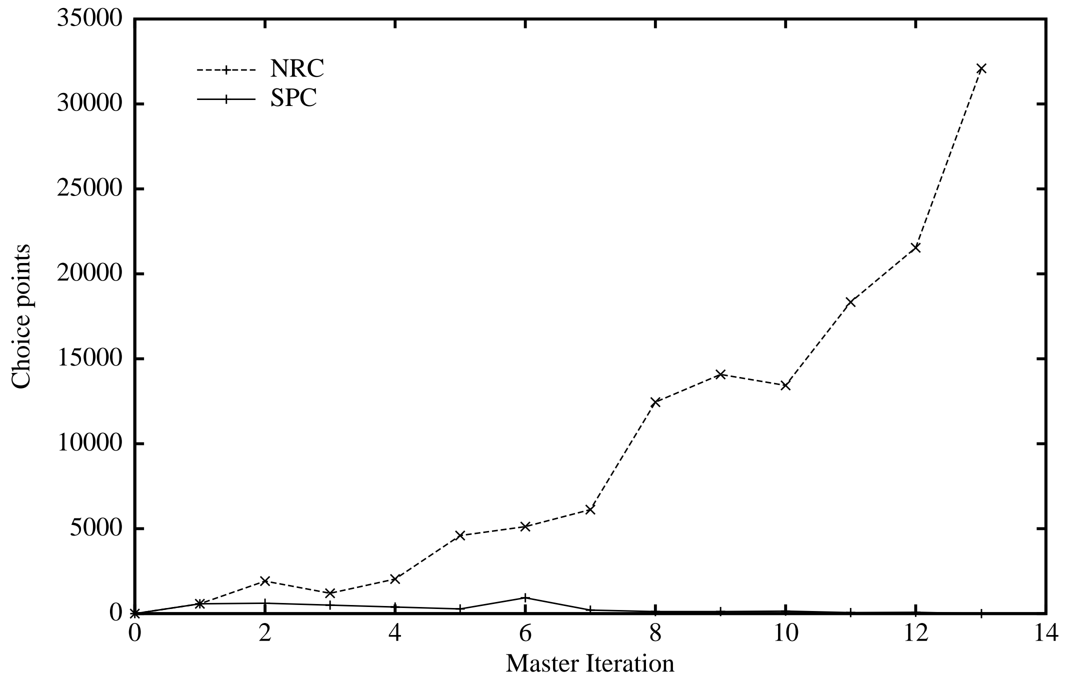
and shortest path constraint (SPC). We compare both results with a total enumeration, where neither NRC nor SPC is used to reduce the search space. The left picture shows the reduction in choice points. In the end SPC used less than half the number of choice points than the NRC. This gain is not consumed by a significant increase in computation time per choice point. As shown in the left figure the decrease in running time is quite similar to the decrease in the number of choice points. As expected, total enumeration is not competitive at all.

To demonstrate the superiority of the shortest path constraint against the negative reduced cost constraint in more detail we run a test on a small instance where in each master iteration the number of choice points was noted. Figure 4.4 again shows that the shortest path constraint uses much less choice points than the negative reduced costs constraint. Furthermore, in the last iteration the shortest path constraint does not find any columns with negative reduced costs anymore, thus proving optimality for the continuous relaxation of the master problem. The negative reduced cost constraint, however, still visits an increasing number of choice points per iteration.

One reason for the efficiency of the shortest path constraint and the reason why there is almost no gap between the reduction in choice points and the reduction in time is the use of the incremental version as mentioned in Sect. 4.2.2. In Fig. 4.5 we compare a non-incremental version of the shortest path constraint with an incremental one. For a fixed time of $10\,000$ sec. for the entire optimization the faster incremental version uses only 2 000 seconds for the propagation, whereas for the non-incremental version almost 60% of total calculation time goes into that part of the algorithm. Thus, the incremental version allows to perform nearly 3 times as many propagations as the non-incremental version and hence helps to improve solution quality.

Figure 4.6 shows a time versus quality comparison of NRC and SPC. After a first big drop in the objective, the NRC falls into huge search trees that only consist of rosters with non-negative reduced costs. The SPC can prune those search trees much earlier and therefore continuously reduces the objective without stalling.

The numerical results clearly prove the potential of SPC and that the overhead created in the subproblem pays off when comparing it to NRC. However, our experiments also showed



Constraint	choice points per master iteration												
	1	2	3	4	5	6	7	8	9	10	11	12	13
SPC	574	609	504	392	280	931	210	119	119	147	63	77	0
NRC	574	1918	1197	2037	4599	5117	6118	12446	14077	13433	18340	21532	32095

Figure 4.4: *Number of choice points versus master iteration using SPC, NRC with a data set of type 7-0-30.*

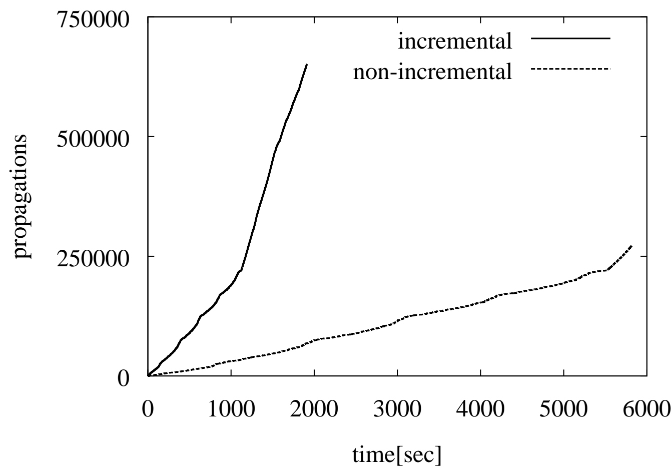


Figure 4.5: *The picture shows time versus the number of calls of the propagation routine using the incremental and the non-incremental implementation of the shortest path constraint. Both versions were stopped after* 10 000 *seconds total CPU time. The experiment was run with a data instance of type 10-0-70.*

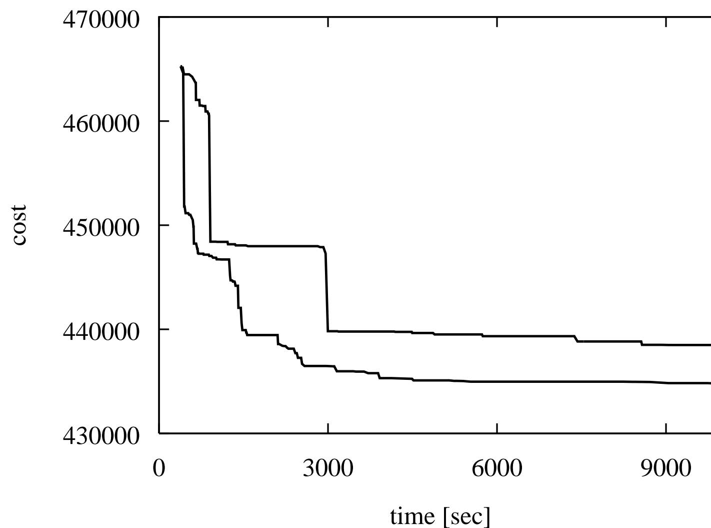


Figure 4.6: *Time versus quality on a data instance of type 67-165-280. The picture shows a comparison of NRC (upper curve) and SPC (lower curve).*

that there is room for improving the generation subproblem, using various techniques to limit the search.

4.5 Conclusions

We have introduced a CP-based column generation approach for the airline CAP. For the case of European airlines with complex rules and regulations common approaches using constrained shortest path algorithms to solve the subproblem in a column generation framework are limited. We therefore formulated the subproblem as a constraint satisfaction problem. Tests with real data from a major European airline showed that the development of a new shortest path constraint combining methods from CP and OR yields a significant decrease in the number of choice points during the generation. An incremental update implementation of this constraint reduces the computational effort per choice point, such that overall computation times are reduced significantly as well. Branching variable selections based on shortest path information further improve the performance.

The presented approach is an example of a successful integration of CP and column generation. We believe that this approach will prove useful for a number of important optimization problems which are currently solved using only OR or only CP methods. There are still refinements and improvements to be done, but this work demonstrates the applicability and efficiency of the approach.

5

Hybrid Approaches for Airline Crew Rostering

Within the PARROT-Project, we developed two different approaches to tackle the Airline Crew Assignment Problem (CAP):

- One following the CP based Column Generation Framework (CGA) (Junker et al. [125], Fahle et al. [73]). We presented that one in the previous chapter,
- and another based on heuristic tree search approach (HTS). This one was designed and implemented by the University of Athens (see Stamatopoulos et al. [194]).

We have seen in the previous chapter that the first one was able to produce reasonably good solutions for airline company A. For two other European companies (companies B and C), results of the CP based column generation approach were not satisfying. Also a CP based heuristic tree search approach was not able to tackle those problems successfully. In this chapter, we show how to combine the two approaches to overcome their inherent limitations. The resulting hybrid approach is able to solve problems of companies B and C, and we present experimental results for these two test cases.

5.1 The Airline Test Cases

We consider test cases stemming from two European airline companies. The instances of company B consist of 50–65 crew members and 766–959 pairings. Company C has 7–30 crew members and 129–279 pairings. Case B covers a planning period of one calendar month, while data sets for C cover two weeks. While case C incorporates mainly 1–2 day pairings, B considers pairings of duration less than 24 hours.

The objective of company C is to achieve a fair distribution of activities over all crew members, whereas in B we aim at satisfying as many preferences expressed by the crew members as possible by minimizing dissatisfaction.

Importantly, the rule sets in both cases are distinct. In B, typical rules such as succession rules and rest time rules, but also more complicated ones like rules ensuring a minimum of days off within gliding windows of variable lengths are incorporated. Also, rules guaranteeing minimum and maximum flight time are enforced. All rules in B are hard constraints, meaning that if they are violated, the solution is considered infeasible.

In C, we consider flight time rules that limit the time actually flown by the crew within certain time periods. These rules are also strict.

The main difference between the two test cases regarding the algorithms we developed is due to the fact that company C does not insist on a partitioning of the work, i.e. in that test case restriction 5b (definition 8) is relaxed to $\bigcup_{1 \leq i \leq m} t_i \subseteq T$. Obviously, this difference requires that our algorithm is able to incorporate two different types of master problems.

5.2 Heuristic Tree Search Constraint Programming Approach

In the the heuristic tree search CP approach (HTS), each complete feasible solution of the CAP is constructed by solving the corresponding constraint satisfaction problem — see Stamatopoulos et al. [194]. The problem is modeled by a set of variables, which correspond to assignable pairings. For each pairing, there is a variable the domain of which represents the crew members that can possibly be assigned to the pairing.[1] For each constrained variable representing the assignment of a pairing, its initial domain comprises all available crew members. The posting of the appropriate constraints reduces the domains of these variables by removing crew members that cannot be allocated to the corresponding pairings. This is possible, for example, due to preassigned activities, or to regulation violations because of the crew member's history, etc. The search tree of the problem is created by iterating over pairings in some specific way and assigning each pairing to a crew member.

Each level of the tree corresponds to the assignment of a pairing. The branch followed from a node represents the allocated crew member to the pairing. Each non-leaf node corresponds to a partial assignment, identified by the path from the root to the node. Leaf nodes correspond to complete legal assignments, i.e. (not necessarily optimal) feasible solutions of the problem. Each allocation of a crew member to a pairing activates the constraint propagation mechanism. More branches of the tree are pruned, as values which are inconsistent with the posted constraints are removed from variables' domains. For example, the assignment of a pairing to a crew member causes the removal from the domain of the crew member of all other pairings that overlap with the one just assigned. When a node is proved to be a dead-end, which means that one or more pairings cannot be assigned to any crew member, backtracking occurs, and decisions taken before are reconsidered.

The constraints of the problem are the regulations of the airline at hand that dictate which rosters are acceptable and which are in violation of the airline rules. A solution to a constraint satisfaction problem is any assignment of values to variables that respects all constraints. A feasible solution to the CAP, formulated as a constraint satisfaction problem, is any assignment

[1] It is assumed that every pairing can only be assigned to one crew member. In case there are more than one crew members necessary to staff a pairing, copies of the pairing are created, and each copy can again only be assigned to a single crew member.

of crew members to pairings such that all airline rules and regulations are respected. Then, the objective function is optimized by searching for improving solutions only.

5.2.1 Tree Traversal

A variety of search methods for traversing the problem tree exists in the literature and we will use DDS and a variant of LDS in this chapter. In addition, we use Large Neighborhood Search.

5.2.1.1 Modified Exact Discrepancy Search

We implemented a variant of LDS (Harvey and Ginsberg [105]). Our variant searches paths with discrepancies lower down the tree before ones with discrepancies higher. Its advantage is that time consuming descends from near the root towards the leaves are avoided. Also, our variant is not iterative. It searches those paths having i or less discrepancies and then exits. Thus, it is not complete. Practically, however, the parameter i can be chosen so that a big enough portion of the tree is explored. In our experiments, this portion of the tree was much bigger than a modern computer could explore in a reasonable amount of time. We call this variant *modified Exact Discrepancy Search* (mEDS).

5.2.1.2 Large Neighborhood Search

Large Neighborhood Search (LNS), introduced by Shaw [190], incorporates local search techniques within the CP framework. The idea is to restrict the search within a fragment of the problem search space. In this way, minor local improvements can be made, which would go unnoticed by most search methods. A reduced search space for a problem with a set of unknown variables V and a known feasible assignment $\mathcal{A}$ can be created as follows: A large subset V_1 of V is selected. All assignments in $\mathcal{A}$ for variables in V are fixed and thus a partial solution is created. Search is performed in the remaining variables with any of the above search methods. After this search is finished (either because the search subspace has been exhausted or any other termination criterion is met), another subspace is selected and the process is repeated. The advantage of LNS is that local improvements are discovered easily, and the objective value is improved quickly. The disadvantage is that the search space cannot be viewed globally. Thus, it is likely that important improvements are missed. A rational strategy when using LNS is to use one of the search methods above in the beginning to guide the search towards a promising area of the search space and to use LNS afterwards to resolve minor local conflicts.

5.3 Constraint Programming based Column Generation Approach

The CP based column generation approach (CGA) was described in the previous section. It divides the airline crew rostering problem into a master problem and a subproblem. The master problem is a set covering or a set partitioning problem. Since we will modify the master problem for the hybrid approaches, we start with a more detailed definition of it:

1	1	1	1	1															=1
					1	1	**1**	1	1	1	1								=1
												1	1	1	1	1	1	1	=1
(1)		1	1		1		(1)	1			1		1		1	1			**=2**
	1		1			1		1	1		1	**1**		1		1	1		=1
		1		1			**1**			1					1				=1
1				1	1						1		1						=1
	1	1				1	**1**		1	1				1	1		1		=1
	1			1	1				1	1		**1**	1				1		=1
1			1			1		1						1		1			=1

Figure 5.1: *Conflicting columns in the CGA when considering a set partitioning problem as the master problem*

The set partitioning problem used in airline crew assignment can be formulated as:

$$
\begin{aligned}
\min \quad & \sum_{i=1,\ldots,k} f((c_{\varphi(i)}, t_i))\, x_i \\
\text{s.t.} \quad & \sum_{\substack{i=1,\ldots,k \\ \text{and } \varphi(i)=j}} x_i = 1 \qquad j = 1, \ldots, m \qquad (5.1) \\
& \sum_{\substack{i=1,\ldots,k \\ \text{and } s \text{ belongs to } t_i}} x_i = 1 \quad s = 1, \ldots, n \qquad (5.2) \\
& x_i \in \{0, 1\}
\end{aligned}
$$

where $\varphi : \{1, \ldots, k\} \to \{1, \ldots, m\}$ maps a column id to a crew member. The m constraints in (5.1) assign exactly one line of work to each crew member. The n constraints in (5.2) ensure that all activities are covered exactly once. In this model, every (legal) roster corresponds to a 0-1 column.

5.3.1 Problems with Column Generation

When being applied to instances of companies B or C two drawbacks of the column generation approach were identified:

Firstly, CGA has difficulties in finding feasible set partitioning (SPP) solutions to the master problem (see Figure 5.1). Finding a feasible solution to the SPP is NP-hard already, see Garey and Johnson [85]. Moreover, the dual information gained from equation constraints is more difficult to exploit than that of cover or packing constraints. Therefore, we would like to relax the master problem to a set covering formulation (that remains an NP-hard problem but can be solved much more easily in our case) by only requiring the pairings to be flown by one *or more* crew members, i.e., we relax (5.2) to $\sum_i x_i \geq 1$. Then, however, to compute a legal schedule, we need a repair mechanism that decides which crew member finally gets an overcovered pairing assigned.

Secondly, dual information is rather poor in the initial master iterations, and thus, convergences is bad. This is due to the fact that we use so called *dummy columns* in the initial master problem. They are needed to obtain a formulation that is guaranteed to contain a feasible solution. The first type of dummy columns covers exactly crew member i, the second exactly

one activity j, for all $i = 1, \ldots, m$ and $j = 1, \ldots, n$. That is, we allow empty rosters and unassigned activities in intermediate steps. By setting the costs for choosing a dummy column to an arbitrary high value, we make sure that they only become part of an optimal solution if the original master problem was infeasible.

Although this procedure works, to achieve meaningful dual information of the master problem, the solution should not be spoiled by dummy costs. Thus, it would be better to generate an initial set of rosters that contains an entire work partitioning schedule.

5.4 Integrating the Approaches

We present two ways of integrating both methods each one motivated by different problem cases. In the first problem case, the construction of a feasible schedule is difficult due to very strict rules called for by the airline company.

We observe that CGA eventually gets close to solutions of good quality, but minor inconsistencies delay it disproportionately long. We show that this can be overcome effectively by letting the CGA approach solve a relaxed (that is Set Covering) version of the problem and then handing possibly overcovered (and thus infeasible) solutions to the HTS approach for fixing.

In the second problem case, the rule set is not that strict. The CGA approach alone proceeds as expected. However, the initial time spent for driving dummy columns out of the basis is considerable. In this phase, dual values are not very meaningful, because penalties dominate the objective. We show how the HTS method can help attacking the problem.

The issue that arises is that of the general applicability of each hybrid method and the possibility of them being combined to one single meta-hybrid, which would be generally applicable. We address these issues in the end of Sec. 5.5.

5.4.1 First Way of Integration: Transforming a Set Covering into a Set Partitioning Solution

The first method is applied on case B. In this company, no pairing can be left unassigned. Moreover, there is a relatively large number of pairings with respect to the number of crew members (for example 959 pairings/65 crews on a typical monthly problem). These conditions make finding a feasible solution difficult for the CGA approach. On the other hand, the HTS approach is able to construct feasible solutions by using sophisticated search methods and heuristics tailored for the specific problem. However, after a short while no improving solutions can be found.

We overcome the problems of both methods by letting the CGA approach find *Set Covering* instead of Set Partitioning Solutions. That is, we relax the pairing partitioning constraints (5.2) by only requiring that every pairing is assigned to *at least* one crew member. The columns generated by the CGA approach are much more easily combinable to SCP solutions. Then, the conversion of SCP to SPP solutions is assigned to the HTS approach, which can resolve local conflicts efficiently by using sophisticated propagation algorithms (see Figure 5.2). An outline of the procedure is shown in Algorithm 8. Here, V is the set of all variables, $\mathcal{A}_X$ is a tuple of assignments $< v, x_v >$ of values x_v to variables v generated by approach X, $a(\mathcal{A}, v)$ is a function which returns the value of variable v in assignment $\mathcal{A}$, DEFAULTSVAR

Algorithm 8 Top level algorithm for the first method

1: $\mathcal{A}_{HTS} \leftarrow$ HTSOPTIMIZE(V, DEFAULTSVAR, DEFAULTSVAL)
2: **repeat**
3: $\mathcal{A}_{CGA} \leftarrow$ CGAOPTIMIZE
4: $(V_1, V_2, V_3) \leftarrow$ PARTITION($\mathcal{A}_{HTS}$, $\mathcal{A}_{CGA}$)
5: **for all** $v \in V_3$ **do**
6: $v \leftarrow a(\mathcal{A}_{CGA}, v)$
7: $\mathcal{A}_{HTS} \leftarrow$ HTSOPTIMIZE($V_1 \cup V_2$, REPAIRSVAR($V_1 \cup V_2$, $\mathcal{A}_{CGA}$, V_2), REPAIRSVAL($V_1 \cup V_2$, $\mathcal{A}_{CGA}$, V_2)))
8: $\mathcal{A}_{HTS} \leftarrow$ LNSOPTIMIZE(V, REPAIRSVAR(V, $\mathcal{A}_{CGA}$, $V_1 \cup V_2 \cup V_3$), REPAIRSVAL(V, $\mathcal{A}_{CGA}$, $V_1 \cup V_2 \cup V_3$), $\mathcal{A}_{HTS}$)
9: **until** (stopping condition)

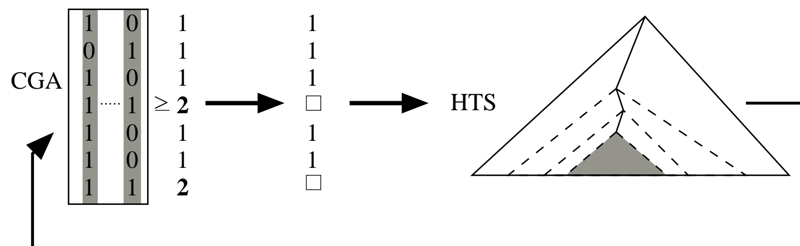


Figure 5.2: *Schematic view on the first way of integrating column generation and heuristic tree search via a set covering approach on the one, and a repair method on the other side. Column from overcovered rows are deassigned, and the remaining information is used to guide the search in the HTS (gray triangle). If the solution cannot be repaired, backtracking allows to deassign more (dashed lines).*

and DEFAULTSVAL are the variable and value selection functions normally used by the HTS approach respectively, REPAIRSVAR and REPAIRSVAL are the corresponding heuristics used for repairing Set Covering solutions and HTSOPTIMIZE, and CGAOPTIMIZE are the HTS and CGA optimization functions. PARTITION is a function which will be explained shortly. LNSOPTIMIZE performs optimization using the LNS method. The time span of the entire schedule is divided into successive time windows. All activities within such a window form a search subspace.

We now explain this algorithm in greater detail. In the first line, one or more initial solutions are found by the HTS approach. This initialization step provides the algorithm with a set of columns, which can be combined to feasible solutions. Not much time is devoted to this phase. The variable and value selection heuristics that would normally be used by the HTS approach are applied here. Any of the methods presented in the previous sections can be plugged in. However, we found mEDS to perform best in our case. The columns constituting these solutions are handed to the CGA approach for optimization in Line 3. The solution produced in this step is correct except for the fact that some pairings are assigned to more than one crew member, which is not legal.

The next task is to use the information found in $\mathcal{A}_{CGA}$ to construct a feasible solution. Let V_1 be the set of variables which correspond to overcovered pairings. One optimistic approach would be to assign the values of the assignment $\mathcal{A}_{CGA}$ to all the variables in $V \setminus V_1$ and let the HTS approach perform a search in the space of the variables in V_1. This, however, could lead

to a failure, since it is not known that the partial solution obtained is expandable to a feasible solution.

Algorithm 9 Heuristics for the first method

REPAIRSVAR($S, \mathcal{A}, V$)

1: $v \leftarrow$ NIL
2: **for all** unbound variables $v \in V$ **do**
3:    **if** ($a(\mathcal{A}, v) \in D_v$) **then**
4:       **return** v
5: **return** DEFAULTSVAR(S)

REPAIRSVAL($S, \mathcal{A}, V, v$)

1: **if** ($v \in V$ and $a(\mathcal{A}, v) \in D_v$) **then**
2:    **return** $a(\mathcal{A}, v)$
3: **else**
4:    **return** DEFAULTSVAL(S, v)

For some scheduling problems, such as the vehicle routing problem with time windows (see e.g. [184]), a partial solution can easily be extended by removing entries for overcovered rows from all but one of the corresponding columns. In our case, though, this approach is not generic enough, as certain rules may cause the resulting rosters to be infeasible. For example, a minimum flight time rule might be violated if a pairing is removed from an otherwise feasible roster. We say that such a rule destroys the *legal subroster property* of a rule set.

We can distinguish three subsets of variables in V: The set V_1 that consists of variables that correspond to overcovered pairings in $\mathcal{A}_{CGA}$, the set V_2 that consists of variables which have different values in $\mathcal{A}_{CGA}$ and $\mathcal{A}_{HTS}$, and the set V_3 which corresponds to variables having the same value in both assignments.

Function PARTITION partitions V in exactly this manner. Assignments of variables in V_3 are known to be expandable to a full solution, since one has already been found. Thus, because there is no information which suggests the contrary, they are realized as soon as possible in each iteration. Assignments in set V_2 may be considered as almost certain. However, they should be realized in a manner that allows backtracking. These issues are handled in Line 7 with respect to the search method and the heuristics used. CGA does not provide meaningful information for variables in V_1, so HTS performs the search for assignments to these variables using the default heuristics.

The variable and value selection functions are modified as shown in Algorithm 9. There, the variables that will be taken into account are variables in S. V is a subset of S for which assignments exist in $\mathcal{A}$. For example, when the variable selection rule is invoked in Line 7 of Algorithm 8, S is $V_1 \cup V_2$, V is V_2 and $\mathcal{A}$ is $\mathcal{A}_{CGA}$. In this case, the variable to be assigned next is any variable in V_2 for which its suggested value exists in its domain. In other words, all possible assignments in $\mathcal{A}_{CGA}$ are realized as soon as possible, in accordance to the intuitive belief that they would most probably lead to an area containing improving solutions. If this is not possible, then a variable in V_1 is selected, and the default heuristic is used.

Whenever possible, the value selection heuristic assigns the value suggested by CGA to each variable. Two important details are worth noting:

1. The variable selection heuristic is consulted *every time* a new assignment has to be made in

the HTS search. That is, if a variable v is selected (because $a(\mathcal{A}_{CGA}, v) \in D_v$) and then, for any reason, the search backtracks beyond that point (removing $a(\mathcal{A}_{CGA}, v)$ from D_v), then another variable might be selected instead of v. Like this, *assignments* and not just variables are *dynamically ordered* throughout the search process in such a way that those decisions contained in $\mathcal{A}_{CGA}$ will always be taken as early as possible.
2. Discrepancy-based search methods are used to express the belief that the assignments in $\mathcal{A}_{CGA}$ are probably good ones. That is, we try to stick to the decisions made by CGA, and we would like to make only few deviations. In our implementation, this issue is handled by using a variant of the LDS search method. In the original LDS proposal, based on the assumption that heuristic decisions are less accurate high in the search tree, early decisions are reconsidered first. But in our case, the assignments for variables in V_2 are realized in the beginning, thus the contrary holds. Therefore, we prefer to use mEDS in this phase, too.

We should also note that the function HTSOPTIMIZE in Line 1 of Algorithm 8 might or might not use LNS. That also holds for Line 8, where LNS is mentioned explicitly. That choice should be tuned towards the specific case. LNS as a stand-alone method is not preferred due to the fact that it is likely to get stuck in a local optimum, soon. However, for our purposes the most reasonable choice would be to use LNS after finding only one solution with a global tree search method. The local optimum will not be a problem, since the main optimization steps will follow, and much time will be gained. In any case, the user should use the method that provides a relatively good solution in the shortest time possible. We show the effect of such a choice in our experimental results. We also use it in Line 8 to overcome a problem that might arise when bounding the values of the variables in V_3: Variables in V_3 belong to assignments which have the same values in both $\mathcal{A}_{HTS}$ and $\mathcal{A}_{CGA}$. And they are bound to their values as proposed by CGA to explore promising regions of the search space. However, this might not be true in all cases. Thus, using LNS on $V_1 \cup V_2 \cup V_3$ instead of only $V_1 \cup V_2$ might help on reviewing some almost certain decisions that might not be as accurate. Of course, it is still a matter of choice to use or not to use LNS and if so, to use it on $V_1 \cup V_2 \cup V_3$ or only on $V_1 \cup V_2$. In our experiments, we used LNS with mEDS as the subtree search method.

5.4.2 Second Way of Integration: Generating Combinable Columns and Exploiting Dual Values

Algorithm 10 Top level algorithm for the second method

1: $\mathcal{C} \leftarrow$ HTSTREESEARCH(V, DEFAULTSVAR, DIVERSESVAL)
2: **repeat**
3: $\mathcal{A}$, ***duals*** $\leftarrow$ CGAOPTIMIZE($\mathcal{C}$)
4: HTSPOSTNRC(***duals***)
5: $\mathcal{C} \leftarrow$ HTSLNSTREESEARCH(V, MAXDUALVAR, MAXDUALVAL, $\mathcal{A}$)
6: **until** (stopping condition)

We propose a second integration strategy, that is applied on company C. In this case, the convergence of the CGA approach towards an optimal solution is assisted by HTS first by

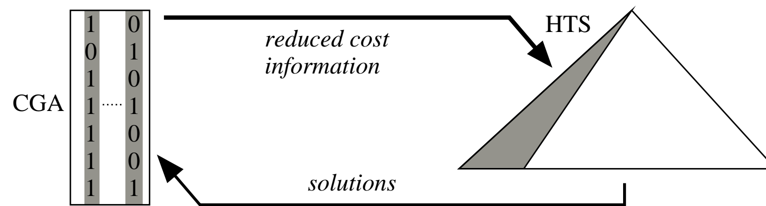


Figure 5.3: *Schematic view on the second way of integrating column generation and heuristic tree search: HTS provides an initial solutions, and thus, meaningful dual values. During optimization, CGA provides new dual values and requests solutions from HTS that respect an objective which is based on reduced costs.*

constructing a set of initial columns that are combinable to complete partitioning solutions in a startup phase, and secondly by constructing columns with negative reduced costs during the main optimization phase (Figure 5.3). These columns are guaranteed to be expandable to a feasible solution, since they are extracted from one. A top level sketch of this method appears in Algorithm 10. $\mathcal{C}$ is a set of rosters, $\mathcal{A}$ is an assignment, and ***duals*** are the dual values corresponding to this assignment (obtained by the CGA). HTSPOSTNRC posts a constraint on the number of rosters with negative reduced costs.

5.4.2.1 Startup Heuristic

In the CGA, columns are generated for each crew member sequentially. By using dual information, columns with negative reduced costs are generated. Thus, when the problem is non-degenerate, they lead to a decrease in the continuous relaxation of the master problem. Therefore, to find high quality rosters, "good" dual values are needed. Especially in the beginning, the information contained in the dual values is very poor. This is because usually no feasible solution is known at this point, and penalties stemming from dummy columns (that have to be introduced in the master problem to guarantee the existence of a solution) have a great impact on the dual values. We need to find a set of rosters that can legally be combined to form a set partitioning solution to the CAP. However, the column generator of the CGA is hardly able to produce such a solution, as it computes one roster at a time and is only indirectly aware of colliding pairings in different rosters.

Algorithm 11 Modified value selection heuristic for the second method

DIVERSESVAL(V, v, A, k)

1: $val \leftarrow$ NIL
2: **repeat**
3: $\quad val \leftarrow$ DEFAULTSVAL(S, v)
4: $\quad$**if** (the assignment $< v, val >$ appears more than k times in A) **then**
5: $\quad\quad$remove val from D_v
6: $\quad$**else**
7: $\quad\quad$**return** val
8: **until** ($val \neq$ NIL or D_v is empty)

HTS can help here. In an integrated approach, it is used to generate a bunch of complete feasible solutions in the beginning, thereby providing one column for each crew member with

every schedule found. Thus, a first set of columns that we know can be feasibly combined to a complete Set Partitioning Solution provides the CGA with the necessary "grip" to accelerate towards promising parts of the search space with respect to the "real" objective without disturbing penalties.

Line 1 of the Algorithm 10 realizes this idea. HTS searches for an initial number of solutions without performing optimization. The number of solutions to be found is a parameter that has to be tuned with respect to the time spent in this phase and the quality of the initial dual values.

Another parameter that has to be taken into concern is the diversity of the columns generated. It may be desirable to have many diverse rosters at hand that allow more and more profitable combinations in the master problem. One rule of thumb used in practice is that no crew-pairing assignment should appear more than a certain number of times in these columns. This restriction is taken into account by the slightly modified value selection heuristic DIVERSESVAL, which appears in Algorithm 11. It works exactly as the value selection heuristic that would normally be used, but it also records the assignments made and limits the number of times a crew member can be assigned to a pairing.

In Algorithm 11, A is the current set of solutions. Each time a solution is found by HTS, this solution is stored in A. Before assigning the value that would normally be selected by DEFAULTSVAL in Line 3, it is checked whether the assignment appears less than k times in A. Otherwise, the value is removed from v's domain. This heuristic, in coordination with Depth-Bounded Discrepancy Search, see Walsh [206], guarantees that columns will be adequately different from each other to make the CGA method even more efficient.

Especially for large data sets, many initial solutions are needed. To speed up their computation, we try to shrink the search space: First, only one solution is computed. Then, the LNS search procedure is applied to obtain solutions that satisfy the diversity conditions only in local areas of the search space. For example, time windows can be used to limit the search space.

5.4.2.2 Main Optimization Loop

As shown in Line 3 of Algorithm 10, CGA performs an optimization run taking the columns produced by HTS as input. It returns an assignment $\mathcal{A}$ as well as the corresponding dual values for the crews and pairings. The solution returned is feasible with respect to all the company's rules and regulations. Then, starting from this point, HTS performs a locally limited search for columns with negative reduced costs.

The constraint posted in Line 4 of the algorithm asserts that a certain number of the columns corresponding to each solution found will have negative reduced costs. This number is defined empirically. Finding a schedule that consists of columns with negative reduced costs only is rather unlikely. On the other hand, producing only few such columns is a wasted effort. Our experiments showed that schedules consisting of 30% columns with associated negative reduced costs can be achieved for our test set. But that does not mean that 70% of the columns produced are garbage! Instead, those columns guarantee that all newly generated columns can be extended to a feasible solution. Thus, the additional columns produced are important with respect to integer feasibility, whereas the columns with negative reduced costs reflect our search for improving solutions with respect to a linear objective.

Line 5 performs an LNS search with few deviations regarding the solution provided by

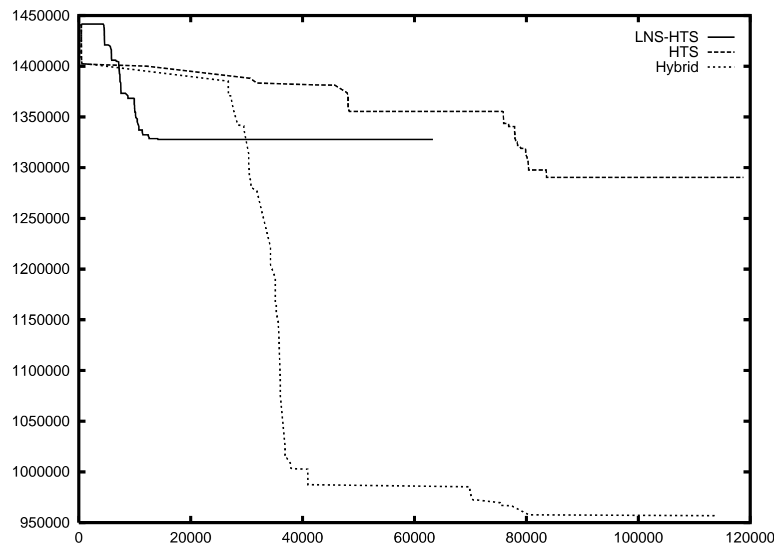


Figure 5.4: *Data set with 65 crew members and 959 pairings. Column generation fails to find a solution within 120 000 sec.*

CGA. The pairing with the maximum dual is assigned to the crew with the maximum dual as long as this crew member's reduced cost is not guaranteed to be negative already. Again our search method of choice is mEDS.

5.5 Numerical Results

To demonstrate the superiority of combined approaches integrating CP and OR techniques, we applied the hybrid algorithms as presented above to real-world crew assignment problems (see Sec. 5.1). Both the CGA and the HTS are prototype implementations only. Within a research project, it is not realistic to develop implementations for the CAP that could compete with the best industrial codes regarding overall speed, because those codes were produced during hundreds of person-years. Therefore, we just try to highlight the gain in efficiency that can be obtained when combining methods from OR and CP.

We applied each method integrating HTS and CGA on the airline cases that motivated their development. All algorithms were implemented in C++ on top of Ilog software [114, 116]. The first integration strategy was applied on two monthly data sets from company B. Experiments for this case were performed on a 640 MB, 296 MHz SUN UltraSPARC-II, with a time limit of 120 000 seconds.[2] The efficiency of our algorithm improves on the production system which company B currently uses.

Figure 5.4 is a cost (i.e., dissatisfaction) versus time graph showing the performance of the hybrid and the pure HTS methods applied on a monthly data set containing 959 pairings and 65 crew members. The problem is a minimization problem. The curve marked "LNS-HTS" corresponds to a hasty strategy in which, after one solution is obtained, LNS is used to achieve some good solutions quickly. The "HTS" curve shows a more mature strategy, where the search finds several good solutions before LNS is applied to locally optimize them. The curve

[2]Curves stopping before this threshold indicate that no better solution was found from the moment corresponding to the end of the curve until the time limit has been reached.

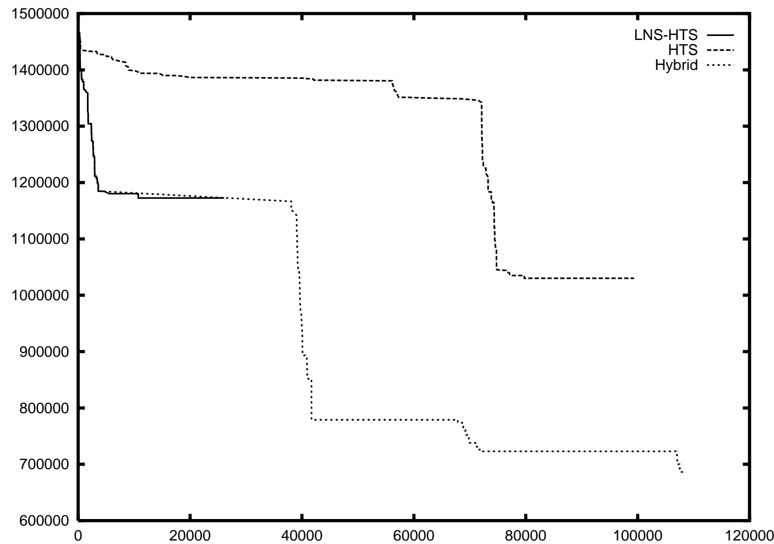


Figure 5.5: *Data set with 50 crew members and 766 pairings. Column generation fails to find a solution within 120 000 sec.*

marked “hybrid” shows the performance of the hybrid approach, which clearly outperforms both. Interestingly, the pure CGA cannot detect any feasible solution at all. Within 120 000 seconds, it was not able to remove all dummy columns from the solution, i.e., the original master problem without dummy columns still is infeasible.

In these specific experiments, for exhibition purposes only, we call the HTS strategy in Line 1 of Algorithm 8 to show that it would have the best performance regardless of the startup phase. That is the reason why “LNS-HTS” outperforms “hybrid” in the beginning. Of course, we repeat that a reasonable choice for the startup phase of Algorithm 8 would be a strategy more like “LNS-HTS”. This strategy is used in the experiments of Figure 5.5, which shows the performance of the same methods on another company B monthly data set containing 766 pairings and 50 crew members.

The following set of experiments is carried out to investigate the second way of integration. Experiments for the company B data were performed on a 128 MB, 143 MHz SUN UltraSPARC, with a time limit of 20 000 or 70 000 seconds depending on the problem size. Figure 5.6 shows the costs versus time plot for CGA, HTS and the second, so called, *consolidated approach* for a data set with 7 crew members and 129 pairings. Initially, HTS generates a solution and passes it over to CGA, which performs one optimization iteration. The resulting schedule is passed back to HTS, which rebuilds it and then locally searches for solutions containing as many rosters with negative reduced costs as possible. The PostNrc constraint guarantees that an adequate number of such rosters will be returned. These rosters are then passed back to the CGA, and the process is repeated.

The same approach is used on a bigger problem instance, as shown in Figure 5.7. The plots depict the expected behavior of CGA and HTS. CGA steadily optimizes the objective, but the quality of the initial solution is poor. Moreover, the time needed to find a first solution grows with the problem size. On the other hand, HTS finds relatively good solutions quickly by using heuristic information, but soon gets stuck. The consolidated approach benefits from both approaches: it finds good solutions quickly because of HTS and then steadily continues to refine the solutions due to the help of CGA.

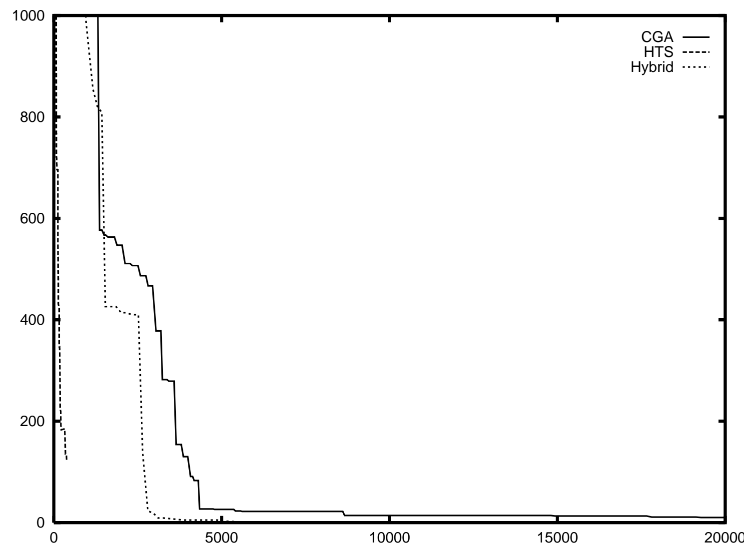


Figure 5.6: *Data set with 7 crew members and 129 pairings.*

It can also be seen that the integrated approach is slower than HTS early in the experiments. During that time, the hybrid approach is using the HTS module to create an initial set of columns according to the startup heuristic. The reason why HTS is slower in the consolidated case is that the goal is not to find better and better solutions, since the main optimization burden lies on the CGA side. Instead, HTS rather tries to find diverse rosters, which help CGA to find better solutions in the following.

The experiments regarding the second way of integration show that it is always useful to assign the task of finding a set of initial solutions to the HTS approach. The best number of solutions computed initially depends on the rule set as well as on the characteristics of the instance. Assigning the main optimization burden to CGA is the default choice, as it views the problem globally taking into account all variables and constraints at a time. If minor local adjustments can lead to quality improvements, then having HTS perform LNS searches throughout the process is cost effective. Furthermore, if the column generation process gets stuck, i.e., if a significant number of columns with negative reduced costs proves not be combinable to an IP solution, then having HTS generate solutions incorporating columns with negative reduced costs is cost effective, too.

5.6 Combining Both Hybrid Methods

Experimental results clearly show that each hybrid approach is successful on the airline case on which it is applied in our experiments. The question that arises is whether the two hybrids can generally be combined or not.

We believe that orthogonality generally holds: A meta-hybrid could start off by having the HTS construct a set of solutions out of which diverse and feasibly combinable columns can be extracted. Then, the CGA approach can be used to improve on a relaxed version of the problem, which is repaired by the HTS approach.

We found that whether or not the use of one of the hybrid approaches we presented can speed up the computation of a good solution is problem dependent:

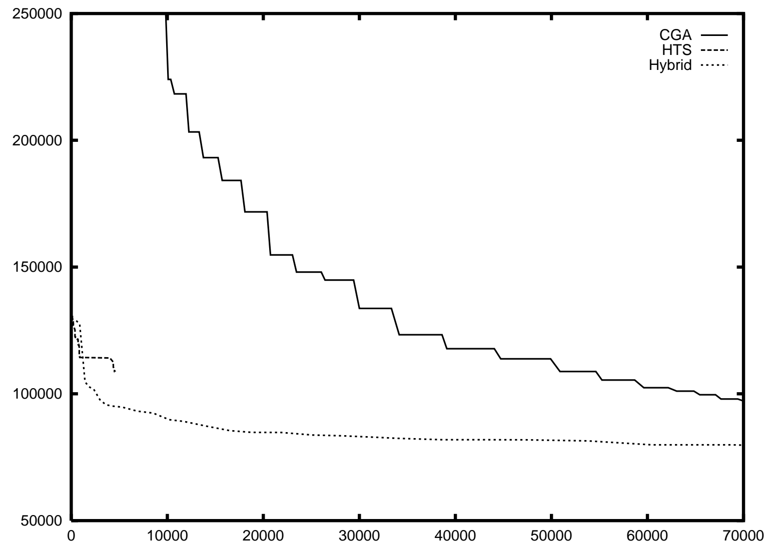


Figure 5.7: *Data set with 30 crew members and 279 pairings.*

- Of course, the first hybrid can only be applied profitably, if the master problem is hard enough to justify the use of a relaxation that must be repaired at some point. Regarding airline case C, this precondition is not fulfilled, which is why we cannot apply hybrid 1 on this case.
- Using initial solutions provided by the HTS approach, in order to speed up the starting phase of CGA, only pays off when the CGA approach alone has difficulties in driving dummy columns out of the basis or spends too much time on this phase of the process. This is not given in airline case B, which causes that hybrid 2 cannot be used profitably here.

We conclude that generally the two hybrids can be combined, but the usefulness of a meta-hybrid is problem dependent. And its tuning heavily relies on inherent problem properties, which might not be known a priori.

5.7 Conclusions

For the CAP as an example, we have shown how CP and OR techniques can help each other to overcome their fundamental weak points. We believe that the ideas discussed in this chapter can be generalized for other problems as well, especially in connection with (CP based) column generation. We presented results on large scale real world CAP data, which show clearly visible improvements in performance of the hybrid approaches compared to the solitary methods.

While OR methods view a problem globally and show a good ability to detect promising regions of the search space, CP methods can efficiently handle feasibility problems and are well suited to resolve local conflicts. The first way of integration tries to combine these advantages. It uses the CP based Column Generation approach (CGA) to compute cost efficient yet relaxed solutions to the problem, and then resolves conflicts of overcovered pairings by

applying a heuristic CP tree search (HTS). The synergy effects are particularly visible if a lot of work has to be grouped in relatively few partitions. Then, column generation alone often fails to generate combinable rosters, and the use of HTS as a repairing module helps a lot to increase the overall performance.

The second way of integration that we introduced concerns the use of dual values. We showed how column generation approaches can profit from CP via the computation of diverse combinable initial columns. On the other hand, the use of dual information in a CP based heuristic tree search has shown to be very efficient. It allows to laden the optimization burden on the OR part and away from CP, which then can focus on what it was designed for originally, namely to solve constraint satisfaction problems.

6

Home Health Care

6.1 Introduction

Home health care (HHC), i.e. visiting and nursing patients in their homes, is a growing sector in the medical service business. More and more private companies are now working in this area. As the nursing companies get larger, the problem of how to schedule the nursing staff arises. The challenge of this problem is to combine aspects of vehicle routing and staff rostering. Both are well known combinatorial optimization problems, and good algorithms for each of these two problems are known (see e.g. Toth and Vigo [198] for vehicle routing, and e.g. Barnhart et al. [21], Caprara et al. [45], Fahle et al. [73] for staff rostering in different application fields). To obtain good solutions, however, it is crucial to solve the nurse scheduling problem as a whole due to the high inter-dependencies of optimized routes and rostering constraints. Additionally, soft constraints and preferences require the use of specially designed algorithms.

Rostering constraints include hard ones like qualification requirements or work time limitations and soft ones. Soft constraints are especially difficult to handle, but have to be considered for applicable schedules. Typical examples of soft constraints are:

- patients prefer certain time intervals for being served,
- the right “chemistry” between patients and staff has to be ensured,
- patients do not like frequent changes of nursing staff,
- staff satisfaction concerning e.g. work load and work time should be maximized.

The vehicle routing aspect of the problem has to take time windows, travel times and distances, and inhomogeneous fleets (bicycles, public transport, cars) into account. In Figure 6.1 we present a more detailed view of constraints relevant to the HHC problem.

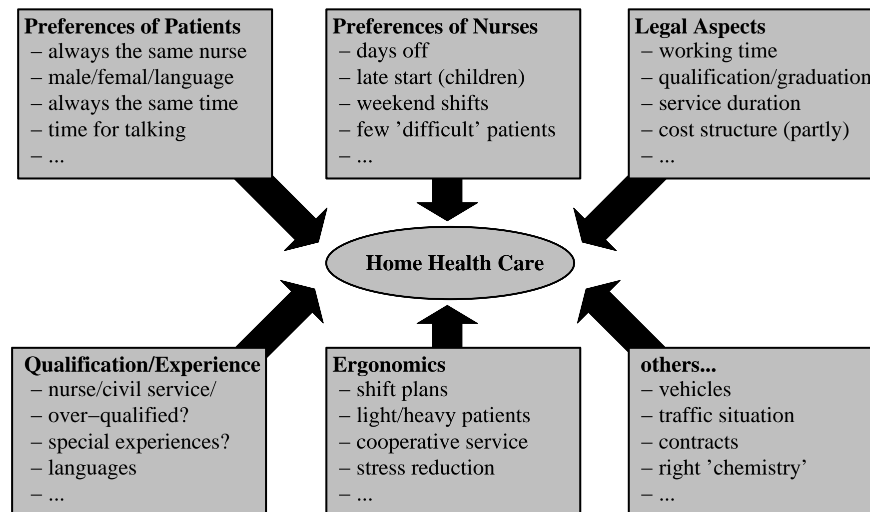


Figure 6.1: *An overview of constraints relevant to the home health care problem.*

The solutions with minimal costs and maximal patients/staff satisfaction are of most interest for companies. Costs in this context may reflect expenses for fuel, etc. or costs of the staff. In countries, where health services are partially included in the social system, a company typically gets a fixed amount of money for a given service—independent of the income of the nurse providing the service. Hence, companies prefer schedules where highly qualified staff are only assigned to patients that need that high qualification. Simpler jobs should be assigned to less skilled, and hence less expensive employees.

Respecting preferences is the second important parameter to be considered in HHC rostering. Patients will simply change to a different health care company if their wishes are not met. For the staff, considering their preferences increases motivation, which on the one hand impacts on the patients, and on the other hand helps to deal with many stressful situations (death, terminal illnesses, late jobs, etc.).

6.1.1 Specific Requirements

The project PARPAP[1] brings together end users (home health care companies), universities (computer science and ergonomics) and partners from software industries to develop an optimization and planning tool for the highly constrained problem of staff rostering and routing in the home health care industry. The aim of the project is to model the problem accurately and to develop suitable algorithms for finding good rosters respecting all hard and soft constraints mentioned above. Since rules and regulations for home health care change frequently, and company philosophies differ also, there is a need for flexibility in both, modeling capability and algorithmic approaches.

Apart from the problem's constraints we experienced that people in the health care business keep a critical distance from fully automated decision or planning systems. One idea of PARPAP therefore is to build an optimization system that presents several possible solutions and allows the dispatcher to interact with the system in order to select one plan, or to re-calculate parts of the solution.

[1] see footnote 3 on page 7

Another real-world requirement in the context of PARPAP is runtime limitation. In the application scenario, planners like to have a good solution after at most 10–15 minutes runtime. From the experiments presented in previous sections this obviously excludes column generation approaches for the HHCP. Instead, we will apply Tabu Search, Simulated Annealing or CP, respectively, to assign staff to jobs. For optimizing an individual work-plan, we will use a hybrid linear and constraint programming module.

All in all, the software prototype consists of a database and a GUI component, as well as a planning and optimization module. This chapter, however, is only dedicated to the optimization modules.

6.1.2 Literature Review

To our knowledge, there are only a few publications on the topic of optimization and scheduling in home health care: Cheng and Rich describe a combined mixed-integer programming (MIP) and heuristics approach [52]. Numerical results for up-to 4 nurses and 10 patients are presented. In Begur et al. [27] a decision support system that is based on simple scheduling heuristics is proposed.

Two related topics, however, have attracted more researchers: Planning systems for hospitals model some aspects that are also needed for home health care. We only mention Abdennadher and Schlenker [2], Burke et al. [42], Cheng et al. [51], Mason and Smith [148] as examples. Most of these use constraint programming techniques in order to model and solve the nurse rostering problem. Vehicle Routing with Time Windows (see e.g. Fisher [80], Gendreau et al. [86]) reflects the mobility aspect of the problem, but ignores any further restriction.

In Sec. 6.2 we will describe how we model the home health care problem. For this presentation, a more compact model than the one in the industrial prototype will be used. The simplified model still reflects the key characteristics of the original problem, but allows us to ignore certain technical details when presenting the algorithms. In brief we present extensions of that model in Sec. 6.2.1.4. Section 6.3 is dedicated to the heuristics developed for the problem, and we will show how to combine these methods in a powerful hybrid approach in Sec. 6.4. Section 6.5 presents numerical results of the various methods, and finally we conclude.

6.2 A Mathematical Model for the Home Health Care Problem

Staff rostering describes the process of assigning staff to tasks. In HHC, the staff consists of employers with various skills, and the tasks are the services to be provided to the patients, training, etc. Timing rules, qualification rules, relationship rules, and structural rules, as well as routing information dictate the way, a "good" roster should look. The "cost" of a roster depends on real costs for wages and transportation, as well as on artificial costs introduced as penalty terms for modeling certain characteristics and preferences.

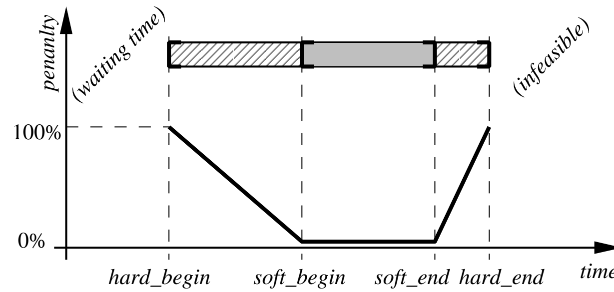


Figure 6.2: *Penalty concept for Time Windows. Arriving before* **hard_begin** *produces waiting time, arriving after* **hard_end** *is infeasible. Penalties proportional to earliness or lateness are used for arrivals within the hard, but before or after the soft time window.*

6.2.1 Parameters and Notation

We are given N nurses $\mathcal{N} = \{1, \ldots, N\}$, P patients $\mathcal{P} = \{1, \ldots, P\}$, and a set of J jobs $\mathcal{J} = \{1, \ldots, J\}$. Jobs represent either a certain service to be provided to a certain patient, or a task to be performed by a nurse during her working hours (like training, breaks, emergency service, etc.). Thus, there is a fixed location for each of these jobs.

For each patient $p \in \mathcal{P}$ there is a subset $\mathcal{J}_p \subseteq \mathcal{J}$ such that these jobs involve patient p, and it holds: $\bigcup_{p \in \mathcal{P}} \mathcal{J}_p \subseteq \mathcal{J}$, and $\mathcal{J}_p \cap \mathcal{J}_q \neq \emptyset \iff p = q$. Since jobs are patient related, we can always identify the patient corresponding to a job. In the following, we will only speak of jobs, and by *preferences of a job* we refer to the preferences of the patient corresponding to the job.

A *time window* represents the time interval in which a job has to be started, or describes the working time interval of a nurse. We distinguish between *soft* time windows and *hard* time windows. Whereas a soft time window is only a preference, which we may violate at the expense of penalty costs, any hard time window has to be met. Waiting time before a hard time window is allowed, beginning service after the hard time window leads to an infeasible roster. The following functions represent the hard and soft time windows of nurses and jobs, respectively:

- $\mathrm{hb}_{\mathcal{J}} : \mathcal{J} \to \mathbb{Q}$, $\mathrm{he}_{\mathcal{J}} : \mathcal{J} \to \mathbb{Q}$, describe start and end, resp. of a job's *hard* time window.
- $\mathrm{sb}_{\mathcal{J}} : \mathcal{J} \to \mathbb{Q}$, $\mathrm{se}_{\mathcal{J}} : \mathcal{J} \to \mathbb{Q}$, describe start and end, resp. of a job's *soft* time window.
- $\mathrm{d}_{\mathcal{J}} : \mathcal{J} \to \mathbb{Q}$ is the duration of the job, i.e. the time needed to complete a job.
- $\mathrm{hb}_{\mathcal{N}} : \mathcal{N} \to \mathbb{Q}$, $\mathrm{he}_{\mathcal{N}} : \mathcal{N} \to \mathbb{Q}$, $\mathrm{sb}_{\mathcal{N}} : \mathcal{N} \to \mathbb{Q}$, $\mathrm{se}_{\mathcal{N}} : \mathcal{N} \to \mathbb{Q}$, represent the start and end of the *hard* and *soft* time window of a nurse.
- $\mathrm{min_time}_{\mathcal{N}} : \mathcal{N} \to \mathbb{Q}$ and $\mathrm{max_time}_{\mathcal{N}} : \mathcal{N} \to \mathbb{Q}$ give minimal and maximal working time of a nurse.

Obviously, for a job $j \in \mathcal{J}$ we have $\mathrm{hb}_{\mathcal{J}}(j) \leq \mathrm{sb}_{\mathcal{J}}(j) \leq \mathrm{se}_{\mathcal{J}}(j) \leq \mathrm{he}_{\mathcal{J}}(j)$. The same holds for time windows of nurses. Figure 6.2 shows an example for these time windows.

Qualifications are the skills possessed by a nurse, or required by a job, respectively. As we will see later, the concept of qualifications offers a rather flexible tool for modeling various characteristics of the HHC problem.

Let S be a set of all qualifications:

- $\text{quali}_{\mathcal{J}} : \mathcal{J} \to 2^S$ are the required qualifications for a job.
- $\text{quali}_{\mathcal{N}} : \mathcal{N} \to 2^S$ are the qualifications possessed by a nurse.

Hard qualifications are those that are vitally required for the job and include e.g. graduation. *Soft constraints* embrace preferences of nurses and patients, and may be ignored at the expense of penalty costs. We model preferences of patients for certain nurses, preferences of nurses for certain patients, experiences for certain jobs, and factors that guide a fair distribution of difficult jobs over all nurses (over some time period). Also, we allow soft qualifications and requests for those. In the compact model, all these parameters are accumulated into one function, which gives the soft constraint penalty value for assigning nurse n to job j:

- $\text{sc} : \mathcal{N} \times \mathcal{J} \to [0 \dots 1]$.

The last group of functions provides geographical information. The *routing* aspects consider travel time between two jobs j, j'. Since each job has a fixed location, we use the travel time between these locations:

- $\text{tr_time} : \mathcal{J} \times \mathcal{J} \to \mathbb{Q}$, travel time between the locations of two jobs.

(In the industrial prototype also travel distances are used.)

6.2.1.1 Core Optimization Problem

In the HHC problem we are looking for an assignment of job schedules to nurses, such that all jobs are taken care of, all hard constraints are respected, only few soft constraints are violated, and such that the overall cost for that assignment is minimal and the number of preferences satisfied is maximal. To be more specific, we formulate the following:

A sequence $R = [(j_1, t_1), \dots, (j_k, t_k)]$, $j_l \in \mathcal{J}$, $t_l \in \mathbb{Q}$, $l = 1 \dots k$ is called a *roster* and contains k jobs and a starting time t_l for each job j_l. We assume that the sequence is ordered in increasing times, that is $t_l < t_{l+1}$ for $l = 1 \dots k - 1$.

We need to find a *solution* $S = \{R^{(1)}, \dots, R^{(N)}\}$ consisting of N rosters, where $R^{(n)} = [(j_1^{(n)}, t_1^{(n)}), \dots, (j_{k_n}^{(n)}, t_{k_n}^{(n)})]$ is the roster for nurse n, such that:

$$\bigcup_{i=1}^{N} \bigcup_{l=1}^{k_i} \{j_l^{(i)}\} = \mathcal{J} \tag{6.1}$$

$$\forall 1 \le i, i' \le N,\ 1 \le l \le k_i,\ 1 \le l' \le k_{i'} : \quad (j_l^{(i)} = j_{l'}^{(i')}) \ \Rightarrow\ (i = i' \ \wedge\ l = l') \tag{6.2}$$

$$\text{hb}_{\mathcal{J}}(j_l^{(i)}) \le t_l^{(i)} \le \text{he}_{\mathcal{J}}(j_l^{(i)}), \quad i = 1 \dots N,\ l = 1 \dots k_i \tag{6.3}$$

$$t_l^{(i)} + \text{d}_{\mathcal{J}}(j_l^{(i)}) + \text{tr_time}(j_l^{(i)}, j_{l+1}^{(i)}) \le t_{l+1}^{(i)}, \quad i = 1 \dots N,\ l = 1 \dots k_i - 1 \tag{6.4}$$

$$\text{min_time}_{\mathcal{N}}(i) \leq t_{k_i}^{(i)} + \mathrm{d}_{\mathcal{J}}(j_{k_i}^{(i)}) - t_1^{(i)}, \quad i = 1 \dots N \tag{6.5}$$

$$\text{max_time}_{\mathcal{N}}(i) \geq t_{k_i}^{(i)} + \mathrm{d}_{\mathcal{J}}(j_{k_i}^{(i)}) - t_1^{(i)}, \quad i = 1 \dots N \tag{6.6}$$

$$\mathrm{hb}_{\mathcal{N}}(i) \leq t_1^{(i)} \quad \text{and} \quad t_{k_i}^{(i)} + \mathrm{d}_{\mathcal{J}}(j_{k_i}^{(i)}) \leq \mathrm{he}_{\mathcal{N}}(i). \tag{6.7}$$

$$\mathrm{quali}_{\mathcal{J}}(j_l^{(i)}) \subseteq \mathrm{quali}_{\mathcal{N}}(i), \quad i = 1 \dots N, \; l = 1 \dots k_i \tag{6.8}$$

These hard constraints can be explained as follows: We need to cover all jobs by our schedules (6.1), but none of them more than once (6.2). Any starting point for a job has to respect the hard time window (6.3). In (6.4) we require enough time to provide the service and to travel to the next job before the next job starts. (6.5) and (6.6) define lower and upper bounds on the working time, and (6.7) ensures that no job is carried out outside the work time interval of the nurse. Finally, we insist on having all hard qualifications of a job covered by the assigned nurse (6.8).

6.2.1.2 Cost Function

In order to model the cost function, we have to define our measure for violating soft constraints. Any violation of a soft time window will be penalized by a factor proportional to the earliness or lateness. For all jobs $j \in \mathcal{J}$ let t_j be the time assigned to job j. Then

$$\mathrm{early}_{\mathcal{J}}(j) := \begin{cases} 0, & \mathrm{sb}_{\mathcal{J}}(j) = \mathrm{hb}_{\mathcal{J}}(j) \\ \frac{\mathrm{sb}_{\mathcal{J}}(j) - t_j}{\mathrm{sb}_{\mathcal{J}}(j) - \mathrm{hb}_{\mathcal{J}}(j)}, & \textit{else} \end{cases} \tag{6.9}$$

$$\mathrm{late}_{\mathcal{J}}(j) := \begin{cases} 0, & \mathrm{he}_{\mathcal{J}}(j) = \mathrm{se}_{\mathcal{J}}(j) \\ \frac{t_j - \mathrm{se}_{\mathcal{J}}(j)}{\mathrm{he}_{\mathcal{J}}(j) - \mathrm{se}_{\mathcal{J}}(j)}, & \textit{else} \end{cases} \tag{6.10}$$

$$\mathrm{p}_{\mathcal{J}}(j) = \max\{\mathrm{early}_{\mathcal{J}}(j), \mathrm{late}_{\mathcal{J}}(j), 0\} \tag{6.11}$$

Since t_j is within the soft time window ($\Rightarrow \mathrm{early}_{\mathcal{J}}(j) \leq 0$ and $\mathrm{late}_{\mathcal{J}}(j) \leq 0$), before the soft time window ($\Rightarrow \mathrm{early}_{\mathcal{J}}(j) > 0$ and $\mathrm{late}_{\mathcal{J}}(j) \leq 0$), or after it ($\Rightarrow \mathrm{early}_{\mathcal{J}}(j) \leq 0$ and $\mathrm{late}_{\mathcal{J}}(j) > 0$), choosing $\mathrm{p}_{\mathcal{J}}(j)$ as in (6.11) gives the correct penalty (see Fig. 6.2).

Violating nurses' soft time window is treated similarly, besides taking care of both, the soft time window's beginning and end. Let k_n denote the last job assigned to nurse n, $n = 1 \dots N$:

$$\mathrm{early}_{\mathcal{N}}(n) := \begin{cases} 0, & \mathrm{sb}_{\mathcal{N}}(n) = \mathrm{hb}_{\mathcal{N}}(n) \\ \frac{\mathrm{sb}_{\mathcal{N}}(n) - t_1}{\mathrm{sb}_{\mathcal{N}}(n) - \mathrm{hb}_{\mathcal{N}}(n)}, & \textit{else} \end{cases} \tag{6.12}$$

$$\mathrm{late}_{\mathcal{N}}(n) := \begin{cases} 0, & \mathrm{he}_{\mathcal{N}}(n) = \mathrm{se}_{\mathcal{N}}(n) \\ \frac{t_{k_n} + \mathrm{d}_{\mathcal{J}}(j_{k_n}) - \mathrm{se}_{\mathcal{N}}(n)}{\mathrm{he}_{\mathcal{N}}(n) - \mathrm{se}_{\mathcal{N}}(n)}, & \textit{else} \end{cases} \tag{6.13}$$

$$\mathrm{p}_{\mathcal{N}}(n) = \frac{1}{2}\left(\max\{\mathrm{early}_{\mathcal{N}}(n), 0\} + \max\{\mathrm{late}_{\mathcal{N}}(n), 0\}\right) \tag{6.14}$$

Let $R = [(j_1, t_1), \ldots, (j_k, t_k)]$, $j_l \in \mathcal{J}$, $t_l \in \mathbb{Q}$, $l = 1 \ldots k$, be a roster assigned to nurse n. Any violated soft qualification is penalized by adding extra costs stemming from assigning these jobs to nurse n, and normalizing them by the number of jobs assigned:

$$\mathrm{p_{sc}}(n) := \frac{1}{k} \cdot \sum_{l=1}^{k} \mathrm{sc}(n, j_k) \quad n = 1 \ldots N \tag{6.15}$$

Having defined a solution of the HHC problem and knowing the measure for soft constraints, we are now able to construct the objective function. In order to reflect both, the routing and the rostering aspects, we combine them into a weighted sum of the total travel time needed for the schedule and the sum of all penalties:

$$\begin{aligned}
\text{minimize} \quad & \mathrm{obj}(R^{(1)}, \ldots, R^{(N)}), \qquad \text{where} \\
\mathrm{obj}(R^{(1)}, \ldots, R^{(N)}) = \; & \alpha_1 \cdot \frac{\sum_{i=1}^{n} \sum_{l=1}^{k_i - 1} \mathrm{tr_time}(j_l^{(i)}, j_{l+1}^{(i)}) - LB_{traveltime}}{UB_{traveltime} - LB_{traveltime} + \varepsilon} && (6.16) \\
& + \alpha_2 \cdot \frac{1}{N} \frac{\sum_{n \in \mathcal{N}} \left[t_{k_n}^{(n)} + \mathrm{d}_{\mathcal{J}}(j_{k_n}^{(n)}) - t_1^{(n)} \right] - LB_{worktime}}{UB_{worktime} - LB_{worktime} + \varepsilon} && (6.17) \\
& + \alpha_3 \cdot \frac{1}{J} \sum_{j \in \mathcal{J}} \mathrm{p}_{\mathcal{J}}(j) + \alpha_4 \cdot \frac{1}{N} \sum_{n \in \mathcal{N}} \mathrm{p}_{\mathcal{N}}(n) && (6.18) \\
& + \alpha_5 \cdot \frac{1}{N} \sum_{n \in \mathcal{N}} \mathrm{p_{sc}}(n) && (6.19) \\
\text{and} \quad & \alpha_1 + \cdots + \alpha_5 = 1, \alpha_i \geq 0 && (6.20)
\end{aligned}$$

Whereas (6.18) and (6.19) only sum up all penalties, and normalize these numbers, (6.16) and (6.17) are a slightly more complicated. In (6.16), the double sum accumulates the travel-times needed to travel between any two consecutive jobs for all schedules. We normalize that value by relating it to some upper and lower bounds for the total travel-time of a given instance. E.g. we can set $LB_{traveltime}$ equal to the sum of the $J - n$ smallest travel-times calculated between jobs, and accordingly, we use the $J - n$ largest travel-times for $UB_{traveltime}$. The ε ensures a valid fraction in case the lower and upper bound overlap. Similarly, (6.17) models the total working time of a nurse. We can set $UB_{worktime} = \sum_{n \in \mathcal{N}} \mathrm{max_time}_{\mathcal{N}}(n)$ and $LB_{worktime} = \sum_{n \in \mathcal{N}} \mathrm{min_time}_{\mathcal{N}}(n)$.

The model defined above can represent several NP-hard optimization problems. E.g. we get a multi-TSP (see Toth and Vigo [198]) with time windows, even if we ignore qualifications (6.8), (6.15), (6.19), work time limits (6.5), (6.6), and soft time windows (6.11), (6.18). Similar adaptations lead to multi-processor scheduling or set covering problems (see Garey and Johnson [85]).

6.2.1.3 Modeling Different Characteristics

Hard and soft time windows and especially the concept of qualifications allow the modeling of various different situations which occur in typical real-world situations. If patients prefer

to be only served by selected nurses, we can model this via soft constraints. A "qualification tag" can be used to distribute staff over several districts. Language requirements can also be modeled by qualifications. Also, different starting and ending locations can be described by introducing dummy jobs that can only be performed by a specific nurse. Thereby we can model starting or ending at the nurse's home, at a patient, or at the care company's office, respectively. Since the preferences of nurses are modeled analogously we have already covered a major part of the constraints presented in Fig. 6.1.

6.2.1.4 Extensions of the Model

In the extended model underlying the industrial prototype, we provide some more expressiveness. E.g. for all time windows, there are factors which adjust the penalty term, thus allowing more emphasis on "critical" patients or staff. Different experience is reflected by the fact that the time needed to serve a patient depends not only on the job, but also on the assigned nurse. Moreover, we distinguish between several classes of vehicles and use different routes, travel times and distances for these classes. Finally, the objective of the extended model includes additional terms for travel distance.

Some requirements cannot be formulated only by parameters. E.g. by only using parameters we cannot formulate: "No two consecutive jobs should require lifting a heavy patient". However, we can easily deal with such a requirement within our algorithmic framework and the extended model provides rules to efficiently handle these special constraints.[2]

6.3 Solving Home Health Care Problems

Our model defined above is a hybrid of a rostering model and a routing model. Good approaches were presented for both models in literature, and we will re-use some of the ideas presented previously to build our heuristics. However, it is not possible to use a two-stage approach that first generates feasible routes, and then assigns nurses to them (violation of hard/soft qualification and nurses' time window constraints). Also the vice versa approach — assigning jobs to nurses and then generating routes — will more than likely produce infeasible or disproportionally expensive routes. Therefore, only an integrated approach that considers time scheduling, rostering, and route planning simultaneously is appropriated for the HHC problem as a whole.

Our approach divides the model into two parts: (a) finding a partition of jobs to nurses, and (b) finding an optimal sequencing for each such partition. The latter one contains in our case a TSP with time windows and is therefore NP-hard.

We have three different approaches to (a): Initial heuristics that quickly generate an initial solution and two improvement heuristics which take a solution and try to improve it via local exchanges. All of these need information regarding a good sequence of jobs within a roster (part (b)), which we determine via a combined CP and LP approach (Section 6.3.2). As a first step, however, we try to reduce the data complexity of a given instance.

[2] Basically, one has to adapt lines 19,20 in Alg. 12. There, we only extend a roster if the structure is feasible according to the additional requirements.

6.3.1 Preprocessing

During the initial data preprocessing step, we determine which job/nurse pair is compatible with the hard qualification constraint (6.8) and time window constraint (6.7), and we store this information. Should there exists a job that can be done by only one nurse, we fix this assignment (without fixing the time at which this job has to be done). Also, we compute the soft constraint values that include all preference parameters. Time windows of jobs can then be shrunk to the earliest or latest time at which a nurse is available (6.7).

In the last step we determine precedences implied by the time windows. That is, for any two jobs we store if they have to be performed in parallel or in an induced order or if they are not related, respectively.

6.3.2 Sequencing a Roster

Sequencing consists of ordering the jobs and assigning starting times. Due to time windows constraints, in the HHC only few permutations correspond to feasible orderings. In our approach we enumerate those orderings by a CP approach, and we use an LP to find optimal start times with respect to the objective.

6.3.2.1 Optimal Start Times for a valid Ordering

Given a set $\{j_1, \ldots, j_k\}$ and a nurse n we need to find an optimal sequencing (i.e. an ordering of these jobs and an assignment of a start time to each of the jobs) such that the sum of travel times and penalties for not hitting the soft time window is optimal for nurse n. For a given ordering and a given nurse, we can easily calculate penalties by applying (6.9)–(6.11). Moreover, a simple LP can be used to optimize the starting times t_l of each job j_l, $l = 1 \ldots k$ (see (6.21)).

We explain LP (6.21) in more detail: For $l = 1 \ldots k$ variable t_l corresponds to the starting time of job l and x_l accounts for soft time window violations at job l. y_1, y_2 is the penalty for violating the nurse's soft time window and w accounts for working time. The first constraint class corresponds to (6.4), the three lines using x_l refer to (6.9)–(6.11), lines 5,6 limit the starting time to the hard time window (6.3). The nurse's soft time window ((6.12)–(6.14)) is respected by lines 7–9 and the corresponding hard time window (6.7) in lines 13,14. Finally, lines 10–12 correspond to the minimal and maximal work-time constraints (6.5) and (6.6), and to work time minimization. If we use the LP in a setting where sets of jobs are generated incrementally we have to replace (#) by $w \geq 0$ to ensure feasibility.

In the LP objective we use the weight factors α_2, α_3, and α_4 of the HHC problem. Worktime w is rated by the global bound on working time.

As stated before, for a given order of jobs the starting times found by the LP are optimal with respect to time window penalties and working time.

6.3.2.2 Generating valid Orderings

The generation of all feasible orderings for jobs $\{j_1, \ldots, j_k\}$ now remains. Theoretically, this involves solving $O(k!)$ many LPs. In our case, due to precedence constraints and time windows, very few permutations correspond to feasible orderings, though. If there are two

$$
\begin{array}{rlrcll}
 & \min & \frac{\alpha_2}{UB_{worktime}} \cdot w + \frac{\alpha_4}{2 \cdot N} \cdot (y_1 + y_2) + \frac{\alpha_3}{J} \cdot \sum_{l=1}^{k} x_l & & & \\
1: & \text{s.t.} & t_{l+1} - t_l & \geq & \mathrm{d}_{\mathcal{J}}(j_l) + \mathrm{tr_time}(j_l, j_{l+1}) & l = 1 \ldots k-1 \\
2: & & \mathrm{sb}_{\mathcal{J}}(j_l) - t_l & \leq & x_l(\mathrm{sb}_{\mathcal{J}}(j_l) - \mathrm{hb}_{\mathcal{J}}(j_l)) & \\
3: & & t_l - \mathrm{se}_{\mathcal{J}}(j_l) & \leq & x_l(\mathrm{he}_{\mathcal{J}}(j_l) - \mathrm{se}_{\mathcal{J}}(j_l)) & \\
4: & & x_l & \geq & 0 & l = 1 \ldots k \\
5: & & t_l & \geq & \mathrm{hb}_{\mathcal{J}}(j_l) & \\
6: & & t_l & \leq & \mathrm{he}_{\mathcal{J}}(j_l) & \\
7: & & \mathrm{sb}_{\mathcal{N}}(n) - t_1 & \leq & y_1 \cdot (\mathrm{sb}_{\mathcal{N}}(n) - \mathrm{hb}_{\mathcal{N}}(n)) & \\
8: & & t_k + \mathrm{d}_{\mathcal{J}}(j_k) - \mathrm{se}_{\mathcal{N}}(n) & \leq & y_2 \cdot (\mathrm{he}_{\mathcal{N}}(n) - \mathrm{se}_{\mathcal{N}}(n)) & \\
9: & & y_1, y_2 & \geq & 0 & \\
10: & & \mathrm{min_time}_{\mathcal{N}}(n) & \leq & w & (\#) \\
11: & & \mathrm{max_time}_{\mathcal{N}}(n) & \geq & w & \\
12: & & t_k + \mathrm{d}_{\mathcal{J}}(j_k) - t_1 & = & w & \\
13: & & t_1 & \geq & \mathrm{hb}_{\mathcal{N}}(n) & \\
14: & & t_k + \mathrm{d}_{\mathcal{J}}(j_k) & \leq & \mathrm{he}_{\mathcal{N}}(n) &
\end{array}
\tag{6.21}
$$

Figure 6.3: *LP (6.21): Finding optimal starting times for a given sequence*

distinct jobs that have to be done in parallel, there is no feasible ordering at all for the given set of jobs. Otherwise, we calculate all permutations that correspond to feasible orderings of jobs via a recursive function. We found through experimental analysis, that very few permutations are valid and thus the algorithm is very fast with typical input (see Sec. 6.5.1). This is due to the fact that in home health care many jobs have to be performed in the morning, after lunch and at bedtime. Hence, there is a lot of overlapping at these times. Intensive care, on the other hand, is provided all day, but usually takes much longer time and thus also overlaps with other long running jobs.

Algorithm 12 describes the procedure: Initially called with $S = \emptyset$, $U = \{j_1, \ldots, j_k\}$, the algorithm moves an element of U to any feasible position in S, and recursively checks for possible extensions of S (lines 15–20). (To ease the notation assumed that inserting before the first or after the last element of S can also be represented by an insertion between two consecutive elements). If U becomes empty, S is a valid ordering, and (6.21) provides optimal starting times for that ordering (lines 12,13). After termination, ***bestSol*** contains the best ordering found.

We use constraint programming ideas to fix any job that needs to be included between two other jobs (lines 1–11) and we stop the current recursion as soon as we find a job for which we cannot fulfill the precedence constraints in the schedule S currently under construction (lines 6 and 10,11). I.e. we have to revise an earlier branching decision, if inserting a job at a required position is not possible because some other time window on the schedule or the nurse's time window will be violated. Basically, such a test requires checking (6.3), (6.4), (6.7) and it can be improved by applying forward and backward push information (see Solomon [193]). Additional constraints on the roster's design may be included in this algorithm as well, and they will further reduce the search space.

To avoid repeated calculations for identical job-sets, we use a cache that stores all optimal

Algorithm 12 Find best sequence for set U

generateSequence (sequence S, set U)

```
1: // try to fix some jobs
2: for i ← 1 ... length(S) + 1 do
3:    if (U = ∅) then goto 12
4:    for all j ∈ U do
5:       if (precedence for j is: after S_{i−1} and before S_i) then
6:          if (inserting j between S_{i−1} and S_i does not violate any time window) then
7:             U ← U \ {j}
8:             insert j between S_{i−1} and S_i
9:             goto 4 // restart loop at first element in U
10:         else
11:            return // infeasible
12: if (U = ∅) then
13:    solve LP (6.21), and eventually update bestSol
14: else
15:    select j ∈ U
16:    for i ← 1 ... length(S) + 1 do
17:       if (inserting j between S_{i−1} and S_i does not violate any time window) then
18:          S′ = S;
19:          insert j between S′_{i−1} and S′_i
20:          generateSequence (S′, U \ {j}) // recursive call
```

schedules found. In the experiments, only some thousand different sequences are determined, whereas some million requests are answered by the cache (see Sec. 6.5.1).

6.3.3 Initial Solutions via Constraint Programming

Initial heuristics are used in obtaining a first solution quickly. They provide us with the first rosters, and if the dispatcher allows more time for optimization, they serve as a starting point for improvement heuristics. Our first approach was to adapt insertion and scheduling heuristics developed for the vehicle routing with time windows (Solomon [193]). Though they are very fast — usually only a split second — the solutions obtained turned out not to be applicable, as in most cases not all jobs could be covered by nurses. Therefore, we decided to follow a CP approach within an incomplete tree search. On typical instances, we usually obtain very good starting solutions within a few seconds. Furthermore, we may trade time vs. quality with such an approach. We can simply stop at any time after having found the first solution and take the best one produced so far.

6.3.3.1 Formulation

We use a redundant modeling for the HHC problem. We represent the roster for nurse n by a set $R^{(n)}$, $n = 1 \dots N$ and each job $j \in \mathcal{J}$ by an integer variable i_j. In a solution, the set $R^{(n)}$ contains all jobs assigned to nurse n, and variable i_j is the nurse who has to serve job j. The fact that we have to cover all jobs (6.1) is implied by this model, since each job is linked to a

nurse. Initially,

$$\mathrm{pos}(R^{(n)}) = \{j \mid \mathrm{quali}_{\mathcal{J}}(j) \subseteq \mathrm{quali}_{\mathcal{N}}(n)\}, \quad n \in \mathcal{N}, \quad \mathrm{req}(R^{(n)}) = \emptyset, \tag{6.22}$$

$$\mathrm{pos}(i_j) = \{n \mid \mathrm{quali}_{\mathcal{J}}(j) \subseteq \mathrm{quali}_{\mathcal{N}}(n)\}, \quad \forall\, j \in \mathcal{J}, \quad \mathrm{req}(i_j) = \emptyset, \tag{6.23}$$

which already covers (6.8). Next, we state that the n rosters must not intersect (6.2) by a global cardinality constraint (Régin [173]). We force consistency between roster and job variables by stating

$$\forall\, n \in \mathcal{N},\, j \in \mathcal{J}: \quad j \in R^{(n)} \iff i_j = n. \tag{6.24}$$

We can improve this model by providing redundant information. If we know from preprocessing that two jobs j, j' have to be served in parallel, they cannot be assigned to just one nurse. Hence, we add

$$i_j \neq i_{j'} \quad \forall j, j' \in \mathcal{J} \text{ that have to be in performed in parallel.} \tag{6.25}$$

Using the sequencing cache, we can add a forward checking. Whenever the shifting of job j to the current required set of a nurse would result in an infeasible ordering, we can remove that job from the nurse's domain:

$$\begin{aligned}&\forall\, n \in \mathcal{N}:\\ &\quad \forall j \in \mathrm{pos}(R^{(n)}) \setminus \mathrm{req}(R^{(n)}):\\ &\quad\quad \mathrm{req}(R^{(n)}) \cup \{j\} \text{ infeasible} \Rightarrow \mathrm{pos}(R^{(n)}) \leftarrow \mathrm{pos}(R^{(n)}) \setminus \{j\}\end{aligned} \tag{6.26}$$

6.3.3.2 Branching and Tree Traversal

If domain filtering alone does not provide a solution or failure, we have to branch. Our strategy here is to select a job j for which the possible set $\mathrm{pos}(j)$ is the smallest, and to first choose a nurse assignment which will result in the best overall improvement.

Algorithm 13 Goal 1: Branch on job with smallest domain, and assign best nurse first

goal1

1: $j \leftarrow$ job with minimal domain, that is not yet fixed.
2: $n \leftarrow$ nurse in $\mathrm{pos}(j)$ with best improvement value
3: left branch: $(i_j = n)$
4: right branch: $(i_j \neq n)$

Having produced two new subproblems, we have to select the next one to process. We use Limited Discrepancy Search (LDS) (Harvey and Ginsberg [105]) to traverse the search tree.

6.3.4 Improvement Heuristics

A solution can often be improved by applying local changes. Metaheuristics, like Simulated Annealing (Kirkpatrick et al. [128], van Laarhoven and Aarts [203]) and Tabu Search (Glover [90, 91]) are widely used and have been shown to produce good solutions in a reasonable amount of time.

In our approach, both metaheuristics are based on a simple *1-shift*, i.e. removing a job from one place and inserting it elsewhere. To flexibly the operator we allow an insertion at a different set of jobs as well as an insertion at a stock Γ. Accordingly, we may remove a job from a set of rosters or from the stock. Of course, deleting a job and re-inserting it in the same set again is prohibited. Moving a job to the stock is highly penalized by adding +1 to the objective.

The *costs* of a 1-shift are defined as the gain from deleting job j from a roster or the stock minus the loss resulting from inserting it again into a certain position in a roster or into the stock.

Using the stock we can start with $\Gamma = \mathcal{J}$ and apply the metaheuristics to an empty solution, or we can use a solution found earlier and try to improve on that. Notice, that we allow intermediate steps with a non-empty stock (i.e. we relax (6.1)). Any solution returned by either metaheuristics, however, requires all jobs to be assigned to nurses.

Since we calculate optimal sequences via the approach presented in Sec. 6.3.2, it suffices to partition $\mathcal{J}$ into good subsets. As with the constraint programming approach, LP (6.21) can only be applied, if we replace the minimum work time constraint (#) for subsets under construction by $w \geq 0$.

6.3.4.1 Simulated Annealing

Simulated Annealing (SA) tries to randomly pick good moves and to improve the current solution by applying these moves. To escape from local optima, deteriorating moves are accepted with a certain probability which is gradually decreased during the optimization process. Because of its similarity to smelting processes, this probability is called temperature. The algorithm terminates, if the system is regarded as “frozen” – either by a low temperature, or by not moving to a better state.

In our implementation, the starting temperature is determined automatically, using a method by Aarts et al. [1]. We start with some 1-shifts at a low temperature T and increase T until an acceptance rate of 100% is reached. This temperature is then used as the starting point for the cooling scheme.

The next move is determined by choosing a job and a nurse randomly. Improving moves are always accepted, whereas deteriorating moves are only accepted with probability $e^{-\frac{\Delta C}{T}}$, where ΔC is the cost change resulting from the current move (see Kirkpatrick et al. [128], van Laarhoven and Aarts [203]). After 1000 iterations we decrease the temperature by a factor of 0.9. Should our time limit exceed or the acceptance rate drop below 0.1% we stop. Simulated Annealing is sketched in Alg. 14.

6.3.4.2 Tabu Search

The Tabu Search (TS) used follows the ideas proposed by Glover [90, 91]. The general strategy of Tabu Search is to systematically explore all possible moves from the current to a neighboring solution. The move leading to the best (non-tabu) neighboring solution is accepted, even if this results in a deterioration of the objective function. To prevent the search from cycling, a tabu list which stores the inverse move for a certain number of iterations is used. Thus, all solutions which can be obtained by applying a move stored in the tabu list are not

Algorithm 14 Simulated Annealing

SimulatedAnnealing

1: $T \leftarrow$ initial temperature
2: **repeat**
3: $\quad$ $iterations \leftarrow 0$
4: $\quad$ **repeat**
5: $\quad\quad$ $iterations \leftarrow iterations + 1$
6: $\quad\quad$ select next move randomly
7: $\quad\quad$ probe that move; call Algorithm 12 (*sequencing*) for sets changed
8: $\quad\quad$ $\Delta C \leftarrow$ cost change caused by that move
9: $\quad\quad$ **if** $((\Delta C < 0)$ **or** $(e^{-\frac{\Delta C}{T}} > \text{rand}()))$ **then**
10: $\quad\quad\quad$ perform that move, update current solution
11: $\quad\quad\quad$ **if** (improved globally) **then**
12: $\quad\quad\quad\quad$ update best solution;
13: $\quad$ **until** $(iterations \leftarrow 1000)$
14: $\quad$ $T \leftarrow 0.9 \cdot T$
15: **until** ((acceptance rate too low) **or** (time() $> limit$))

considered. An aspiration criterion allows us to override this rule, if a move improves the best global solution known so far.

We perform the best 1-shift among all possible ones (in the above sense) to obtain a different solution. I.e., we have to find an optimal sequencing for each member of the 1-shift neighborhood. We set the inverse of that move tabu for the next 10 iterations and update a table fr counting how often a certain job is assigned to a nurse. fr is used for *diversification* by adding a specific penalty for frequently used moves. By doing so, we gradually manipulate the cost function into moving the search away from some hot spots. The penalty term represents the additional costs incurred by moving job j to nurse n. It was originally proposed by Taillard et al. [195]:

$$p(j,n) = \Delta_{\max} \cdot \frac{fr(j,n)}{fr_{\max}}, \tag{6.27}$$

where $fr(j,n)$ is the frequency of moving job j to nurse n, $fr_{\max}$ is the maximal frequency over all nurses and jobs, and $\Delta_{\max}$ is the maximal absolute difference between two consecutive moves performed so far (excluding moves from/to the stock because of their high extra penalty).

Diversification penalties are added to the original objective if no global improving solution is found for some iterations (e.g. 2000). As soon as we find an improving solution we switch back to the original objective function. Should we stall for longer time, we terminate the search. Algorithm 15 sketches the procedure.

We do not consider any specific intensification within the pure Tabu Search. The hybrid approach described in the next section will introduce such a technique.

Algorithm 15 Tabu Search

TabuSearch

1: $iterations \leftarrow 0$
2: **while** (($iteration \leq max_iter$) **and** (time() $\leq limit$)) **do**
3: **if** (diversification required) **then**
4: select best neighbor using diversification, (non-tabu or global improvement)
5: **else**
6: select best neighbor, (non-tabu or global improvement)
7: perform move
8: $iterations \leftarrow iterations + 1$; update fr, $tabu$
9: **if** (improved globally) **then**
10: update best solution

6.4 A Hybrid Solution Approach

As we will see in the experimental evaluation, the approaches we have just presented are able to quickly find good operational solutions. There are two points, which can be improved further: Firstly, if time allows, we would like to optimize more, and achieve better plans than those found so far. And secondly, we would like to collect several diverse plans: We experienced that the acceptance of computer generated solutions increases if not only one optimized plan is presented, but also some alternative plans, and dispatchers as well as nurses know that the final decision is made by a human.

In this section we discuss a hybridization technique which accounts for both points. The approach offers a diverse variety of very good solutions to the dispatcher and lets him/her decide which roster is best.

Before presenting this approach, let us summarize some characteristics of the approaches presented in the previous section:

(i) Different approaches emphasis on different parts of cost- or constraint structure.

(ii) Having found several starting solutions, it is not clear at all which one will be the best starting point for improvement heuristics.

(iii) In many cases improvement heuristics stop because they are trapped in local optima. A different method may easily escape from such an optima.

As a result, applying only one improving algorithm to only the best initial solution is not likely to produce the best possible solution. This problem has been recognized in several application fields and approaches based on the concept of a solution pool have been presented. These strategies turn out to be quite powerful in practical experiments (see Taillard et al. [196] for examples of different heuristics and different problems).

6.4.1 The Solution Pool Concept

The idea of a solution pool is to store some intermediate solutions generated via the improvement heuristics and exploit the stored “know-how” for designing new solutions. In a Tabu

Search context, such an approach refers to the utilization of long term memory (see Glover [90]). Three different approaches can be identified within such a framework:

Combine parts of old solutions into a new solution: For the VRPTW, Rochat and Taillard [175] collect intermediate solutions of Tabu Search. They periodically pick some of the generated routes at random (giving higher probability to routes from good solutions) and combine them with a new solution. For customers not covered by this method, they add new routes, each containing only one of these customers. The newly generated solution is the starting point for the next Tabu Search optimization. Schulze and Fahle [184] improved on this by using a deterministic set covering heuristics. There, all solutions are divided into routes, and a set covering heuristics is used to select those routes that cover all customers at minimal costs. Customers covered on more than one route are deleted from all but one route, thus improving the solution's quality further.

Similar techniques were used in Chapter 5 for airline crew rostering

Apply different heuristics to the solutions in the pool: A bunch of heuristics is applied to all solutions stored in the pool. Caused by different characteristics of the heuristics used, solutions trapped in a local optima via one heuristic method can often be improved in turn by subsequent heuristics (Bouthillier et al. [39]).

Extract statistical data from the pool as a guidance for heuristics: We count how often a certain assignment occurs in the solutions found so far. Then we prefer assignments which occur often to rare ones. Such an approach is very similar to the "fitness" idea used in genetic algorithms (see Goldberg [93]).

In our approach, we combine two of these ideas, namely applying different heuristics to the solutions in the pool and using statistical measures for good solutions. We do not consider the first approach which produces very good solutions, but needs a considerable amount of running time. Thus it is not feasible in the planning environment we are focusing on, where dispatchers like to see a solution within a few minutes.

6.4.2 Using a Good Solution to Improve the Search

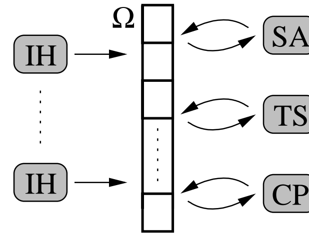
Let Ω be a set of solutions found by some heuristics. In the beginning Ω contains solutions found by initial heuristics (IH). In the optimization loop, we apply our improvement heuristics to each solution in Ω, and replace the old solution by the improved one (see Alg. 16).

Whereas Tabu Search and Simulated Annealing simply start with the solution L provided by the pool, the CP approach has to be adapted slightly. We use the general settings described in Section 6.3.3, and change the branching scheme of goal 1 (Algorithm 13) to consider decisions in L as well. The idea is to select the next job j as before, and to use the same nurse n assigned to j as in solution L. The ratio behind this is that in a good solution many assignments already correspond to assignments in an optimal solution, and we will use as many of these assignments as possible. If the nurse used in L is not available (e.g. because some propagation or earlier branching has assigned her elsewhere), we select the next nurse as in goal 1. The corresponding goal 2 is described in Alg. 17.

Furthermore, we try to improve the CP approach by modifying the search order with respect to soft constraint satisfaction (provided by the input solution). Given a solution L, we can sort

Algorithm 16 A pool Ω stores initial and improved solutions found during optimization

1: $P \leftarrow$ problem instance
2: $\Omega \leftarrow \emptyset$
3: **while** (Ω not full) **do**
4: $S \leftarrow \text{IH}(P)$; $\Omega \leftarrow \Omega \cup \{S\}$
5: **repeat**
6: select $S \in \Omega$; $\Omega \leftarrow \Omega \setminus \{S\}$
7: select a heuristic method $h \in \{\text{TS, CP, SA}\}$
8: $S' \leftarrow h(S, P)$; $\Omega \leftarrow \Omega \cup \{S'\}$
9: **until** (termination criterion)
10: **return** all solutions in Ω



Algorithm 17 Goal 2: Branch on job with smallest domain and assign the nurse that was also used in solution L

goal2
1: $j \leftarrow$ job with minimal domain, that is not yet fixed
2: $n \leftarrow$ nurse that was also used in solution L for job j if possible
3: else $n \leftarrow$ nurse in $\text{pos}(j)$ with best improvement value (as in goal 1)
4: left branch: $(i_j = n)$
5: right branch: $(i_j \neq n)$

the $\text{sc}(n, j)$ values of that solution in a preprocessing step. In goal 3, we chose j as the first job in this order that is not yet assigned, and we determine the nurse as in goal 2. Goal 3 is described as Alg. 18.

Algorithm 18 Goal 3: Branch on job with best soft constraint values first, and assign the nurse that was also used in solution L

preprocessing-step
1: sort $\text{sc}(n, j)$ for all (j, n) pairs found in L

goal3
1: $j \leftarrow$ first job in sorted list that is not yet fixed
2: $n \leftarrow$ nurse that was also used in solution L for job j if possible
3: else $n \leftarrow$ nurse in $\text{pos}(j)$ with best improvement value (as in goal 1)
4: left branch: $(i_j = n)$
5: right branch: $(i_j \neq n)$

6.4.3 Using the Essence of All Solutions

A further solution improvement is gained by using statistical information from the solution pool to guide the constraint programming approach. The idea is to detect job/nurse pairs, that often occur in solutions with high quality, and to consider those assignments early in a systematic search. Vice versa, pairs occurring in low quality solutions only should be consid-

ered late in a systematic search. This idea leads to a modified variable ordering and variable assignment goal within the CP approach.

Let $\Omega = \{S_1, \ldots, S_k\}$ be a pool of k (diverse) solutions, $v(S)$ be the value of solution S. For any nurse $n = 1, \ldots, N$ and any job $j = 1, \ldots, J$, let $\mu(n, j)$ be the accumulated objective value of all solutions in Ω assigning job j to nurse n. Accordingly, let $\kappa(n, j)$ be a counter for the number of solutions containing such an assignment. We consider those assignments that have a low value of $\frac{\mu(n,j)}{\kappa(n,j)}$ as "good". This value is the average quality of solutions containing an assignment of j to n. We order the pairs (n, j), $n = 1, \ldots, N$, $j = 1, \ldots, J$ according to increasing values $\frac{\mu(n,j)}{\kappa(n,j)}$. In the branching decision described in Sec. 6.3.3 we replace the variable selection by a selection step which takes the first pair (n, j) for which an assignment is possible. Then we assign j to nurse n on the left branch, and we exclude j from the possible set of n on the right branch.

Since we usually only perform an incomplete search, it is likely that assignments made in the first part of the search tree will be included in the best solution found after interrupting the search. To prevent the statistical information gathered in $\mu(n, j)$ and $\kappa(n, j)$ from being part of a self-stabilizing process, we refine the collection process as follows: We increase the value $\frac{\mu(n,j)}{\kappa(n,j)}$ by an additional penalty depending on the frequency of a certain assignment. The more often an assignment is used, the higher the penalty added. We use

$$\frac{\mu(n,j)}{\kappa(n,j)} + \left(\frac{\kappa(n,j)}{\max_{n',j'}\{\kappa(n',j')\}} \cdot \left[\max_{n',j'}\{\mu(n',j')\} - \min_{n',j'}\{\mu(n',j')\} \right] \right). \tag{6.28}$$

In doing so, high quality parts are still assigned first, but rarely used parts are preferred to more frequently used ones. We assume this strategy produces more diverse solutions than the pure quality ordering alone. The algorithmic framework is presented in Alg. 19.

Algorithm 19 Goal 4: Branch on job/nurse pairs that occur in good solutions found earlier

preprocessing-step

1: sort (n, j) according to increasing values obtained by (6.28)

goal4

1: select (n, j) as the first in the previously sorted list that is not yet fixed.
2: if no such pair exists, select (n, j) as in goal 3
3: left branch: $(i_j = n)$
4: right branch: $(i_j \neq n)$

6.5 Numerical Evaluation

All algorithms were coded in C++ and compiled by the GNU g++ 2.95.3 compiler using full optimization. Our benchmark tests were run on a Pentium III-933 PC with 512MB RAM operating Linux kernel 2.4.19. For solving LP (6.21) we use ILOG CPLEX 7.5 [119], and for the constraint programming approach we apply ILOG SOLVER 5.2 [120]. Simulated Annealing was implemented using PARSA [130].

roster size	recursive calls	LP calls
2	1.094	0.962
3	1.356	1.028
4	1.443	1.015
5	1.665	1.073
6	2.057	1.139
7	2.837	1.344
8	3.986	1.555
9	6.548	2.585
10	7.936	2.988
11	6.566	1.882
12	7.342	1.911
13	18.398	4.162
14	24.560	4.317
15	36.350	3.649

Table 6.1: *Sequencing algorithm efficiency*

We used 10 synthetic test scenarios, containing between 20 and 50 nurses, and between 111 and 326 jobs. The data was generated according to real-world input. Each job lasts between 6 and 72 minutes, nurses' hard time windows between 5 and 9 hours. Locations were chosen randomly, and euclidean distances were calculated between these locations. Also, the soft constraint factor for each job was selected randomly.

The tables and figures presented in the following show the quality value as defined by the objective. It contains routing quality, time window penalties as well as the soft qualification measure. All terms were equally weighted by $\alpha_i = \frac{1}{5}$. The run times are given in seconds. To reflect a real-world setting, all run times are limited to 600sec or 840sec, respectively. A deeper experimental analysis of our methods is given by Bertels [29].

6.5.1 Optimal Sequences

Table 6.1 shows that enumerating all possible orderings for a given set of jobs does not drastically boost the computing time, if we use time window constraints and job precedences to limit the enumeration tree. Algorithm 12 builds only few sequences and even less LPs have to be solved. Notice, that a complete enumeration for a roster of size k would result in $k!$ recursive calls. E.g. there are $6\ 227\ 020\ 800$ possible permutations for a 13-job-roster whereas we need less than 19 calls on the average. An even smaller number corresponds to valid orderings (which have to be processed by the LP).

In Fig. 6.4 the efficiency of the sequence cache for the basic solution methods is shown. We can see that CP generates many identical rosters due to the systematic search. TS benefits considerably from the cache when enumerating the neighborhood. Due to its randomness SA arbitrarily jumps around in the solution space. SA is forced to behave more locally only after a decrease in temperature and thus can benefit from previously optimized rosters. The peak of the CP+TS curve results from a series of diversification steps that guide the search to a new part of the search space. The small bend of the CP curve is due to the internal implementation of LDS in ILOG SOLVER, see [120, Ref. Manual, p. 90].

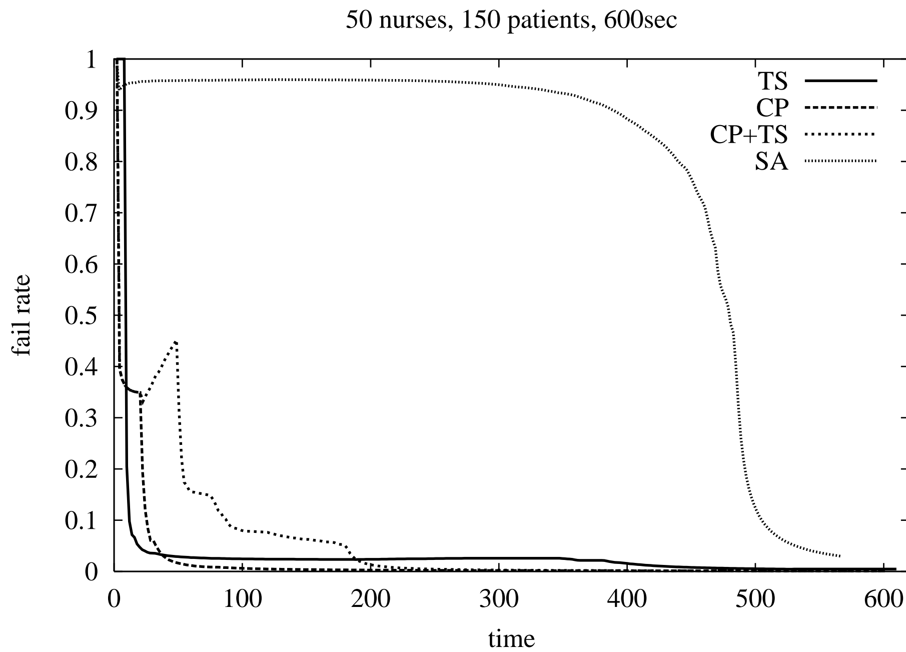


Figure 6.4: *Sequencing cache fail rate*

6.5.2 Initial Solutions via Constraint Programming

In Table 6.2 we show results obtained for the initial solution. That is, we applied CP (using goal 1) to each benchmark set until the first solution was found. Whenever the first solution was found in less than 10 seconds, we also show the best value found after 10sec. (To get an idea of the quality obtained, these results can be compared to those in Table 6.3).

As can be seen in Table 6.2, a good starting solution can be found in a short time even for larger instances. Typically, the first solution, and the one found after 10sec are of the same quality. The 50 nurse instances take considerably longer than the smaller ones. There are two possible reasons for this: The CP heuristics might be bad, making many wrong decisions before finding the first solution, or the time used in each choice point may be high. When

sizes	first solution		after 10sec
$\|\mathcal{N}\|$-$\|\mathcal{P}\|$-$\|\mathcal{J}\|$	time in sec.	objective	objective
20-40-111	1.70	0.201277	0.175840
20-60-123	2.70	0.224217	0.218209
20-80-137	3.49	0.198873	0.187321
30-60-184	5.14	0.196026	0.195011
30-90-184	6.34	0.192929	0.188494
30-120-177	6.02	0.182207	0.179576
30-150-195	7.36	0.188585	0.185647
50-100-304	15.16	0.181553	—
50-150-319	20.19	0.169044	—
50-200-326	22.45	0.171422	—

Table 6.2: *Finding initial solutions via CP*

analyzing the results in more detail we found that the number of choice points explored, and the number of failures encountered were reasonable. The time spent for sequencing sets, however dominates the overall running time. In the beginning sequencing rosters is quite expensive in terms of computing time, since no information is stored in the cache. Later on, many sequencing requests can be answered by the cache. Thus succeeding computations benefit greatly from the bigger effort required in the beginning.

6.5.3 Comparing the Heuristics

Unfortunately, if CP continues running for 10min, the good start is not followed by a good convergence. The first column in Table 6.3 gives solution values which are only slightly better than the initial ones found by CP within the first few seconds. Simulated Annealing (started on an empty initial solution) produces better solutions than CP, but cannot compete against Tabu Search (also started on an empty solution) as can be seen in column three of that table. Tabu Search outdoes both of the previously mentioned approaches — if it can find a solution at all. In three cases Tabu Search seems to be too aggressive, and it is not able to generate a feasible solution at all within 600 seconds.

Interestingly, although Simulated Annealing and Tabu Search are both based on the same 1-shift neighborhood, their solution quality differs significantly. Such a behavior seems to be partly approach immanent (see e.g. Osman [157]). Another reason is, again, the sequencing. Whereas TS and CP systematically check their neighborhood and thus can reuse previously generated sequencing information, SA "jumps" randomly, resulting in much more cache fails and thus, much more calculations of sequences (see Fig. 6.4).

Motivated by the success of TS we started TS on the first solution found by CP (we refer to this approach as CP+TS). Not only does this approach ensure obtaining a feasible solution, it also appears to be the best approach in this test. Except for the smallest instances, it surpasses all other methods.

Figure 6.5 shows a typical plot comparing our four approaches. SA when starting from scratch takes quite some time to improve the solution. SA does not reach the solution quality of TS, or CP+TS. CP quickly finds a solution, but has difficulties improving it. TS needs some time to find a valid starting point which it then quickly improves. Combining CP and TS brings together the advantages of both approaches.

6.5.4 Combining Approaches

Instead of stopping the search after the termination of TS, we can also trade our computing time differently. In one test we ran CP for two minutes to find 10 different initial solutions which we stored in Ω. Then each of these solutions was optimized via a Tabu Search limited to one minute each (starting diversification after 500 non-improving rounds). Finally, all information collected in the $\mu(\boldsymbol{n}, \boldsymbol{j})$, and $\kappa(\boldsymbol{n}, \boldsymbol{j})$ tables was used to run CP using goal 4 for additional two minutes.

The other test consists of running CP for an initial solution (goal 1), followed by alternating runs of TS and CP on the best solution found (we used goal 2 which produced a similar solution quality as goal 3). The time limit of a round was raised whenever the previous round did not improve the global best solution.

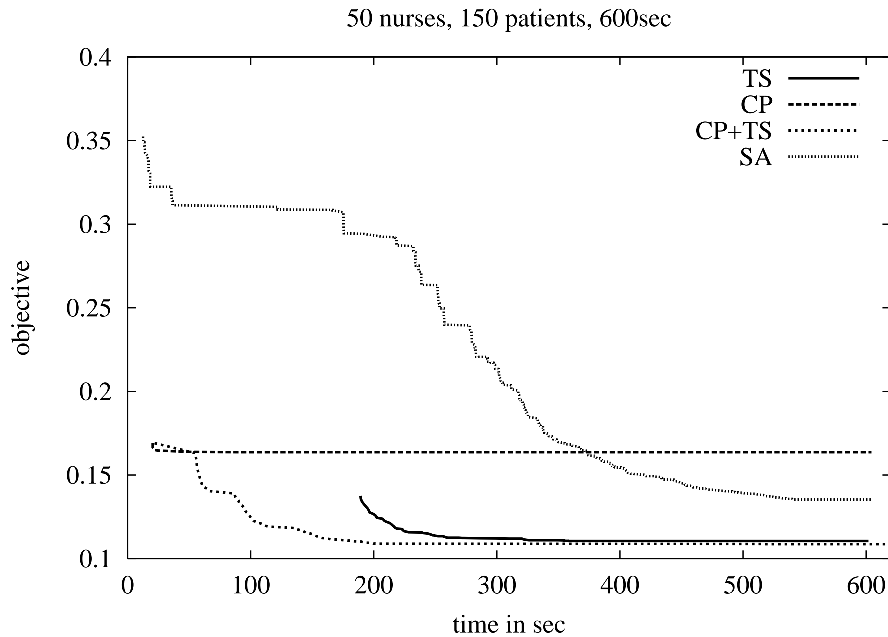


Figure 6.5: *Comparison of the heuristics*

sizes	objective after 600sec			
$\lvert\mathcal{N}\rvert$-$\lvert\mathcal{P}\rvert$-$\lvert\mathcal{J}\rvert$	CP	SA	TS	CP+TS
20-40-111	0.171069	0.140114	**0.130188**	0.132068
20-60-123	0.207763	0.169643	**0.153611**	0.159384
20-80-137	0.185795	0.140229	—	**0.131078**
30-60-184	0.190910	0.157389	—	**0.130886**
30-90-184	0.186522	0.137593	0.122436	**0.119256**
30-120-177	0.176804	0.139311	0.121989	**0.121133**
30-150-195	0.183405	0.141220	0.121386	**0.120928**
50-100-304	0.174755	0.143110	0.119550	**0.116424**
50-150-319	0.163636	0.135278	0.110572	**0.108694**
50-200-326	0.167059	0.137677	—	**0.115819**

Table 6.3: *Results for constraint programming (CP), Simulated Annealing (SA), and Tabu Search (TS) started from an empty solution, and for Tabu Search started on the first solution found by constraint programming (CP+TS). A dash indicates that no solution was found. Each approach ran for 600sec.*

sizes		objective	
$\lvert\mathcal{N}\rvert$-$\lvert\mathcal{P}\rvert$-$\lvert\mathcal{J}\rvert$	CP+TS	CP+TS loop	10 * CP+TS
20-40-111	0.132068	0.136548	**0.130534**
20-60-123	0.159384	0.158962	**0.153450**
20-80-137	0.131078	0.130157	**0.129449**
30-60-184	0.130886	0.132782	**0.129801**
30-90-184	0.119256	**0.118036**	0.118690
30-120-177	0.121133	0.125103	**0.120917**
30-150-195	0.120928	0.121823	**0.120806**
50-100-304	0.116424	**0.113853**	0.115200
50-150-319	0.108694	0.108091	**0.107043**
50-200-326	0.115819	**0.113778**	0.115029

Table 6.4: *Comparing combined approaches. Results for CP+TS, alternating CP and TS (CP+TS loop), and applying TS to 10 initial CP solutions (10* CP+TS). Running times between 600sec and 840sec.*

In Table 6.4 we present the results obtained by these two methods. Evidently, applying short optimization runs to a larger set of solutions outperforms both, the simple approach (CP+TS) considered in the previous section, as well as the alternating approach. Goal 4 was seldom able to find even better solutions.

Figure 6.6 compares the CP+TS method vs. alternating CP and TS vs. TS alone on 10 solutions.

6.6 Conclusions

We presented a compact model to the home health care problem which is flexible enough to break down most real-world HHC problems of different characteristics. Furthermore, we developed several solution approaches for the model. The approaches find good quality rosters that satisfy routing, qualification, time windows and soft constraints.

We combined the effectiveness of LP for finding optimal starting points of jobs and the effectiveness of CP for generating feasible ordering. This module is utilized by CP and local search methods. We can offer several good solutions to the end users by using a pool of solutions, and we can use information collected in the pool to improve the quality of these solutions. Experimental results show significant gains when these different methods are combined.

The industrial prototype is currently undergoing intensive user evaluation, where solutions generated by our optimization methods are competing in real-world scenarios. In a second step, requests for additional legal and company constraints, or credit point systems (contributed by researchers from ergonomics) may be integrated into our algorithmic framework.

Furthermore, it would be interesting to investigate the experimental behavior of our methods when allowing more computing time and different solution techniques.

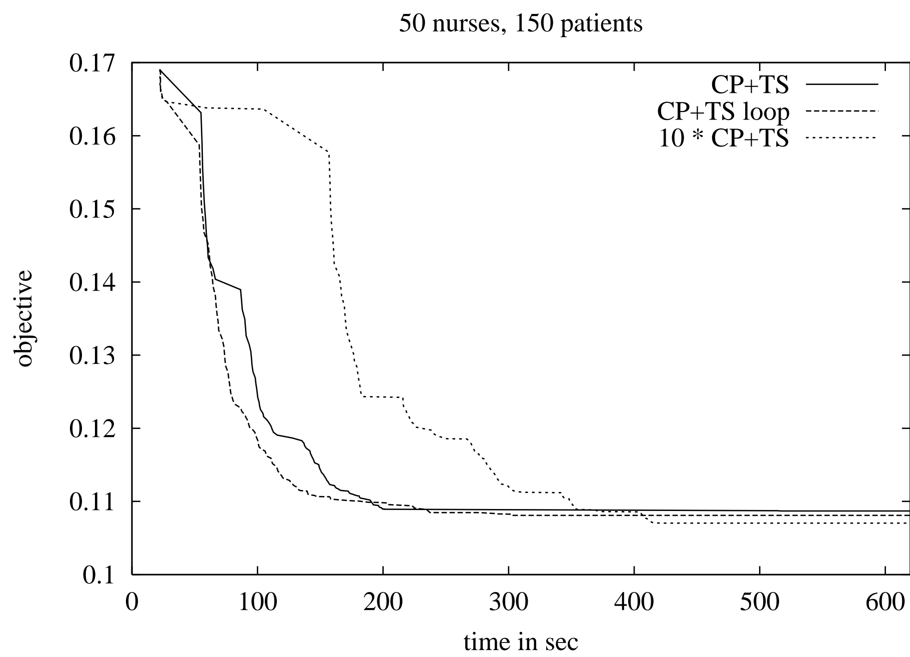


Figure 6.6: *Comparison of combined heuristics*

— 7 —

The Automatic Recording Problem

The *Automatic Recording Problem (ARP)* is an example of a problem that is constituted by two simpler constraints. We focus on algorithms that solve the problem exactly and give a tightened formulation of the ARP as an integer program (IP). Especially, we test the method of coupling domain filtering algorithms via Lagrangian relaxation on this multimedia application.

The technology of digital television offers new possibilities for individualized services that cannot be provided by analog broadcasts nowadays. Additional information like the classification of content, or starting and finishing times can be submitted within the digital broadcast stream. With this information at hand, new services that make use of individual profiles and maximize customer satisfaction can be provided.

One service – which is available already today – is an "intelligent" digital video recorder that is aware of its user's preferences and records automatically [10]. Such a recorder tries to match a given user profile with the information submitted by the different TV channels. E.g., a user may be interested in thrillers, the more recent the better. The digital video recorder is supposed to record programs so that the users' satisfaction is maximized. As the number of channels may be significant (more than 100 digital channels are possible), a service that automatically provides an individual selection seems to be reasonable and is in the center of current research activities e.g. within the *UP-TV* project[1] or the *TV-Anytime Forum*[2].

In this context, two restrictions have to be met. First of all, the storage capacity is limited (10 hours of MPEG-2 video need about 18 GB). And secondly, only one video can be recorded at a time (see Fig. 7.1). We define the problem more formally as follows:

Definition 9 *Let $n \in \mathbb{N}$, $V = \{1, \ldots, n\}$ the set of programs, $\mathbf{start}(i) < \mathbf{end}(i)\ \forall\ i \in V$ the corresponding starting and finishing times. $w = (w_1, \ldots, w_n) \in \mathbb{Q}_+^n$ the storage requirements, $K \in \mathbb{Q}_+$ the storage capacity, and $p = (p_1, \ldots, p_n) \in \mathbb{N}^n$ the profit vector.*

[1] Commission of the European Union, IST program, project number 1999-20751

[2] *"The global TV-Anytime Forum is an association of organizations which seeks to develop specifications to enable audio-visual and other services based on mass-market high volume digital storage in consumer platforms"* (cited from `http://www.tv-anytime.org/`)

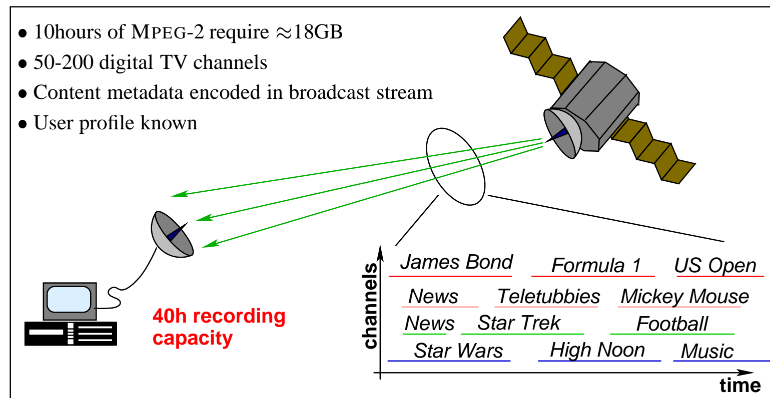


Figure 7.1: *The scenario for the Automatic Recording Problem.*

We say that the interval $I_i := [start(i), end(i)]$ corresponds *to program* $i \in V$*, and call two programs* $i, j \in V$ overlapping *whose corresponding intervals overlap, i.e.* $I_i \cap I_j \neq \emptyset$*. For* $X \subseteq V$ *we call* $p_X := \sum_{i \in X} p_i$ *the* user satisfaction (with respect to X).

The Automatic Recording Problem (ARP) *then is to find a subset* $X \subseteq V$ *such that*

(a) *X can be stored within the given disk size, i.e.* $\sum_{i \in X} w_i \leq K$.

(b) *one program at most must be recorded at a time, i.e.* $I_i \cap I_j = \emptyset \; \forall \; i \neq j \in X$

(c) *X maximizes the user satisfaction, i.e.* $p_X \geq p_Y \; \forall \; Y \subseteq V$*, Y respecting (a) and (b).*

Obviously, even if all programs are pairwise non-overlapping (i.e., if restriction (b) is obsolete), it remains to solve a knapsack problem. Thus, the ARP is NP-hard. But it can be approximated to arbitrary precision, since there is a simple fully polynomial time approximation scheme (FPTAS) for the ARP (a description can be found in Sellmann and Fahle [186]). Basically it is a slight extension of the well-known FPTAS for knapsack problems (see e.g. Martello and Toth [145]).

In the ARP's original setting, resources for solving the problem are limited. A typical VCR will have some megabytes of RAM and a simple processing unit. Approaches requiring huge amounts of memory, or high-end CPUs are not suited in such a scenario.

We will ignore that fact for a moment, and discuss several approaches for the ARP, among them three based on Lagrangian coupling (this chapter), and two alternative approaches in Chapter 8.

7.1 An Integer Linear Programming Formulation

Linear programming based branch-and-bound approaches have proved to be efficient, widely applicable and thus are quite commonly used. In every choice point, a bound based on the LP relaxation is being computed. If that bound is worse than the objective value B of the incumbent solution, backtracking occurs.

The ARP can be modeled straightforwardly as an ILP. We use a vector $x = (x_1, \ldots, x_n)$ of binary variables to describe whether movie i is selected in an optimal solution ($x_i = 1$) or not

($x_i = 0$). Thus, definition 9(a) is directly translated into a knapsack constraint ($\sum_i x_i \cdot w_i \leq K$). Def. 9(b) corresponds to a bundle of constraints enforcing that at most one variable may be set to one if the corresponding movies overlap in time. And the objective is to maximize $\sum_i p_i \cdot x_i$:

$$\begin{array}{lrcll} \text{Maximize} & \sum_{i=1}^n p_i x_i & & & \\ \text{subject to} & x_i + x_j & \leq & 1 & \forall\, i \neq j \in V\ I_i \cap I_j \neq \emptyset \\ & \sum_{i=1}^n w_i x_i & \leq & K & \\ & x & \in & \{0,1\}^n & \end{array} \tag{7.1}$$

The objective function maximizes the user satisfaction. Constraints of the form $x_i + x_j \leq 1$ ensure that for any two overlapping intervals I_i, I_j at most one program can be selected. Storage restrictions are enforced by the last row. The formulation can be tightened by replacing the "non-overlapping" constraints with maximal conflict clique constraints.

Definition 10 *A set $C \subseteq V$ is called a* conflict clique*, if $I_i \cap I_j \neq \emptyset\ \forall\ i,j \in C$. A conflict clique C is called* maximal*, if $\forall\ D \subseteq V$, D conflict clique: $C \subseteq D \Rightarrow C = D$. Let $M := \{C_1, \ldots, C_m\} \subseteq 2^V$ denote the set of maximal conflict cliques.*

Then, restrictions of the form $\sum_{i \in C_p} x_i \leq 1\ \ p = 1, \ldots, m$ imply that $x_i + x_j \leq 1$ for all nodes $i, j \in V$ whose corresponding intervals overlap. On the other hand, if $x_i + x_j \leq 1$ for all overlapping intervals, it is also true that $\sum_{i \in C_p} x_i \leq 1$, $p = 1, \ldots, m$. We will detail our discussion on maximal clique constraints in section 8.1.1. Here, we continue with the resulting IP, which is

$$\begin{array}{lrcll} \text{Maximize} & \sum_{i=1}^n p_i x_i & & & \\ \text{subject to} & \sum_{i \in C_p} x_i & \leq & 1 & p = 1, \ldots, m \\ & \sum_{i=1}^n w_i x_i & \leq & K & \\ & x & \in & \{0,1\}^n & \end{array} \tag{7.2}$$

7.2 Solving the ARP via CP based Lagrangian Relaxation

Domain reduction can help to improve the performance of a branch-and-bound search if the filtering is both, effective and efficient. Effective means, that it has to be able to filter many values, whereas efficiency measures how quickly the routine works.

We therefore solve the ARP via a constraint programming based branch-and-bound approach that enables us to add (real-world) constraints to the problem without changing the core ideas.

The effectiveness of a filtering algorithm depends mainly on the quality of the bounds it uses to estimate the impact of fixing a variable to one of its values. For the ARP, our experiments show that the continuous relaxation bound yields a good estimate of the solution quality that can be reached. Thus, it can be used for pruning purposes in a branch-and-bound approach. But it is not obvious to see, how this bound could be used effectively for filtering purposes, that is, other than by probing via full re-optimization, which is inefficient. On the other hand, domain reduction with respect to reduced cost information can be done quickly, but is not very effective.

7.2.1 Substructures of the ARP

The ARP can be viewed as a combination of two simpler optimization constraints: a knapsack constraint and a shortest path constraint. The latter requires to move costs of a node to the outgoing edges and to revers their sign. A different view on ARP allows us to identify another possible combination of simpler optimization constraints: a knapsack constraint, and a maximum weighted stable set constraint on an interval graph.

Therefore two possible CP models are available. The first model uses the shortest path constraint presented in Chapter 4 and the knapsack constraint for which a domain filtering algorithm is described in Chapter 3. The shortest path constraint needs time $O(|V|+|E|)$ and on a dense graph like in our application this results to a running time of $O(|V|^2)$ per choice point.

The second model also relies on the knapsack constraint and a constraint specially developed for maximum weighted stable set substructure (MWSSP) of the ARP (see Sellmann and Fahle [186]). It runs in time $\Theta(n \log n)$, and in amortized linear time for $\Omega(\log n)$ choice points.

7.2.2 CP based Lagrangian Relaxation for the ARP

In the remaining, we show how to apply CP based Lagrangian relaxation to the ARP. Asymptotic running time of the more general shortest path constraint is higher than of the specially designed maximum weighted stable set constraint. Thus, we will use the maximum weighted stable set constraint in the descriptions and in the experiments. The idea of that constraint is to use apply the simplex algorithm to the weighted stable set problem. Using a problem specific pivot strategy it is possible to achieve the running time mentioned above. This idea is due to M. Sellmann and we refer to Sellmann and Fahle [186] for technical details.

With the two domain filtering algorithms at hand, we are able to perform domain reduction on the two natural substructures of the ARP. According to the abstract description in Sec. 2.3.3, we will now tie the two filtering algorithms together:

As the domain filtering algorithm for MWSSP allows us to incorporate changing objectives at a low computational cost, we decide to relax the capacity constraint. We introduce a non-negative Lagrangian multiplier $\lambda \geq 0$ and define the Lagrangian subproblem

$$\begin{array}{llcll} \text{Maximize} & z(\lambda) := z & & & L(\lambda) \\ \text{subject to} & z & = & \sum_{i=1}^{n}(p_i - \lambda w_i)x_i + \lambda K & \\ & \sum_{i \in C_p} x_i & \leq & 1 & p = 1, \ldots, m \\ & x & \in & \{0,1\}^n & \end{array}$$

The Lagrangian multiplier problem then is to solve Minimize $z(\lambda)$, such that $\lambda \geq 0$. For every $\lambda \geq 0$, $z(\lambda)$ is a valid upper bound on the objective. Therefore, we can apply cost based filtering for the MWSS constraint on interval graphs each time we solve the Lagrangian subproblem. After we have found an optimal Lagrangian multiplier λ^*, we now use dual information $\pi \in \mathbb{Q}^m$ from the corresponding stable set subproblem to perform variable fixing with respect to the knapsack substructure. By (Lagrangian) relaxing the maximal clique constraints with multipliers $\pi \geq 0$, we obtain a knapsack problem. Let $\mu_i := \sum_{j\,:\,i \in C_j} \pi_j \; i = 1, \ldots, n$ and $\overline{\pi} := \sum_{j=1}^{m} \pi_j$. The problem then is to

$$\begin{array}{lrcl} \text{Maximize} & \sum_{i=1}^{n}(p_i+\mu_i)x_i - \overline{\pi} & & \\ \text{subject to} & \sum_{i=1}^{n} w_i x_i & \leq & K \\ & x & \in & \{0,1\}^n \end{array}$$

Relaxations of this problem again yield a valid upper bound, and we can apply domain filtering of the knapsack optimization constraint using the modified objective.

7.2.3 Implementation Details

We use four different approaches for our experiments: the first is a pure branch-and-bound algorithm without any problem tightening (referred to as *LG-0*). The second applies the domain filtering algorithms for knapsack and maximum weighted stable set problems on the original objective (*LG-1*). The third and the fourth approach (*LG-2* and *LG-3*) realize the idea of coupling domain filtering algorithms for linear optimization constraints via Lagrangian relaxation: *LG-2* calls for domain reduction only after the Lagrangian dual has been solved, whereas *LG-3* also applies domain filtering on the maximum weighted stable set constraint during the search for optimal Lagrangian multipliers.

7.2.3.1 Continuous Bound Computation

For pruning, the computation of a linear bound on the objective is needed. *LG-2* and *LG-3* obviously use the objective value corresponding to $L_{\mathcal{B}}(\lambda^*)$ for this purpose. As the computation via Lagrangian relaxation with stable set subproblems turned out to be very efficient, we used that algorithm for all four approaches.

7.2.3.2 Computation of λ^*

To determine λ^*, we used a method based on the golden section to maximize one-dimensional concave functions. We obtain a sequence of λ^k, $k \in \mathbb{N}$. Let $e_{max} := \max\{p_i/w_i \mid i = 1,\ldots,n\}$. Then, for all $\varepsilon > 0$ there exists a constant $c > 0$ such that

$$|\lambda^k - \lambda^*| < \varepsilon \qquad \forall\, k \geq c \cdot \log e_{max}$$

Thus, after $O(\log e_{max})$ iterations we can numerically approximate the optimal Lagrangian multiplier λ^*. Each iteration cost amortized linear time for at least $\Omega(\log n)$ choice points. Finally, in every choice point we add $O(n\log n)$ for the succeeding knapsack domain filtering algorithm. Therefore, the coupled domain filtering algorithm for the tight global continuous relaxation bound runs in time $O(n\log e_{max} + n\log n)$.

Notice, that the Lagrangian subproblem is totally unimodular. Thus, the Lagrangian relaxation bound has the same value as the bound that evolves from a linear continuous relaxation.

7.2.3.3 Branching Variable Selection

All algorithms choose the first node on the shortest path[3] with maximal efficiency p_i/w_i as the branching variable.

[3]according to the optimal reduced costs objective $\sum_{i=1}^{n}(p_i - \lambda^* w_i)x_i + \lambda^* K$

7.3 Numerical Results

All algorithms described above were tested on some synthetic benchmark sets. The experiments were performed on a PC with an AMD-Athlon 600 MHz processor and 256 MB RAM running Linux 2.2. The implementation was done in C++ and compiled by gcc 2.95 with maximal optimization (O3). The constraint programming based algorithms were built on top of ILOG SOLVER 5.0 [117], the linear integer programming approach utilized ILOG CPLEX 7.1 [118].

One major outcome of the experimental study is the ranking of the approaches described previously. We will see in Chapter 8 that other approaches are significantly superior to all four CP based Lagrangian variants. Nevertheless, we present a detailed interpretation for all approaches here. There are two reasons for this:

(i) Our aim is to show the effects of CP based Lagrangian relaxation when using different strategies within this setting.

(ii) Additional constraints may prevent the use of the approaches of Chapter 8, thus more flexible alternatives are required. We will discuss some of these constraints in Sec. 8.5.

7.3.1 Test Instance Generation

The tests were conducted on several sets of randomly generated instances. In order to achieve scenarios which we believe to be of relevance to the real-world application, each set of instances is generated by specifying the time horizon (half a day to 3 days) and the number of channels (20 – 100). The generator sequentially fills the channels by starting each new program one minute after the previous. For each new program a *class* is randomly chosen. That class then determines the interval from which the length is randomly chosen. We consider either 3, 5, or 7 different classes, respectively. The lengths of programs in the classes vary from 5 ± 2 minutes to 150 ± 50 minutes. The disk space necessary to store each program equals its length, and the storage capacity is randomly chosen as 45%–55% of the entire time horizon.

To achieve a complete instance, we must choose the associated profits of programs. For the experiments, we used four different strategies for the computation of an objective function:

- For the *class usefulness (CU)* instances, the associated profit values are determined with respect to the chosen class, where the associated profit values of a class can vary between zero and 600 ± 200.
- In the *time correlated (TC)* instances, each 15 minute time interval is assigned a random value between 0 and 10. Then, the profit of a program is determined as the sum of all intervals with which the program has a non-empty intersection.
- For the *weakly correlated (TWC)* instances, the TC value is perturbed by a noise of $\pm 20\%$.
- Finally, in the *subset sum (SSS)* data, the profit of a program simply equals its length.

The different objectives try to emulate some effects we believe to occur real-world instances. In the CU instances for example, programs of the same class cause similar attractions. And

the TC and TWC instances cause many conflicts regarding the choice of programs that are being broadcasted at the same time. However, the different strategies we considered are only intuitively justified. The feasibility of our approach for real-world instances can only be concluded from the fact that we achieve similar results for all choices of the objective.

We identify a test set by the parameters with which the generator was started. According to the above description those parameters are:

- time horizon in hours [6h, 12h, 24h, 72h, or 120h],
- number of channels [5ch, 20ch, 50ch, or 100ch],
- number of different classes [3, 5, or 7], and
- objective type [CU, TC, TWC, or SSS].

7.3.2 Numerical Results for Lagrangian Coupling

The numerical results presented in the following tables were ran on different test sets, each consisting out of 50 random instances. For each instance, our four approaches were run to find and prove an optimal solution. The tables present various characteristics found during the tests. We refer to Appendix A for a complete list of all results obtained. We protocol running times and the number of choice points needed for an exhaustive search. For an easier reference we start with a summary of the Lagrangian coupling approaches:

Name	Description
LG-0	Lagrangian bound, branch-and-bound, no domain filtering
LG-1	Lagrangian bound, branch-and-bound, domain filtering on original objective
LG-2	Lagrangian bound, branch-and-bound, Lagrangian relaxation based domain filtering after Lagrangian dual has been solved
LG-3	Lagrangian bound, branch-and-bound, Lagrangian relaxation based domain filtering during search for optimal Lagrangian multipliers.

7.3.2.1 Initial Solutions

Our approaches *LG-0* – *LG-3* do not use any primal heuristics. Nevertheless, they do find good solutions early on, indicating that the branching variable selection we used supports finding near-optimal solutions efficiently in a non-exhaustive search.

These observations indicate that for the ARP the majority of the work lies in the proof of optimality rather than in the construction of the solution.

In the following tables, numbers are printed in boldface when the corresponding approach was the best (according to Def. 14) among all seven approaches for the ARP (i.e. the Lagrangian approaches in this chapter as well as the two alternative approaches in Chapter 8).

5 …	*LG-0*		*LG-1*		*LG-2*		*LG-3*	
	time	**nodes**	**time**	**nodes**	**time**	**nodes**	**time**	**nodes**
CU	9.5	519.4	5.2	295.5	7.0	198.5	8.6	184.0
TC	441.8	40155.7	67.2	4525.8	18.0	696.5	28.4	575.6
TWC	15.5	1136.1	12.0	802.6	6.2	339.9	9.5	321.2

Table 7.1: *Test sets with 5 classes, 12 hours, 20 channels and different objectives. Times in seconds and choice points are averages for 50 randomly generated instances. The average number of programs per instance is between 607.6 and 612.6.*

7.3.2.2 Impact of the Objective

Table 7.1 shows the performance of all seven approaches on test sets generated with a time horizon of 12 hours and 20 channels using 5 different program classes and CU, TC, and TWC to determine the objective function.

When comparing the different types of objectives, we find that for all four *LG* approaches the TC instances are much harder than CU and TWC, which are comparably difficult to solve. This is a general observation we made for all kinds of different test sets using 3 or 7 classes as well as different time horizons and numbers of channels.

7.3.2.3 Impact on Number of Choice Points

For the Lagrangian approaches we observe that a higher degree of coupling between the two optimization constraints results in a partially drastic reduction in the number of choice points of up to a factor of about 70 in the difficult time correlated instances. Regarding the computation time, there is a trade-off between the reduction in choice points and the time spent per choice point(TpCP). The TC and TWC instances show, that *LG-2* can outperform *LG-3* because of the shorter TpCP that is needed for that degree of integration. When comparing *LG-1* and *LG-3* in the CU instances, the reduction in choice points is not large enough to justify the bigger TpCP needed, and *LG-1* is the approach that takes the least computation time.

Generally, a greater reduction in choice points is more likely to pay off when the absolute TpCP needed is rather high. This particularly holds for applications where additional constraints have to be handled on top of the objective constraint itself. For the ARP, the optimization constraint is the only active constraint. Therefore, to justify the high TpCP caused by the more complicated filtering algorithm, a noticeable reduction in the number of choice points must be achieved. The *LG-2* and *LG-3* approaches obtain a sufficient reduction in choice points in the more difficult TC test sets, and also for larger test instances.

7.3.2.4 Different Instances within Class 5 CU

Table 7.2 shows the performance of all approaches in test instances that were generated using 5 different program classes with different time horizons and numbers of channels. The objective was computed according to the chosen classes, i.e., according to CU. As expected, for the larger instances with a time horizon of 72 hours (3 days) and 20 channels, the two coupling approaches *LG-2* and *LG-3* outperform *LG-1* by a factor of roughly 4 in relation to the number of choice points and almost a factor of 2 with respect to the computation time needed. The

5 CU		LG-0 time	LG-0 nodes	LG-1 time	LG-1 nodes	LG-2 time	LG-2 nodes	LG-3 time	LG-3 nodes
	avg	*2.4*	*238.3*	*1.3*	*129.9*	*1.4*	*108.6*	*2.1*	*89.9*
12h	min	0.1	5.0	0.1	5.0	0.1	5.0	0.1	5.0
20ch	max	25.4	2216.0	15.1	1531.0	12.9	1269.0	24.2	1045.0
	std	4.2	396.8	2.3	243.6	2.5	206.5	4.0	173.5
	avg	*16.5*	*741.9*	*8.2*	*370.1*	*9.9*	*272.0*	*14.2*	*250.5*
12h	min	0.1	3.0	0.2	3.0	0.2	3.0	0.2	3.0
50ch	max	167.3	7058.0	82.6	3615.0	154.1	2664.0	156.5	2664.0
	std	30.8	1377.6	14.5	658.9	22.9	506.9	26.8	465.0
	avg	*9.5*	*519.4*	*5.2*	*295.5*	*7.0*	*198.5*	*8.6*	*184.0*
24h	min	0.5	21.0	0.4	13.0	0.3	10.0	0.5	10.0
20ch	max	87.5	4416.0	59.6	3762.0	93.4	2094.0	98.1	2067.0
	std	15.2	829.9	9.3	587.6	17.2	377.8	17.7	374.9
	avg	*1104.9*	*24301.4*	*585.2*	*14219.3*	*883.3*	*8371.9*	*921.5*	*8286.8*
24h	min	0.8	12.0	1.0	12.0	0.7	9.0	1.1	9.0
50ch	max	31045.5	675235.0	15625.2	368440.0	33281.3	292753.0	31573.5	292753.0
	std	4448.4	97121.0	2272.6	54288.6	4662.0	41139.4	4441.2	41121.2
	avg	*2627.7*	*40901.5*	*1786.7*	*27662.0*	*920.4*	*6674.7*	*990.9*	*6514.7*
72h	min	2.0	29.0	2.4	29.0	3.0	29.0	5.5	29.0
20ch	max	32751.9	460350.0	30520.3	412421.0	11766.0	90397.0	13724.7	89589.0
	std	5514.7	85325.8	4543.1	65188.4	1996.8	14515.9	2189.7	14379.4

Table 7.2: *Test sets with 5 classes, objective CU for various time horizons (in hours) and channel numbers (ch). Italic numbers give the average (avg) time and nodes, resp., of 50 instances. Numbers below are: minimum (min), maximum (max), and standard deviation (std) for these 50 instances. The average number of programs per instance is 315.2 for (12h/20ch), 793.5 (12h/50ch), 607.6 (24h/20ch), 1512.1 (24h/50ch), and 1782.6 (72h/20ch), resp.*

minimal, maximal and standard deviations indicate that the average numbers presented are not biased by very few outliers, but represent meaningful values for the evaluation of the algorithms performance.

7.3.2.5 A Vertical View

In Table 7.3 we compare the different approaches in 150 instances that were generated using very different parameters and objective functions. Again, relevant and partially significant reductions in the number of choice points can be obtained by CP based Lagrangian relaxation realized in *LG-2* and *LG-3*.

7.3.2.6 Subset-Sum Instances

So far, we have left out comparisons regarding the choice of the objective according to SSS. Table 7.4 shows the results obtained for a collection of very different test sets generated with SSS. Two facts stand out:

Firstly, a comparison with Table 7.1 shows, that the SSS instances are much easier to solve than for other choices of the objective. On the first view this result is striking, since the knapsack LP bound for strongly correlated, or subset-sum data is known to be weak, or meaningless, respectively (as discussed in section 3.7.1). However, if length of and profit for a movie coincide, any solution completely filling the disk's capacity K is already optimal and we can stop searching.

test set	LG-0		LG-1		LG-2		LG-3	
	time	nodes	time	nodes	time	nodes	time	nodes
3 CU 120h 20ch	5210.3	60839.2	1734.4	30676.4	455.9	3433.9	490.1	2945.1
5 TWC 72h 20ch	11600.1	293386.8	1526.8	35718.4	261.0	3683.6	411.7	3134.5
7 TC 24h 50ch	8349.0	250367.1	4066.3	105572.6	403.4	6235.4	533.0	4219.1

Table 7.3: *Effectiveness of the different approaches for different benchmark classes. We have an average of 1956.7 programs for the 120h/20ch test, 1782.6 for 72h/20ch, and 1423.3 for 24h/50ch.*

5 SSS			LG-0		LG-1		LG-2		LG-3	
			time	nodes	time	nodes	time	nodes	time	nodes
12h	20ch	316.4p	0.2	23.1	0.2	15.2	0.2	15.2	0.3	15.2
12h	50ch	792.4p	**0.5**	18.8	**0.5**	13.9	**0.5**	13.9	0.8	13.9
24h	20ch	611.2p	0.5	26.9	0.6	21.9	0.6	21.9	1.0	21.9
24h	50ch	1527.3p	**1.6**	29.1	1.8	23.2	1.7	23.2	3.0	23.2
72h	20ch	1778.3p	3.5	53.4	4.6	51.7	5.0	51.7	8.7	51.7
72h	50ch	4464.3p	11.0	54.4	14.4	52.8	15.4	52.8	27.2	52.8

Table 7.4: *Subset sum data sets for 12 hours, 20 channels, up to 72 hours and 50 channels. The average number of programs is given as parameter p in the upper table.*

Secondly, *LG-1* achieves only a slight reduction in choice points compared to *LG-0* which cannot be improved by *LG-2* and *LG-3* at all. The effect is not surprising: We considered the somewhat artificial SSS test sets because of their obvious relation to subset sum benchmarks for knapsack problems. Because of the equal efficiency p_i/w_i of all programs, the knapsack optimization constraint has great difficulties in including or excluding programs. Thus, that constraint is not effective, and the burden of domain reduction lies only on the MWSSP optimization constraint. In total, using the optimization constraint for pruning purposes only is most time efficient here.

7.3.2.7 Varying Time Horizon and Number of Channels

Finally, we investigate the impact of the number of channels. Table 7.5 shows a comparison of three different test sets that were generated using 3 different program classes and CU objectives. All instances have a similar size and contain roughly 1000 – 1200 programs. We observe that the instances become more difficult to solve for all Lagrangian approaches when the number of channels decreases. This surprising result may be caused by the fact that a smaller number of channels increases the relative importance of the knapsack optimization constraint which is more difficult than the MWSSP constraint. However, a reliable answer to that question can only be given by further extensive investigation. We must also note that for TC data sets, we observed a converse behavior: The instances become more difficult the more

3 CU	avg. no of programs	LG-0		LG-1		LG-2		LG-3	
		time	nodes	time	nodes	time	nodes	time	nodes
12h 100ch	1048.3	18.8	680.8	9.1	326.3	5.1	108.6	7.6	95.5
24h 50ch	1013.4	37.8	1461.1	19.5	734.4	11.1	187.9	13.8	178.5
72h 20ch	1175.0	177.4	3003.1	111.7	1897.2	36.6	468.4	42.9	401.2

Table 7.5: *Different benchmark sets for* $\approx$ **1000 – 1200** *programs for 3 classes and objective CU.*

channels are involved, which is obviously caused by many temporally conflicting programs of similar value.

7.4 Conclusions

Motivated by current research activities in the area of digital television, we formalized a simple mathematical model for the automatic recording problem (ARP).

We studied the idea of coupling (cost based) domain filtering algorithms via Lagrangian relaxation (see Sect. 2.3.3). It allows us to combine several domain filtering algorithms designed for linear optimization constraints. Thereby, we obtain effective and efficient filtering algorithms based on tight global bounds. We believe, that this idea is generic and independent of the specific application we presented on which an empirical evaluation is based. The numerical results show a significant improvement due to the coupling method with respect to the computation time and the number of choice points. The method is suitable for linear optimization problems for which bounds based on continuous or Lagrangian relaxations can be effectively used.

As an open topic in this context it remains to investigate, how general algorithms for the solution of the Lagrangian dual (such as subgradient or multiplier adjustment algorithms) must be adapted to enable domain reductions during the search for optimal Lagrangian multipliers. Taking subgradient algorithms as an example: Is it really necessary to reset the iteration limit and the step length after every domain reduction that has taken place, or can convergence still be proved for more optimistic strategies?

8

More Efficient Approaches for the Automatic Recording Problem

In the previous chapter we demonstrated how the concepts of Section 2.3.3 can be successfully applied to the ARP. A tighter Lagrangian coupling allows us to use a more effective domain filtering which then prunes the search space more effectively.

This chapter is devoted to two alternative approaches that do not rely on Lagrangian coupling ideas, and that solve the ARP significantly faster than the Lagrangian approaches. The first one is based on branch-and-cut, the second one on dynamic programming. CP is not used in these approaches but could be used at least in the branch-and-cut method. Both approaches are based on known results. Nevertheless, we present them here in order to show that the ARP can be solved a lot faster than by methods presented before.

8.1 Solving the ARP via Branch-and-Cut

A successful application of the LP-based branch-and-bound paradigm relies heavily on tight LP bounds that can be quickly computed. The LP relaxation of (7.1) is obtained by relaxing $x \in \{0,1\}^n$ to $x \in [0,1]^n$. Using some well-known results on valid inequalities, we can tighten that LP formulation, thus speeding up convergence in a branch-and-bound approach.

We rewrite the n-dimensional solution space $\mathcal{R}_{ARP}$ of the LP relaxation of (7.1) as

$$\mathcal{R}_{ARP} = \mathcal{R}_K \cap \mathcal{R}_{VC}, \quad \text{where} \tag{8.1}$$

$$\mathcal{R}_K = \{x \in [0,1]^n \mid \sum_{i=1}^{n} w_i x_i \leq K\} \tag{8.2}$$

$$\mathcal{R}_{VC} = \{x \in [0,1]^n \mid x_i + x_j \leq 1, \ \forall\, 1 \leq i < j \leq n, \ I_i \cap I_j \neq \emptyset\} \tag{8.3}$$

and obtain the LP relaxation value as

$$\max_x \{\sum_{i=1}^{n} p_i x_i \mid x \in \mathcal{R}_{ARP}\}.$$

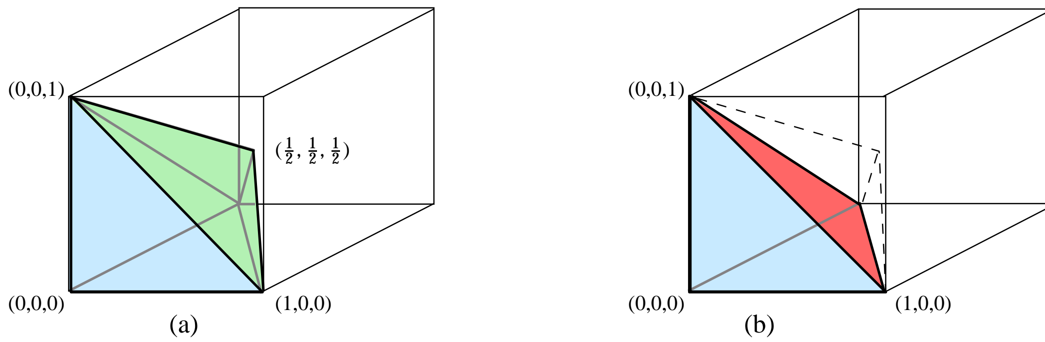


Figure 8.1: *The polytope of (a)* $\{x \in [0,1]^3 \mid x_1 + x_2 \leq 1 \ \wedge \ x_1 + x_3 \leq 1 \ \wedge \ x_2 + x_3 \leq 1\}$ *and the corresponding valid inequality (b)* $\{x \in [0,1]^3 \mid x_1 + x_2 + x_3 \leq 1\}$.

We also define the corresponding integral points sets which contain the solution of (7.1) when solved as an IP.

$$\mathcal{S}_{ARP} = \mathcal{S}_K \cap \mathcal{S}_{VC}, \quad \text{where} \tag{8.4}$$

$$\mathcal{S}_K = \mathcal{R}_K \cap \{0,1\}^n \tag{8.5}$$

$$\mathcal{S}_{VC} = \mathcal{R}_{VC} \cap \{0,1\}^n \tag{8.6}$$

The sets $\mathcal{R}_K$ and $\mathcal{R}_{VC}$ are subsets of $\mathbb{R}^n$ and can be described by a finite number of linear inequalities. Furthermore, the x-variables are bounded, and hence, $\mathcal{R}_K$ and $\mathcal{R}_{VC}$ are *polytopes*. The same holds for $\mathcal{R}_{ARP}$ and the convex hulls $\mathbf{conv}(\mathcal{S}_{ARP})$, $\mathbf{conv}(\mathcal{S}_{KP})$ and $\mathbf{conv}(\mathcal{S}_{VC})$, as they are intersections of a finite number of polytopes. (We use $\mathbf{conv}(A)$ to denote the convex hull of some finite set A.)

The *vertex cover polytope* $\mathcal{R}_{VC}$ and the *knapsack polytope* $\mathcal{R}_K$ have been studied intensively in recent years. For both polytopes valid inequalities are known that describe (parts of) the convex hull of $\mathcal{S}_{VC}$ and $\mathcal{S}_{VC}$, respectively. Since $\mathbf{conv}(\mathcal{S}_{ARP})$ is the intersection of these two polytopes, we can (partly) describe $\mathbf{conv}(\mathcal{S}_{ARP})$ by describing the convex hulls of $\mathcal{S}_{VC}$ and $\mathcal{S}_K$. We refer to the book by Nemhauser and Wolsey [156] for an introduction on the topic and for further reading on results that we only briefly present in the following sections.

8.1.1 Valid Inequalities for the Vertex Cover Polytope

The structure of the vertex cover polytope has been studied intensively during the 1970s by Fulkerson, Padberg, Nemhauser and Trotter jr. and others. Fulkerson [83] and Padberg [160] discovered the result on valid clique inequalities that we are going to use in the context of the ARP. We refer to Nemhauser and Wolsey [156] and the references given there for a deeper insight into the polyhedral structure of vertex cover.

It is known (see Grötschel et al. [99, Prop. 1.3]) that $\mathcal{R}_{VC}$ as given in (8.3) is sufficient to describe the vertex cover polytope $\mathbf{conv}(\mathcal{S}_{VC})$ of a graph G if and only if G is bipartite. We aim at getting a tighter formulation for the graph G used in the ARP.

Lemma 1 *Let* $\{C_1, \ldots, C_m\}$ *be the set of all maximal conflict cliques of an ARP. Define* $\mathcal{R}'_{VC} = \{x \in [0,1]^n \mid \sum_{i \in C_p} x_i \leq 1, \ p = 1, \ldots, m\}$. *Then*

(i) $\mathcal{R}'_{VC} \subseteq \mathcal{R}_{VC}$*, and* $\mathcal{R}'_{VC} \subset \mathcal{R}_{VC}$ *if there is a conflict clique with more than 2 members.*

(ii) $\mathcal{R}'_{VC}$ *and* $\mathcal{R}_{VC}$ *contain the same integral points.*

Proof:

(i) Let $x \in \mathcal{R}'_{VC}$. Then $\sum_{i \in C_p} x_i \leq 1,\ p = 1, \ldots, m$. Especially for any two overlapping movies m_i, m_j there exists a p such that $m_i, m_j \in C$. That is: $x_i + x_j \leq 1$, showing $x \in \mathcal{R}_{VC}$.

Let $y = (\frac{1}{2}, \ldots, \frac{1}{2}) \in \mathbb{R}^n$. Then $y \in \mathcal{R}_{VC}$ since the sum of any two entries of y is one. Let C be a largest conflict clique. We have $\sum_{i \in C} y_i = \frac{1}{2}|C| > 1 \iff |C| > 2$. Thus, $y \notin \mathcal{R}'_{VC} \iff |C| > 2$.

(ii) Let $y \in \mathcal{R}_{VC} \cap \{0,1\}^n$ and assume $y \notin \mathcal{R}'_{VC} \cap \{0,1\}^n$. That is, there is a conflict clique C such that $\sum_{i \in C} y_i > 1$. Since $y_j \in \{0,1\}$ there are at least two indices $s, t \in C$, $s \neq t$ such that $y_s = y_t = 1$. All movies indexed by C are pairwise overlapping, hence $y_t + y_s \leq 1$ is valid for $\mathcal{R}_{VC}$. This contradicts our initial assumption. Therefore $(\mathcal{R}_{VC} \cap \{0,1\}^n) \subseteq (\mathcal{R}'_{VC} \cap \{0,1\}^n)$. Applying *(i)* now completes the proof. ∎

Though finding maximum cliques is NP-hard on general graphs (Garey and Johnson [85]), it is simple on the graph defined by our application: We can record two movies m_i, m_j only if they do not overlap in time. The graph defined by the broadcast contains arcs from node i to node j if and only if $I_i \cap I_j = \emptyset$. Complements of theses graphs are known as interval graphs:

Definition 11 *A graph* $G = (V, E)$ *is called an* interval graph *iff intervals* $I_1, \ldots, I_{|V|} \subset \mathbb{Q}$ *exist such that* $\forall\ v_i, v_j \in V : \{v_i, v_j\} \in E \iff I_i \cap I_j \neq \emptyset$.

Definition 12 *A graph* $G = (V, E)$ *is called a* perfect graph *if* $\omega(G(H)) = \chi(G(H))$ *for each* $H \subseteq V$. *I.e. the chromatic number* $\chi(G(H))$ *equals the size of a maximum clique* $\omega(G(H))$.

Remark 1 *Interval graphs as well as their complements belong to the family of* perfect graphs *(see Golumbic [94]).*

Many hard problems are rather easy to solve on perfect graphs. On interval graphs, in particular, the computation of all maximal cliques can be performed in time $\Theta(n \log n)$ using a simple scan-line algorithm (Gupta et al. [103]). After sorting the nodes according to increasing starting times the algorithm pushes a node into a priority queue H when the node's interval starts, and removes it when the interval ends. Just before a node is removed from H, all nodes in H form a maximal clique. Hence, if we have to report all cliques (rather than only detecting them), we need time $O(n^2)$ to print $O(n)$ times the content of H.

Using $\mathcal{R}'_{VC}$ thus gives tighter bounds when considering LP-based solution approaches (see Figure 8.1 for an example). Consequently, we reformulate (7.1) by using conflict cliques as

$$\begin{array}{llcll} \text{Maximize} & \sum_{i=1}^n p_i x_i & & & \\ \text{subject to} & \sum_{i \in C_p} x_i & \leq & 1 & p = 1, \ldots, m \\ & \sum_{i=1}^n w_i x_i & \leq & K & \\ & x & \in & \{0,1\}^n & \end{array} \tag{8.7}$$

and obtain the following

Corollary 3 *IP (8.7) can be formulated in polynomial time. It contains $|V|$ conflict cliques at most.*

Finally, we note that for the vertex cover polytope, a valid inequality derived from a conflict clique C is *facet defining* if and only if C is maximal (see Nemhauser and Wolsey [156] for a simple proof). Facet defining inequalities are the most helpful ones when looking for integral solutions: They describe parts of the convex hull of feasible integral points.

On general graphs, conflict cliques do not describe the convex hull $\mathrm{conv}(\mathcal{S}_{VC})$ completely. Fortunately, perfect graphs play a different role in this context:

Lemma 2 (Grötschel et al., 1986) *The inequalities $\sum_{i \in C_p} x_i \leq 1$ for all maximal cliques $C_1, \ldots, C_p$ of G are sufficient to describe the vertex cover polytope if and only if G is perfect.*

Proof: see Grötschel et al. [99], Theorem 1.5. ∎

8.1.2 Valid Inequalities for the Knapsack Polytope

Next, we will tighten the polytope $\mathcal{R}_K = \{x \in [0,1]^n \mid \sum_{i=1}^n w_i x_i \leq K\}$ given in (8.2). We use a well-known result on the knapsack polytope that was published by several authors: Balas [13], Hammer et al. [104], Padberg [161], Wolsey [210]. Our presentation here partly follows Nemhauser and Wolsey [156] and Johnson et al. [123].

Without loss of generality, we assume $w_i > 0$ ($w_i = 0$ can be ignored and if $w_i < 0$ we can replace x_i by $1 - x_i$), and $w_i \leq K$ (as $w_i > K$ implies $x_i = 0$).

Definition 13 *A set $C \subseteq \{1, \ldots, n\}$ is called a* minimal cover *if $\sum_{i \in C} w_i > K$ and for all $j \in C$: $\sum_{i \in C \setminus \{j\}} w_i \leq K$*

Let C be a minimal cover. From those variables that are indexed by C, $|C| - 1$ variables at most can have the value one, which allows us to derive the valid inequality

$$\sum_{i \in C} x_i \leq |C| - 1 \tag{8.8}$$

In fact, (8.8) is not only a valid inequality. On a $|C|$-dimensional subspace of $\{0,1\}^n$ it is as tight as possible. To be more precise: It is facet defining for the convex hull of the $0-1$ knapsack set defined by

$$\{x \in \{0,1\}^n \mid \sum_{i \in C} w_i x_i \leq K, \ x_j = 0 \Leftrightarrow j \notin C\} \tag{8.9}$$

(see Cor. 2.4, p. 267 in [156]).

Figure 8.2 shows an example on a polytope that was tightened by a cover inequality. Unfortunately, there may be $O(2^n)$ different minimal covers and we cannot generate them all in a preprocessing step as we did for $\mathcal{R}'_{VC}$. We can, however, generate those inequalities that are violated by the current LP solution and add them to our LP in order to strengthen it.

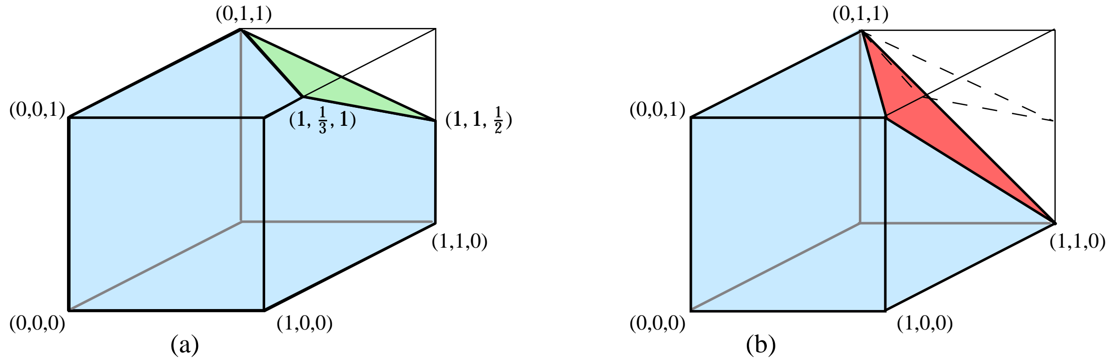


Figure 8.2: *The polytope of (a)* $\{x \in [0,1]^3 \mid 2x_1 + 3x_2 + 4x_3 \leq 7\}$ *and the tighter representation (b)* $\{x \in [0,1]^3 \mid 2x_1 + 3x_2 + 4x_3 \leq 7 \ \wedge \ x_1 + x_2 + x_3 \leq 2\}$.

8.1.2.1 Generating Violated Inequalities

Let $\overline{x} = \arg\max\{\sum_{i=1}^{n} p_i x_i \mid \sum_{i=1}^{n} w_i x_i \leq K, \ x \in [0,1]^n\}$ be an optimal linear programming solution to the knapsack problem. If the LP solution $\overline{x}$ satisfies a minimal cover C then $\sum_{i\in C} \overline{x_i} \leq |C| - 1$, which means $\sum_{i\in C}(1 - \overline{x_i}) \geq 1$. Vice versa, if for some minimal cover $C \subseteq \{1,\ldots,n\}$ it holds $\sum_{i\in C}(1 - \overline{x_i}) < 1$, then the cover inequality corresponding to C is violated. This gives rise to an 'automatic' detection of a violated cover inequality via the following linear integer program:

$$\begin{array}{ll} \text{Minimize} & \sum_{i=1}^{n}(1-\overline{x_i})y_i \\ \text{subject to} & \sum_{i=1}^{n} w_i y_i > K \\ & y \in \{0,1\}^n \end{array} \tag{8.10}$$

If the optimal value of (8.10) is smaller than one, $C = \{i \mid y_i = 1\}$ is the minimal cover most violated by the current solution $\overline{x}$ of the original LP. We are not helped if we have to solve the separation (8.10) via an exact approach, since (8.10) is of the same computational complexity as (8.7). As we do not need an exact solution of (8.10) (any violated minimal cover is fine), and also as we do not require all inequalities to be found (we can always branch on a fractional variable), (8.10) is usually solved via some greedy heuristic.

As mentioned above, (8.8) is facet defining only for the subspace given in (8.9). In order to obtain a facet-defining inequality for the original space, we have to lift (8.8) to the n-dimensional space.

Lifting is done by introducing a single variable not already included in the inequality and adjusting its coefficient such that a new valid inequality is obtained, that is facet defining on the next higher dimensional space. This procedure is performed until all n variables have been taken into account. A formal description and proof of this technical procedure is given in Nemhauser and Wolsey [156, pp. 261–265], see also Gu et al. [100, 101] for details on efficient computational handling. The result of lifting (8.8) to the n-dimension space is

$$\sum_{i\in C_1} x_i + \sum_{i\in\{1,\ldots,n\}\setminus\{C_1\cup C_2\}} \alpha_i x_i + \sum_{i\in C_2} \gamma_i x_i \ \leq \ |C| - 1 + \sum_{i\in C_2} \gamma_i \tag{8.11}$$

where $C_1 = \{i \in C \mid \overline{x_i} < 1\}$, $C_2 = C \setminus C_1$, and coefficients α_i, γ_j are determined during lifting.

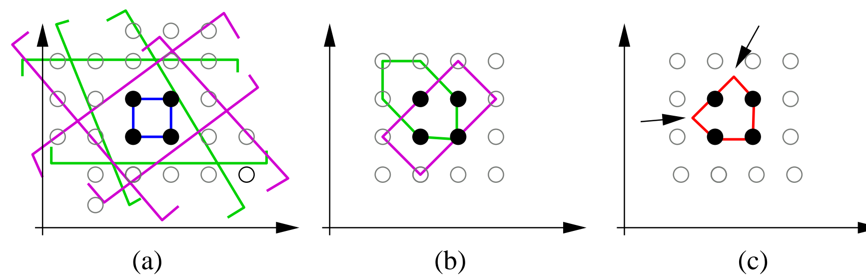


Figure 8.3: *A well-known problem: Extreme points may occur in the intersection of polytopes. (a) shows the initial LP inequalities (green/magenta) of two substructure and the convex hull of feasible integer points (blue). Using tight cuts for the two substructures we find the convex hull of either substructure (b). When intersecting these regions (c), new fractional extreme points occur (arrows).*

8.1.2.2 An Integer Linear Programming Approach

Modern Branch-and-Cut systems (see e.g. Bixby [35], Elf et al. [69], Johnson et al. [123], Martin [147]) are well-suited to solve the model described above. Indeed, we obtained very good numerical results for the ARP using a branch-and-cut approach based on CPLEX 7.1 [118]. We used the strong formulation (8.7) for the ARP, where the maximal cliques were determined via a scan-line algorithm. That model was then transferred to CPLEX.

CPLEX provides heuristics for automatically separating and lifting violated cover inequalities. These cuts are generated and added to the model whenever CPLEX considers it helpful. Furthermore, CPLEX utilizes powerful primal heuristics that find near optimal solutions to the ARP model in the root node of the branch-and-bound-tree. In most cases an optimal solution to the ARP was found and proved within a few seconds.

Some benchmark instances required long running times and — since CPLEX applied best-bound-search at default — huge amounts of memory. In the experiments, we therefore limited the available memory to 128 MB. After reaching that limit, we switched the node selection strategy to depth-first-search, and allowed a further $500\,000$ branch-and-bound nodes to be explored. After reaching that limit, we stopped the search without proving optimality. The latter happened in eleven out of 4150 benchmark tests.

One problem that is partly responsible for these long running times is the fact, that not all fractional extreme points in the intersection of two polytopes are eliminated by tight cuts on either polytope (see Fig. 8.3). When intersecting the two sets, new fractional extreme points may occur. If many of these new points appear in the area of an optimal integer solution, the LP relaxation will find these instead and additional branching is required to eliminate them. This problem is well-known in IP theory.

Before discussing the numerical results in detail, we define the second approach.

8.2 Solving the ARP via Dynamic Programming

Solution approaches for the ARP can also be defined via dynamic programming. The key to two dynamic programming approaches is again the ARP's knapsack substructure. Our dynamic programs differ only slightly from well-known algorithms for knapsack problems (see e.g. Papadimitriou and Stieglitz [163, Sec. 17.3]). One of the approaches was presented

in Sellmann and Fahle [186]. It runs in $O(n^2 p_{max})$, where $p_{max} := \max\{p_i \mid i = 1, \ldots, n\}$, and it requires space $O(n^2 p_{max})$. Furthermore, an FPTAS can be derived easily from that dynamic program (see [186]).

We define an alternative approach in the following. This one needs $O(nK)$ space, and runs in time $O(cnK)$, where c is the number of channels. Notice that typically $c < n$ for the ARP, as each channel will broadcast several movies over some time period.

In our application scenario a running time of $O(cnK)$ and a memory requirement of $O(nK)$ seems to be more desirable than the previously mentioned $O(n^2 p_{max})$. Parameters K and c, as well as p_{max} can be considered as fixed in the application scenario — p_{max} is limited by the measure for users' satisfaction, K is bounded by the VCR's hard disk capacity, and c is the number of channels provided. Thus, for n increasing, $O(cnK)$ scales linearly, whereas the $O(n^2 p_{max})$ approach depends quadratically on n.

8.2.1 A Simple Approach

We start with the basic definition of the new approach. Let $\delta(i, k)$ denote an optimal solution to an ARP that has disk size $k \le K$, and that considers the first $i \le n$ movies at most, i.e.

$$\begin{aligned} \delta(i,k) \quad = \max\{ & \textstyle\sum_{j=1}^{i} p_j x_j \mid \sum_{j=1}^{i} w_j x_j \le k, \\ & (x_r = 1 \wedge x_q = 1) \Rightarrow I_r \cap I_q = \emptyset, \ \forall\ r \ne q \le i \\ & x \in \{0,1\}^n\} \end{aligned} \tag{8.12}$$

We assume that $\{start(i), end(i) \mid i = 1, \ldots, n\}$ contains exactly $2n$ elements, i.e. all times are unique. We order all movies according to increasing starting times. To ease the notation we add two artificial movies "0" and "-1" to V. Both end before the earliest movie m_1 starts, i.e. $end(-1) < end(0) < start(m_1)$. For $i \in V$ we define

$$\mathcal{T}(i) = \{\ell \in V \mid end(\ell) < start(i)\} \tag{8.13}$$

as the set of movies that finish before movie i starts. Now, $\delta(i, k)$ can be computed by the following recursion:

$$\delta(i,k) = \begin{cases} -\infty, & k < 0 \\ 0, & i = -1, 0 \text{ and } k \ge 0 \\ \max\limits_{j \in \mathcal{T}(i)} \{\delta(j,k),\ \delta(j, k - w_i) + p_i\}, & else \end{cases} \tag{8.14}$$

This recursion is almost identical to the classical recursion for knapsack problems (see Martello and Toth [145]). The only difference is that (8.14) checks the Bellman-equation for all non-overlapping movies in $\mathcal{T}(i)$. Dynamic program (8.14) requires space $O(nK)$, and can be computed in time $O(n^2 K)$. The latter results from the fact that for every $\delta(i, k)$ we have to consider all j such that $end(j) < start(i)$.

8.2.2 An Improved Dynamic Program

In the ARP, only one of the c channels can be selected at each time step. Instead of iterating over *all* programs j that finish before movie i starts, it suffices to check the current program

j of *each channel*. Thus, through some small changes, we can replace n by c in the running time for (8.14).

For each $i \in V$ we consider all movies that end before movie i starts. Let $\sigma(i)$ denote the movie that starts latest among all these movies, and let $\mathcal{D}(i)$ denote the set of movies ending between the start of $\sigma(i)$ and the start of i. In other words,

$$\sigma(i) = \arg\max_j \{start(j) \mid end(j) < start(i)\} \tag{8.15}$$

$$\mathcal{D}(i) = \{j \mid start(\sigma(i)) < end(j) < start(i)\} \tag{8.16}$$

These values can be calculated by a simple $O(cn + n\log n)$ preprocessing step. Notice that

$$\mathcal{T}(i) = \mathcal{D}(i) \cup \mathcal{T}(\sigma(i)) \quad \text{and} \quad \mathcal{D}(i) \cap \mathcal{T}(\sigma(i)) = \emptyset \tag{8.17}$$

In order to improve on the simple dynamic program (8.14) we need to consider movies $j \in \mathcal{D}(i)$, rather than $j \in \mathcal{T}(i)$:

$$\mu(i,k) = \begin{cases} -\infty, & k < 0 \\ 0, & i = -1, 0 \text{ and } k \geq 0 \\ \max\limits_{j \in \mathcal{D}(i)} \{\mu(j,k),\ \mu(j,k-w_i) + p_i\}, & else \end{cases} \tag{8.18}$$

In comparison to (8.14), the space requirement is not affected by our small change, whereas the running time is improved:

Lemma 3 *The dynamic programs (8.14) and (8.18) compute the same values for a state* (i,k), $i \in V$, $k = 1,\ldots,K$. *The running time of (8.18) (after preprocessing) is* $O(cnK)$, *where* c *denotes the number of different broadcast channels.*

Proof: First, we prove $(*)$: $\bigcup_{j \in \mathcal{T}(i)} \mathcal{T}(j) = \bigcup_{j \in \mathcal{D}(i)} \mathcal{T}(j)$.

"$\supseteq$" Because of (8.17) it holds $\bigcup_{j \in \mathcal{T}(i)} \mathcal{T}(j) \supseteq \bigcup_{j \in \mathcal{D}(i)} \mathcal{T}(j)$.

"$\subseteq$" Now, select $\alpha \in \bigcup_{j \in \mathcal{T}(i)} \mathcal{T}(j) \Rightarrow \exists j \in \mathcal{T}(i) :\ end(\alpha) < start(j) < end(j) < start(i)$. Especially, as $\forall j \in T(i) : start(j) \leq start(\sigma(i))$ we obtain $end(\alpha) < start(\sigma(i))$. Since $\sigma(i) \in \mathcal{D}(i)$ it follows $\alpha \in \bigcup_{j \in \mathcal{D}(i)} \mathcal{T}(j)$.

We prove correctness of the lemma by induction. For $i = -1$ and for $i = 0$ it holds: $\mu(i,k) = \delta(i,k)$. Our hypothesis is that we have shown $\mu(i',k) = \delta(i',k)$ for all movies i' that finish before a certain movie $i \in V \setminus \{-1, 0\}$ starts. We have to show, that equality also holds for i, i.e. $\mu(i,k) = \delta(i,k)$:

$$\mu(i,k) = \max_{j \in \mathcal{D}(i)} \{\mu(j,k),\ \mu(j,k-w_i) + p_i\} \tag{8.19}$$

It holds $start(j) < end(j) < start(i)\ \forall j \in \mathcal{D}(i)$ and we can apply our inductive hypothesis

$$= \max_{j \in \mathcal{D}(i)} \{\delta(j,k),\ \delta(j,k-w_i) + p_i\} \tag{8.20}$$

We apply the definition of $\delta(j,k)$.

$$= \max_{j \in \mathcal{D}(i)} \left\{ \begin{array}{l} \max\limits_{\ell \in \mathcal{T}(j)} \{\delta(\ell,k),\ \delta(\ell,k-w_j)+p_j\}, \\ \max\limits_{\ell \in \mathcal{T}(j)} \{\delta(\ell,(k-w_i)),\ \delta(\ell,(k-w_i)-w_j)+p_j\}+p_i \end{array} \right\} \tag{8.21}$$

Using $(*)$ gives

$$= \max_{j \in \mathcal{T}(i)} \left\{ \begin{array}{l} \max\limits_{\ell \in \mathcal{T}(j)} \{\delta(\ell,k),\ \delta(\ell,k-w_j)+p_j\}, \\ \max\limits_{\ell \in \mathcal{T}(j)} \{\delta(\ell,(k-w_i)),\ \delta(\ell,(k-w_i)-w_j)+p_j\}+p_i \end{array} \right\} \tag{8.22}$$

$$= \max_{j \in \mathcal{T}(i)} \{\delta(j,k),\ \delta(j,k-w_i)+p_i\} \tag{8.23}$$

$$= \delta(i,k) \tag{8.24}$$

The running time of (8.18) is $O(K(\sum_{i\in V}|\mathcal{D}(i)|))$. From its definition in (8.16) $\mathcal{D}(i)$ collects movies that start before and end after time $start(\sigma(i))$. As each channel can only broadcast one movie at a time it follows $|\mathcal{D}(i)| \leq c$, and we obtain $O(K(\sum_{i\in V}|\mathcal{D}(i)|)) = O(K|V|c)$. ∎

8.3 Results

Numerical evaluations on the three methods discussed above were performed in the same setting as those in the previous chapter. I.e. we used the same computer hardware (Athlon 600 MHz PC) and the same benchmark files. In this section we present selected tables that correspond to those presented in the previous chapter. Thus, runtime for branch-and-cut and dynamic programming is directly comparable to the runtime for the Lagrangian coupling approaches. We use the following abbreviations:

Name	Description
BC	CPLEX based branch-and-bound, using clique and cover inequalities.
DP	dynamic program (8.14), running in $O(n^2K)$
*DP**	dynamic program (8.18), running in $O(cnK)$

8.3.1 Numerical Results for Branch-and-Cut and Dynamic Programming

8.3.1.1 Initial Solutions

Using rounding heuristics, *BC* always finds a first solution in the root node. Typically, that solution is less than 0.5% away from the optimal solution value. (Tables are given in Appendix A). Even when running *BC* without primal heuristics, the first solution is found early by the first deep dive into the search tree — a technique often used in branch-and-bound. In general, the overall running time is only slightly affected by turning off heuristics.

5 ...	DP		DP*		BC	
	avg.	(max) time	avg.	(max) time	avg.	(max) time
CU	1.16	(1.67)	**0.08**	(0.11)	0.21	(1.59)
TC	1.20	(1.86)	**0.08**	(0.13)	**0.10**	(0.28)
TWC	1.16	(1.72)	**0.08**	(0.11)	**0.10**	(0.28)

Table 8.1: *Test sets with 5 classes, 12 hours, 20 channels and different objectives. Times in seconds and choice points are averages for 50 randomly generated instances. The average number of movies per instance is between 607.6 and 612.6.*

8.3.1.2 Impact of the Objective

Table 8.1 shows the performance of *BC*, *DP*, and DP^* on test sets generated with a time horizon of 12 hours and 20 channels using 5 different program classes and CU, TC, and TWC to determine the objective function.

In the *BC* approach the running time varies slightly. The ranking given in Table 8.1 is typical also of other benchmark instances: Class CU has a longer running time than the other classes. In fact, the only runs that were interrupted (because more than $500\,000$ branch-and-bound nodes were needed) occurred on the 20h and 50h CU instances.

Our dynamic programs are not at all affected by the characteristics of the objective. Their running time only depends on the values of K, n, and c. This behavior obviously appears on all instances.

8.3.1.3 Different Instances within Class 5 CU

Table 8.2 shows the performance of *BC*, *DP*, and DP^* test instances using 5CU. Also on these instances, *DP*, DP^*, and *BC* are significantly superior (compared to Table 7.2), especially when size increases. The theoretical factor between runtime of *DP* and DP^* is $O(\frac{c}{n})$, and indeed, DP^* is significantly faster than *DP*. However, runtime is also affected by various compiler and machine dependent effects. E.g. we expect DP^* to have an increased cache hit rate compared to *DP* as the latter has to check *all* movies that finish before the current one. On the other hand, data-structures for DP^* are less regular. Thus, the concrete factor is deviating from its theoretical value when changing the instances.

In 5CU, *BC* is less effective than the best dynamic program. This is often the case for the CU instances, whereas for the other three classes (TC, TWC, SSS), *BC* may be slightly better than DP^*, though their ranking is quite similar.

8.3.1.4 A Vertical View

In Table 8.3 we compare the different approaches on 150 instances that were generated using very different parameters and objective functions. Even the slowest dynamic program (*DP*) still beats the fastest Lagrangian approach in one case, and DP^* outperforms them all by a time factor of 32, 39, or 171, respectively (compared to Table 7.3). *BC* prevails in this comparison. When supported by its primal heuristics, CPLEX finds optimal solutions already in the root node in most cases. In the remaining nodes, it basically performs a proof of optimality. *BC* is 1.5 – 7.5 times faster than DP^* in these sample sets.

5 CU		DP time	DP* time	BC time	BC nodes
	avg	*1.16*	***0.08***	*0.21*	*37.82*
12h	min	0.69	0.05	0.06	0.00
20ch	max	1.67	0.11	1.59	484.00
	std	0.20	0.01	0.28	99.55
	avg	*8.44*	***0.44***	*1.08*	*103.98*
12h	min	6.34	0.35	0.36	0.00
50ch	max	11.09	0.58	9.11	1312.00
	std	1.13	0.05	1.41	240.16
	avg	*10.28*	***0.47***	*1.38*	*234.96*
24h	min	8.06	0.39	0.18	0.00
20ch	max	13.48	0.57	43.64	7618.00
	std	1.08	0.04	6.06	1074.40
	avg	*70.46*	***2.40***	*176.54*	*8851.52*
24h	min	56.41	1.94	0.86	0.00
50ch	max	87.90	3.11	4357.75	235195.00
	std	7.36	0.26	715.71	37095.52
	avg	*295.83*	***6.60***	*516.96*	*52576.38*
72h	min	256.36	5.69	0.54	0.00
20ch	max	360.12	7.86	7650.35	1498720.00
	std	24.21	0.47	1658.57	223318.36

Table 8.2: *Test sets with 5 classes, objective CU for various time horizons (in hours) and channel numbers (ch). Italic numbers give the average (avg) time and nodes, resp., of 50 instances. Numbers below are: minimum (min), maximum (max), and standard deviation (std) for these 50 instances. The average number of movies per instance is 315.2 for (12 h/20 ch), 793.5 (12 h/50 ch), 607.6 (24 h/20 ch), 1512.1 (24 h/50 ch), and 1782.6 (72h/20 ch), resp.*

test set	DP	DP*	BC	
	avg. (max) time	avg. (max) time	avg. (max) time	avg. (max) nodes
3 CU 120h 20ch	630.21 (723.36)	14.21 (15.83)	**1.92** (7.56)	18.28 (180.00)
5 TWC 72h 20ch	297.63 (364.60)	6.62 (7.85)	**0.97** (5.41)	48.32 (785.00)
7 TC 24h 50ch	62.51 (76.67)	2.35 (2.86)	**1.51** (4.88)	32.18 (370.00)

Table 8.3: *Effectiveness of the different approaches for different benchmark classes. We have an average of 1956.7 programs for the 120h/20 ch test, 1782.6 for 72h/20 ch, and 1423.3 for 24 h/50 ch.*

5 SSS		*DP* avg. (max) time	*DP** avg. (max) time	*BC* avg. (max) time	*BC* avg. (max) nodes
12h	20ch	1.16 (1.55)	**0.08** (0.10)	0.15 (0.41)	6.60 (63.00)
12h	50ch	8.45 (10.62)	**0.44** (0.68)	0.62 (2.32)	7.88 (74.00)
24h	20ch	10.41 (12.85)	0.46 (0.57)	**0.37** (1.23)	11.08 (89.00)
24h	50ch	72.54 (93.84)	2.42 (2.92)	**1.62** (6.85)	9.34 (78.00)
72h	20ch	290.84 (347.07)	6.46 (7.60)	**1.33** (3.98)	7.00 (74.00)
72h	50ch	2232.33 (2581.61)	32.54 (37.04)	**6.84** (32.29)	9.94 (89.00)

Table 8.4: *Subset sum data sets for 12 hours, 20 channels, up to 72 hours and 50 channels.*

3 CU	*DP* avg. (max) time	*DP** avg. (max) time	*BC* avg. (max) time
12h 100ch	14.63 (16.91)	**0.94** (1.09)	1.40 (3.22)
24h 50ch	30.32 (36.25)	1.47 (2.05)	**1.03** (2.87)
72h 20ch	127.99 (148.24)	4.29 (4.98)	**1.04** (3.12)

Table 8.5: *Different benchmark sets for* $\approx 1000 - 1200$ *programs for 3 classes and objective CU.*

8.3.1.5 Subset-Sum Instances

The comparison on SSS date is given in Table 8.4. For *BC* potentially weak LP bounds on SSS data are compensated by cover inequalities that improve the LP relaxation. Martello and Toth have observed these effects in [146] for cardinality constraints which are a restricted form of cover inequalities.

A direct comparison to the corresponding Table 7.4 shows that on SSS instances the Lagrangian approaches are somewhat competitive. I.e. for 12 h/50 ch and for 24 h/50 ch they run as fast as the fastest of the formerly prevailing approaches. They even beat DP^* on larger instances. Still, they cannot compete against *BC*. We believe that the *BC* approach benefits from its primal heuristics here, and that a combination of a primal heuristic and *LG-0* or *LG-1* would be a good alternative to the branch-and-cut approach on SSS data.

8.3.1.6 Varying Time Horizon and Number of Channels

Finally, in Table 8.5 we investigate the impact of the number of channels for a sample set containing about 1000 – 1200 programs each. For *DP* and DP^* the increase in running time follows the increase in the disk's capacity. As mentioned in Sec. 7.3.1, we choose K as 45% – 55% of the entire time horizon. The *BC* approach is not really affected by changed parameters. The largest computing time of 1.40 sec in the 12 h/100 ch case is produced by some longer preprocessing times. In fact, the number of branch-and-bound nodes is comparable to the other two samples (see tables in Appendix A).

8.3.1.7 Robustness within a Test Set

All benchmark data presented in the last two chapters are average values of a sample of 50 individual random instances. As indicated already in Table 8.2 (and in the appendix) run time is smooth for *DP* and DP^*, whereas min, max, and standard deviation for the other approaches show some variation within each test set. Figure 8.4 gives the run time for each individual instance in the set 5CU, 24 hours, 50 channels for all approaches considered. Both dynamic programs basically have the same run time in each instance, and vary only slightly. *BC* needs less time than the best dynamic program in many case. However, some few long-running test sets negatively impact on the average runtime, and *BC* comes second in this competition.

For the *LG* instances we have much more variation in runtime than for *BC*. Noteworthy, the *LG* approaches react similarly, i.e. in many cases a high or a low runtime occurs for all approaches simultaneously. We assume that primal heuristics in *BC* and different node selection strategies for *BC* and *LG* cause this striking difference. Whereas CPLEX finds a good solution in the root node already, and then applies best-first-search, the *LG*-approaches use depth-first-search. The latter requires long running times if a good solution is not found in the left part of the search tree.

It should be noted that the plots are representative: Except for some SSS instances, where figures tend to be more smooth, all other runtime plots show about the same behavior as Figure 8.4.

8.3.2 Comparing the Approaches

For the ARP we used 4150 benchmark instances grouped in 83 classes. When comparing the approaches developed in this and in the previous chapter we find that *BC* is among the fastest on 50 benchmark classes, DP^* wins on 35 (with respect to our runtime measure, see Sec. 2.4.3). In 7 classes *LG-2* is among the best approaches, followed by *LG-0* and *LG-1* each being among the best in 5 classes (see tables in Appendix A). Notice, that the runtime is often less than half a second, when the *LG*-approaches are among the best results, whereas for harder and long running instances they never excel. Furthermore, the *LG*-approaches cannot finalize 8 out of 83 benchmark classes. This clearly shows that the ARP as defined in Def. 9 should be solved by branch-and-cut or dynamic programming. Quantitative differences in runtime (up to a factor of more than 200) are documented in Appendix A.

8.4 Conclusion

For the ARP, constraint programming based Lagrangian relaxation is clearly beaten by dynamic programming and branch-and-cut in three out of four benchmark categories. Only for the somewhat artificial SSS data is the runtime for the CP approach sometimes competitive. While this does not in general question the idea of improving domain filtering via Lagrangian coupling, it shows that without additional complex constraints, the ARP should not be solved via reduction techniques + bounding alone. As we have seen in the previous chapter, using CP based Lagrangian relaxation is much better than not using it, but the clear structure of the ARP allows for more efficient algorithms to be used.

On the other hand, the CP approach finds near optimal solutions quickly, thus allowing us

to work with heuristics solutions rather than optimal ones. Furthermore, the approach does not require large memory resources. As an advantage over branch-and-cut, the CP technology for the ARP is less complex. Finally, among the three approaches tested, the CP approach is open to non-linear constraints, whereas branch-and-cut or dynamic programming can only handle subclasses of non-linear constraints.

8.4.1 The Branch-and-Cut Approach

Together with the dynamic programming approach branch-and-cut is the fastest method in almost all instances. Furthermore, it can provide a very good (often optimal) solution in the root node. Without affecting the structure of valid inequalities used, additional linear constraints can be added to the model. These include the extensions mentioned in Sec. 8.5.

Branch-and-cut requires a state-of-the-art CPU with high precision floating point support. In some cases a huge amount of memory is required, too. It is doubtful that a typical digital VCR would satisfy these requirements in the near future. Furthermore, efficient branch-and-cut systems, and numerically stable LP solvers are highly complex technologies that may be too expensive for electronic mass products.

Valid inequalities for the vertex cover and the knapsack polytope were used to tighten the LP relaxation. Unfortunately, even if all fraction extreme points in either of the two polytopes are eliminated, there fractional extreme points in the intersection of both polytopes may exist. Further studies could concentrate on how to eliminate these fractional points efficiently by using additional cuts.

8.4.2 The Dynamic Programming Approach

The improved dynamic program is able to solve many instances to optimality significantly faster than any other approach. It is not affected by any data characteristics, and its runtime is predictable. The dynamic program depends on the disk size K, both in runtime as well as in memory consumption. During our tests, up to 120 MB of RAM were needed by the dynamic program.

For dynamic programs in the digital VCR setting there is a trade-off between the accuracy of K and the runtime needed. Using only approximated values for K requires only few states in the dynamic program, but does not guarantee an optimal solution. On the other hand, if K is very accurate (which means K is large), runtime is negatively affected. Label dominance approaches (see Desrochers and Soumis [64]) might be the right choice here as they eliminate useless states during calculation, thereby reducing runtime and memory requirements.

8.5 Extending the Basic Model

Several extensions of the basic ARP model are possible on the application level. We present three of them here and show how they can be integrated into the Lagrangian coupling, or the *BC* approach. We have not investigated how to integrate the extensions into dynamic programs.

8.5.1 Avoiding Multiple Copies

A user usually does not want to have the same movie recorded several time (when repeated or offered by different channels). This requirement is easy to integrate into the linear program (7.1) by adding $x_i + x_j \leq 1$ whenever the content of movies i and j is identical, $i, j \in V$. We can do better by using the idea of conflict cliques again. After some appropriate ordering, we can assume that indices $1, \ldots, r$ correspond to pairwise different broadcasts, and that any movie $i = r+1, \ldots, n$ is already contained in the first r movies. We define

$$\mathcal{M}_i = \{j \mid \text{movies } i, j \text{ are identical }, \ j = 1, \ldots, n\}, \quad i = 1, \ldots, r \tag{8.25}$$

as the representative classes for each movie $i = 1, \ldots, r$. From each class $\mathcal{M}_i$, $i = 1, \ldots, r$, one broadcast at most can be recorded. Thus, the indices in $\mathcal{M}_i$ correspond to a valid conflict clique. Furthermore, $\mathcal{M}_i$ is maximal, and the resulting valid inequality is facet-defining. We obtain

$$\begin{array}{llcll} \text{Maximize} & \sum_{i=1}^{n} p_i x_i & & & \\ \text{subject to} & \sum_{i \in C_p} x_i & \leq & 1 & p = 1, \ldots, m \\ & \sum_{j \in \mathcal{M}_i} x_j & \leq & 1 & i = 1, \ldots, r \\ & \sum_{i=1}^{n} w_i x_i & \leq & K & \\ & x & \in & \{0,1\}^n & \end{array} \tag{8.26}$$

The branch-and-cut approach can cope directly with the additional constraints. Both families of conflict cliques tighten the LP relaxation. The number of additional constraints is low as $r \leq n$. It is obvious see that the additional computational effort for finding these sets is $O(n \log n)$ when using a priority queue and the content id as the key.

It should be mentioned that an even tighter LP relaxation can be obtained by combining both families of valid inequalities. This however, requires some more computational effort in the preprocessing, and the number of valid inequalities will also grow.

The sets $\mathcal{M}_i$ can also be used for filtering in the Lagrangian approaches. We can define additional artificial intervals for the n movies: To each $j \in \mathcal{M}_i$ we assign the interval $[3 \cdot i, 3 \cdot i + 1]$, $i = 1, \ldots, r$. Thus, we can apply the MWSS constraint used in Sec. 7.2 in this context as well. A CP based Lagrangian relaxation then copes with three families of constraints rather than two.

8.5.2 Using Multiple Recording Units

A digital video recorder may have multiple recording units which allow the recording of a limited number b of channels simultaneously. Again, in an IP context, this modification can be easily introduced. We change the right hand side of each non-overlapping constraint from 1 to b:

$$\begin{array}{llcll} \text{Maximize} & \sum_{i=1}^{n} p_i x_i & & & \\ \text{subject to} & \sum_{i \in C_p} x_i & \leq & b & p = 1, \ldots, m \\ & \sum_{i=1}^{n} w_i x_i & \leq & K & \\ & x & \in & \{0,1\}^n & \end{array} \tag{8.27}$$

For the Lagrangian coupling approaches, a fast and efficient domain filtering algorithm for this type of relaxed non-overlapping constraint is subject to further research.

8.5.3 Using Multiple Storage Units

Finally, our VCR may have d hard disks instead of just one. As we can split a video stream at some point, and store it in parts on different disks, this specification does not require further attention. Things change, if hard disks are available on d *individual recorders*, and a global plan for all machines is required. We can model this situation by using multi-knapsack constraints (see [145]) that respect individual disk capacities K_h, $h = 1, \ldots, d$. Furthermore, multi-knapsacks ensure storing a movie i on at most one recorder. Again, this modification is easy to implement in a linear programming context:

$$\begin{array}{llcll} \text{Maximize} & \sum_{i=1}^{n} p_i x_i & & & \\ \text{subject to} & \sum_{i \in C_p} x_i & \leq & d & p = 1, \ldots, m \\ & \sum_{h=1}^{d} y_{ih} & = & x_i & i = 1, \ldots, n \\ & \sum_{i=1}^{n} w_i y_{ih} & \leq & K_h & h = 1, \ldots, d \\ & \multicolumn{4}{c}{x \in \{0,1\}^n, \quad y \in \{0,1\}^{n \times d}} \end{array} \tag{8.28}$$

Variables y_{ih} are used for selecting the storage unit for movie i in case it should be recorded (i.e. $x_i = 1$). In that case $y_{ih} = 1$ if storage unit h was selected, and $y_{iq} = 0$ for all $q \neq h$. Notice, that $d = 1$ implies $y_{i1} = x_i$ for all $i \in V$. Thus, (8.28) reduces to (8.7).

Within a CP approach, Lagrangian coupling could be used to cope with the this requirement. If we relax all but one of the knapsack constraints, and also all of the constraints linking $\sum_h y_{ih}$ to x_i, the remaining IP has the same structure as (8.7). This allows the use of all techniques described previously.

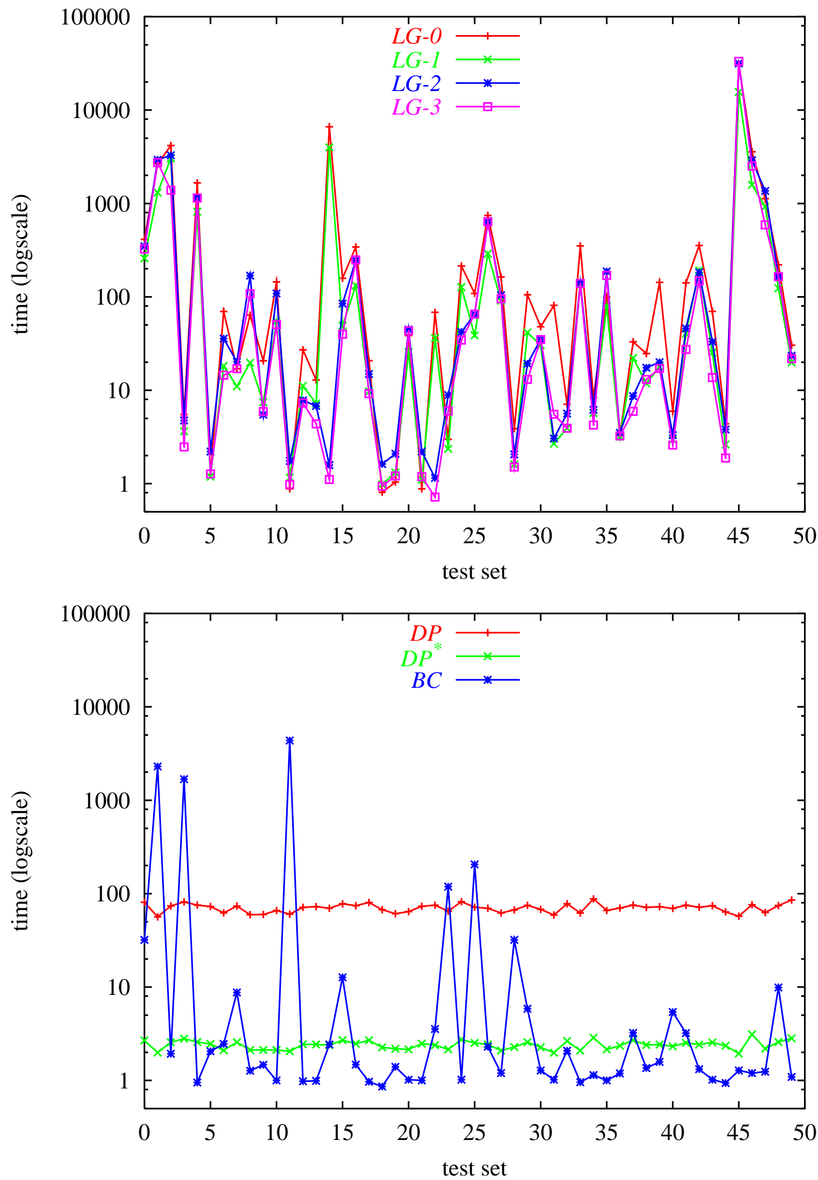


Figure 8.4: *Robustness of the approaches for all 50 instances in 5CU, 24 h, 50 ch. Runtime is given on a logarithmic scale. Figures for other classes look similar. (Lines connecting measuring points are given for easier readability and do not mark intermediate values.)*

9

Domain Filtering for Maximum Clique

A *Clique* is an undirected graph $C = (V_C, E_C)$ where all nodes in V_C are pairwise adjacent. The *Maximum Clique Problem (MC)* on a graph $G = (V, E)$ asks for a clique $C = (V_C, E_C)$ in G having a largest node set V_C.

MC was among the first problems shown to be NP-hard on general graphs (Karp [127]). Håstad [106] proved that MC is not approximable within $|V|^{1/2-\varepsilon}$ for any $\varepsilon > 0$. An $O(|V|/\log|V|^2)$ approximation was developed by Boppana and Halldórsson [38]. On restricted graph families (e.g. perfect graphs) polynomial-time algorithms for MC are known. We refer to e.g. Golumbic [94] for an overview.

The maximum clique problem is important in many fields. Cliques are used in integer programming for presolving [183] or deriving valid cuts [9]. Air traffic control problems can be modeled using conflict cliques (see Barnier and Brisset [22]). Maximum cliques appear in coding theory, fault diagnostics, or pattern recognition. Details on these as well as on other prominent applications of MC are given in Bomze et al. [37].

9.1 Introduction

In this chapter, we introduce some simple cost based domain filtering techniques [81]. Later, we will test these techniques on two branch-and-bound algorithms: One was proposed by Carraghan and Pardalos [48]. Their algorithm is still believed to be one of the fastest clique solver for sparse graphs. It recursively enlarges a clique by adding nodes of a *candidate set* (i.e. a set of nodes that can possibly extend the clique in the current choice point) to it. Pruning based on simple bounds is used to cut off useless parts of the search tree. The other algorithm considered was recently introduced by Östergård [158]. It starts on an one-node graph and adds more and more nodes to it until it works on the entire problem. Cliques found on intermediate graphs are used for an efficient bounding.

Our domain filtering tightens the candidate set by removing or fixing certain nodes. We analytically show that this domain filtering is in a sense as tight as typical upper bounds for MC. As we will see later, our filtering dominates 7 out of 8 combinatorial bounds proposed

for MC. Furthermore, we analyze the relation between LP bounds and domain filtering. The bounds that are not dominated in certain cases are the vertex coloring bounds and the LP bounds.

We add domain filtering, coloring bounds and a primal heuristic to the Carraghan and Pardalos algorithm and achieve a significant improvement on the DIMACS benchmark set. Within a 6 hours time limit the enhanced algorithm can prove optimal results for 46 out of 66 benchmark instances, whereas the original approach is only able to finalize 33 instances. Furthermore, it turns out that domain filtering improves 44 out of 66 test cases compared to applying coloring bounds alone. Similar improvements can be found when introducing these techniques in an algorithm based on Östergård's idea [158].

9.1.1 Literature Review

The maximum clique problem has been attracting the attention of researchers in computer science, operations research and other fields for many years. We can therefore only mention some results and have to refer to a recent overview by Bomze et al. [37] for further details. Some results refer to close relatives of the maximum cliques problem, namely the *vertex cover (VC)* or *independent set (IS)* problem. They stem from simple transformations: $V_C \subseteq V$ defines a MC of $G = (V, E) \Leftrightarrow V_C$ defines a maximum IS in $\overline{G} = (V, (V \times V) \setminus E) \Leftrightarrow V \setminus V_C$ is a minimum VC of $\overline{G}$.

Tarjan and Trojanowski [197] gave a recursive algorithm for the vertex cover problem that runs in time $O(2^{n/3}) \approx O(1.26^n)$. This result was later improved and the asymptotically fastest algorithm is due to Robson [174] and has a time complexity of $O(2^{0.276n}) \approx O(1.22^n)$.

Branch-and-Bound algorithms were contributed by many researchers. Balas and Yu [16] introduced a new idea for implicitly enumerating cliques. They search for a maximal induced triangulated subgraph T of G in which a maximum clique C is then searched. In the second phase they extend T in a way that does not enlarge the maximum clique. When applied in a branch-and-bound scheme their approach was quite successful as the bounds generated turned out to be quite tight.

The methods of Carraghan and Pardalos [48] and Östergård [158] are of certain interests for this work. We will return to those algorithms in Sec. 9.2. A branch-and-bound algorithm using vertex coloring bounds was proposed by Wood [212].

A quadratic programming approach was proposed by Caprara et al. [46]: They developed a solver for the quadratic knapsack problem. The method was also applied to the maximum clique problem. As it was not designed for MC, their approach was inferior to specialized algorithms for MC.

9.1.2 Notation

Throughout this chapter we denote the neighborhood of a node $v \in V$ as $N(v) = \{u \in V | \{u, v\} \in E\}$ and the degree of a node $v \in V$ as $\delta(v) = |N(v)|$. Given a node set U, the graph $G_U = (U, E \cap (U \times U))$ is the graph *induced* by U. We write $N_U(v) = \{u \in U | \{u, v\} \in (U \times U) \cap E\}$ and $\delta_U(v) = |N_U(v)|$ if we speak of the neighborhood, or degree, respectively, of a node v in the subgraph of G induced by U. For a clique C we call $P = \bigcap_{v \in C} N(v)$ the *candidate set of* C. A clique $\widetilde{C}$ is an *extension* of a clique

C if $C \subseteq \widetilde{C}$. (To ease the notation, we often identify a clique C with its set of nodes.) $\mathcal{Z}(G) = \{I \subseteq V \mid G_I$ is a connected component in $G\}$ contains the *node sets of connected components* of the graph $G = (V, E)$. Finally, $\omega(G)$ denotes the size of a maximum clique in G, and $\chi(G)$ refers to the chromatic number of G.

9.2 Branch-And-Bound Algorithms for Maximum Clique

9.2.1 The Carraghan/Pardalos Algorithm

Carraghan and Pardalos [48] enumerate the cliques of a graph $G = (V, E)$ according to some lexicographical order. Without pruning, the algorithm finds the largest clique in G_V containing node v_1, then the largest clique in $G_{V \setminus \{v_1\}}$ containing v_2, etc. A candidate set $P = \bigcap_{v \in C} N(v)$ containing all nodes that are adjacent to the clique C currently under construction is used for bounding. A simple depth-first-search and static variable ordering is used to guide the branch-and-bound (Alg. 20). The well-known `dfmax` [7] program by D. Applegate and D. Johnson is an efficient implementation of this idea.

Algorithm 20 The Carraghan/Pardalos Algorithm for Finding a Maximum Clique C^*

function findClique(set C, set P)
1: **if** $(|C| > |C^*|)$ **then**
2: $\quad C^* \leftarrow C$
3: **if** $(|C| + |P| > |C^*|)$ **then**
4: $\quad$ **for all** $p \in P$ in predetermined order: **do**
5: $\qquad P \leftarrow P \setminus \{p\}$
6: $\qquad C' \leftarrow C \cup \{p\}$
7: $\qquad P' \leftarrow P \cap N(p)$
8: $\qquad$ findClique(C', P')

function main( )
1: $C^* \leftarrow \emptyset$
2: findClique($\emptyset$, V)
3: return C^*

9.2.2 The Östergård Algorithm

Östergård [158] developed a variant of the previous method. (We present this approach as Alg. 21 with some slight adaption to our notation). Instead of determining the cliques in a decreasing node set, his algorithm starts on the graph induced by $\{v_n\}$ and finds the largest clique on that graph. Then it considers the graph induced by $\{v_{n-1}, v_n\}$ and searches the largest clique there. By adding more and more nodes, and determining cliques in each of those sets, it ends up with a maximum clique in G. During this incremental process, bounds $\beta[v_i]$ on the largest cliques containing a certain node v_i can be determined as a by-product (line 5 of main()). With these bounds at hand, Östergård can significantly speed up finding cliques (line 9 of findClique($\cdot, \cdot$)). As each call of the recursion (line 4 in main()) increases C by one node, the search for an improved solution can be stopped as soon as an increased

clique is found (lines 4, 12 in findClique(·, ·)). The idea of finding bounds on small instances of a problem and using them for larger ones is similar to the so-called "Russian Doll Search" of Verfaillie et al. [204].

Algorithm 21 The Östergård Algorithm for Finding a Maximum Clique C^*

function findClique(set C, set P)
```
1:  if (|P| = 0) then
2:     if (|C| > |C*|) then
3:        C* ← C
4:        found ← true
5:  else
6:     if (|C| + |P| > |C*|) then
7:        while (P ≠ ∅) do
8:           select p ∈ P in some predetermined order:
9:           if (|C| + β[p] > |C*|) then
10:             P ← P \ {p}
11:             findClique(C ∪ {p}, P ∩ N(p))
12:             if (found) then break
```
function main( )
```
1:  C* ← ∅
2:  for i ← n down to 1 do
3:     found ← false
4:     findClique({v_i, ..., v_n} ∩ N(v_i), {v_i})
5:     β[v_i] ← |C*|
6:  return C*
```

9.3 Domain Filtering for Maximum Clique

Both algorithms sketched above share the idea of a candidate set $P = \bigcap_{v \in C} N(v)$ containing only those nodes that may extend the clique C currently under construction. P can also be used for pruning, since the largest clique which can be discovered in the current part of the search tree can contain $|C| + |P|$ nodes at most. We will now show that two simple observations can help to tighten the candidate set, and thus, potentially reduce the number of choice points explored.

Looking at the nodes in P in more detail, we can characterize those that can never extend a given clique to a maximum clique:

Lemma 4 *Let $G = (V, E)$ be a graph, C be a clique on G, and P be a candidate set, i.e. $P = \bigcap_{v \in C} N(v)$. Moreover, let $\sigma \in \mathbb{N}$ be a lower bound on the size of a maximum clique in G. Then, for every $v \in P$ such that $|C| + \delta_P(v) < \sigma - 1$, v cannot extend C to a maximum clique of G.*

Proof: Assume we have a node $v \in P$ such that $|C| + \delta_P(v) < \sigma - 1$, and v extends C to a maximum clique $C^* \supseteq C$ with $|C^*| = k$ nodes. Then for all $u \in C^*$ we have:

$\delta_{C^*}(u) = k-1$. Since σ is a lower bound on $|C^*|$ it holds $\sigma - 1 \leq k-1 = \delta_{C^*}(v)$. The degrees in C^* can be partitioned into $\delta_{C^*}(v) = \delta_C(v) + \delta_{C^* \setminus C}(v)$. Hence, we obtain $\sigma - 1 \leq k-1 = \delta_{C^*}(v) = \delta_C(v) + \delta_{C^* \setminus C}(v) \leq |C| - 1 + \delta_P(v) < \sigma - 1$ which is a contradiction. ■

The next lemma identifies nodes that will be part of any extension of the current clique to a maximum one:

Lemma 5 *Let $G = (V, E)$ be a graph, C be a clique on G, and P be a candidate set, i.e. $P = \bigcap_{v \in C} N(v)$. Then, every $v \in P$ such that $\delta_P(v) = |P| - 1$, is contained in any maximum clique of G that also contains C.*

Proof: Let C^* be a maximum clique and $C \subseteq C^*$. Then $C^* \subseteq C \cup P$ (by definition of P). Assume, $v \in P$ such that $\delta_P(v) = |P| - 1$, but $v \notin C^*$. By construction, v is adjacent to all nodes in C (because it is a member of the candidate set) and all nodes in P (because of its degree in P). Hence, v is also adjacent to any node u in $C^* \subseteq C \cup P$ which means $C^* \cup \{v\}$ is a clique containing C, but being larger than C^*. This contradicts the choice of C^* as a maximum clique. ■

9.3.1 A Domain Filtering Algorithm

With these observations at hand we are ready to present a domain filtering procedure for MC (Alg. 22). For Lemma 4, we use $\sigma = (|C^*| + 1)$ as a lower bound for the necessary checks in line 2. Afterwards, Lemma 5 fixes those nodes required in any maximal clique extending

Algorithm 22 Domain Filtering for Maximum Clique Problem

function DomainFilter(set C, set P)

1: // *reduce possible set*
2: **while** $(\exists\ v \in P : \deg_P(v) + |C| < |C^*|)$ **do**
3: $\quad P \leftarrow P \setminus \{v\}$ // *lemma 4*
4: // *increase required set*
5: **while** $(\exists\ v \in P : \deg_P(v) = |P| - 1)$ **do**
6: $\quad C \leftarrow C \cup \{v\}$ // *lemma 5*
7: $\quad P \leftarrow P \setminus \{v\}$ // *lemma 5*
8: **return** (C, P)

C. We show correctness and running time of Alg. 22 by

Lemma 6 *Let $G = (V, E)$ be a graph, C be a clique on G, and P be a candidate set, i.e. $P = \bigcap_{v \in C} N(v)$.*

(i) If Lemma 4 and Lemma 5 are applicable simultaneously, i.e. there is $v \in P$ such that $|P| - 1 = \delta_P(v)$ and $\delta_P(v) + |C| < |C^*|$, *then a maximal clique in $C \cup P$ is not larger than the incumbent maximum clique C^*.*

(ii) Domain filtering can be done in one single round, i.e. after first applying Lemma 4, then Lemma 5 it holds: $\forall v \in P : (|C^*| - |C|) \leq \deg_P(v) < (|P| - 1)$ and neither Lemma can be applied again.

(iii) The result of Alg. 22 is independent of the order in which nodes are considered within each of the two while-loops (lines 2,5).

(iv) Alg. 22 terminates after time $O(|P|^2)$.

Proof:

(i) Let $v \in P$ such that $|P| - 1 = \delta_P(v)$ and $\delta_P(v) + |C| < |C^*|$. Then $|P| - 1 + |C| < |C^*| \Rightarrow |P| + |C| \leq |C^*|$, hence we cannot find a larger clique than $|C^*|$ in $C \cup P$.

(ii) After leaving the loop in lines 2,3 it holds $\forall v \in P : \deg_P(v) + |C| \geq |C^*|$. Now, any node p that meets the condition in the while-loop of line 5 is deleted from P, and added to C. This reduces $\deg_P(u)$ by one for all nodes $u \in P \setminus \{p\}$ (p is adjacent to all nodes in P) and increases $|C|$ by one (P and C are disjoint). Hence, $\forall v \in P \setminus \{p\}$: $\deg_{P\setminus\{p\}}(v) + |C \cup \{p\}| = \deg_P(v) - 1 + |C| + 1 < |C^*|$ and thus, the condition in line 2 is not fulfilled.

(iii) Let $v, w \in P$ be two nodes that fulfill the condition of the while-loop in line 2, i.e. $\deg_P(v) + |C| < |C^*|$ and $\deg_P(w) + |C| < |C^*|$. We have to show that both nodes will be removed, no matter which one is considered first. W.l.o.g. we start with v: If v is removed from P, the degree of any other node in P remains the same, or decreases by one: $\deg_P(w) - 1 \leq \deg_{P\setminus\{v\}}(w) \leq \deg_P(w)$. Hence, after the removal, $\deg_{P\setminus\{v\}}(w) + |C| < |C^*|$, and w will also be removed.

Similarly, let $v, w \in P$ fulfill the condition of the while-loop in line 5, i.e. $\deg_P(v) = |P| - 1$ and $\deg_P(w) = |P| - 1$. W.l.o.g. we remove v first. The degree of all nodes in P is decreased by exactly one (since v was adjacent to all nodes in P). Also, the candidate set's size decreases by one. Hence, $\deg_{P\setminus\{v\}}(w) = \deg_P(w) - 1 = |P| - 1 - 1 = |P \setminus \{v\}| - 1$, and w will be removed as well.

(iv) Any node v in P is adjacent to $|P| - 1$ other nodes in P at most. Removing v therefore requires $|P| - 1$ updates of degree values at most. As P decreases monotonically within the while-loops, $|P|$ removals at most are possible, summing up to a total number of $O(|P|^2)$ removals and updates.

In order to support an efficient handling, we use a queue Q here. Each deletion of an element from P is recorded by storing its neighbors into Q. We perform a degree update on all elements of Q, and check whether domain filtering would fix that element. If so, that element is removed, thus its neighbors are stored into the queue, and we continue until the queue is empty. In so doing, we achieve the running time claimed. ■

In chapter 10 we will introduce our domain filtering into some branch-and-bound approach and use it within each branch-and-bound node. According to lemma 6*(iv)* the running time per node then is $O(|P|^2)$. But as we cannot continue searching after P is empty, running time for the entire search path from the root node to a leaf in the search tree is also bounded by $O(|P|^2)$.

Corollary 4 *When applied in a tree search the accumulated running time of algorithm 22 for an entire search path from the root node to a descended node in the search tree is $O(|P|^2)$.*

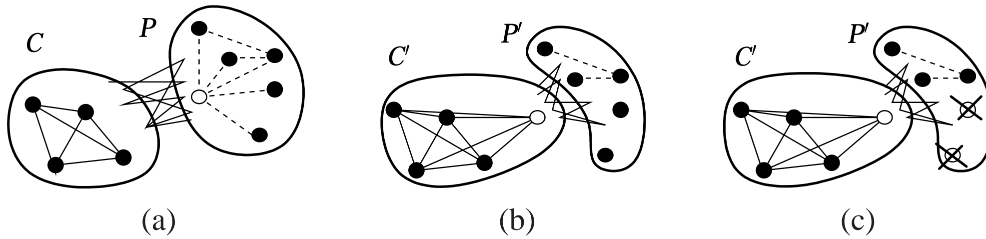


Figure 9.1: *(a) a clique C and its candidate set P. (b) applying lemma 5 fixes one more node. (c) as a result, two nodes now have degree zero, and can be eliminated according to lemma 4.*

Finally, we illustrate the behavior of Algorithm 22 in Figure 9.1: (a) represents the settings without domain filtering, (b),(c) sketches situations when domain filtering can be applied successfully.

9.4 Upper Bounds vs. Domain Filtering: Analytical Results

As an upper bound, the branch-and-bound approach of Carraghan and Pardalos [48], as well as the one of Östergård [158], uses $|C|+|P|$, the size of the clique currently at hand plus the size of the candidate set. The approach of Wood [212] mentioned in the introduction uses coloring bounds. In this section we analytically study the power of domain filtering compared to several upper bounds used in OR approaches. We show in particular that some upper bounds for MC used in OR methods are already dominated by the domain filtering.

9.4.1 Integer Programming Models for Maximum Clique

The maximum clique problems can be adequately modeled by linear integer programs. Possibly the simplest way is provided by the so-called *edge-formulation*. It uses one 0-1 variable for each node in V and a constraint for each edge not presented in G. The idea is to search for the largest number of nodes such that no two non-adjacent nodes are selected. This corresponds exactly to an independent set problem on $\overline{G}=(V,\overline{E})$, where $\overline{E}=\{\{i,j\}\mid\{i,j\}\in(V\times V)\setminus E,\ i\neq j\}$:

$$\begin{array}{rl} \max & \sum_{i=1}^{n} x_i \\ \text{s.t.} & x_i + x_j \leq 1 \quad \forall\, \{i,j\} \in \overline{E} \\ & x \in \{0,1\}^n \end{array} \tag{9.1}$$

A branch-and-cut approach based on this integer programming formulation has been investigated in [14]. We mentioned already in a previous chapter that the LP relaxation of (9.1) is tight if and only if $\overline{G}$ is bipartite (see Grötschel et al. [99, Prop. 1.3]). A tighter formulation is based on the fact that from any independent set in G one node at most can be part of a clique. Let I_G denote the set of all maximal independent sets of G. Then MC can be described as:

$$\begin{array}{rl} \max & \sum_{i=1}^{n} x_i \\ \text{s.t.} & \sum_{i\in S} x_i \leq 1 \quad \forall\, S \in I_G \\ & x \in \{0,1\}^n \end{array} \tag{9.2}$$

Unfortunately, $|I_G|$ is exponential, and in fact Grötschel et al. [98] showed that even the LP relaxation of (9.2) is NP-hard on general graphs. Furthermore, it is polynomially solvable on perfect graphs, and G is perfect if and only if the optimal solution of the LP relaxation of (9.2) always is integral (Grötschel et al. [99]).

In both models we can replace the integrality constraint $x_i \in \{0,1\}$ by $0 \leq x_i \leq 1$ and obtain linear programming bounds for MC. We start our discussion with the LP bounds derived from (9.1).

9.4.1.1 Linear Programming Bounds

In both models we can replace the integrality constraint $x_i \in \{0,1\}$ by $0 \leq x_i \leq 1$ and obtain linear programming bounds for MC. We start our discussion with the LP bounds derived from (9.1).

It is easy to see, that $x_i = \frac{1}{2}$ is always a feasible, but not necessarily optimal solution to the LP relaxation of (9.1). This fact will be used in some of our arguments if an exact solution to the LP is not needed. Surprisingly, even the exact value of an optimal solution to the LP relaxation of (9.1) is rather simple: Any variable is assigned either $0, \frac{1}{2}$ or 1:

Theorem 2 (Nemhauser and Trotter jr., 1974) *Let $G = (V,E)$ be an unweighted graph, and let $\overline{x} = \arg\max_x\{\sum_{i\in V} x_i \mid x_i + x_j \leq 1\ \forall\ \{i,j\} \in \overline{E},\ 0 \leq x_i \leq 1,\ \forall i \in V\}$ be an optimal solution to the LP relaxation of (9.1). Then $\overline{x}_i \in \{0, \frac{1}{2}, 1\}$, $i = 1,\ldots,n$.*

Proof: (see [154, Proposition 2.1]) ■

An interesting theoretical property of the edge formulation is that variables assigned to one in the LP relaxation can be fixed without loosing an optimal IP solution:

Theorem 3 (Nemhauser and Trotter jr., 1975) *Suppose $\overline{x}$ is an optimum $\{0, \frac{1}{2}, 1\}$-valued solution to the linear relaxation of (9.1), and $\mathcal{P} = \{i \mid \overline{x}_i = 1\}$. Then an optimal (integer) solution x^* to (9.1) exists such that $x_i^* = 1\ \forall i \in \mathcal{P}$.*

Proof: (see [155, Theorem 2]) ■

Only few variables, though, tend to have the value one in an LP relaxation. E.g. in 63 out of the 66 benchmark instances considered in our benchmark tests, all variables of the LP relaxation had a value of $\frac{1}{2}$. Thus, the above theorem rarely helps to reduce the problem in practical applications. Moreover, the LP/IP gap can be large:

Remark 2 *Consider a graph G with no edges: $V = \{1,\ldots,n\}$ and $E = \emptyset$. Then the optimal IP solution is 1. Let us consider the LP relaxation: Setting $x_i = \frac{1}{2},\ \forall i \in V$ is a feasible solution to the edge-formulation, and $\sum_{i\in V} x_i = \frac{|V|}{2}$. The resulting LP/IP gap therefore is $\frac{\sum_{i\in V} x_i - 1}{1} = \frac{|V|}{2} - 1 = O(|V|)$.*

Now, let us consider (9.2). From its definitions, the linear relaxation of (9.2) is at least as tight as that of (9.1), since all constraints of (9.1) are also contained in (9.2). In fact, the LP relaxation of (9.2) always gives a value that is at most as large as the chromatic number $\chi(G)$ of the graph currently at hand:

Lemma 7 *Let $G = (V, E)$ be an unweighted graph, and consider the value z of the LP relaxation of (9.2): $z = \max_x\{\sum_{i\in V} x_i \mid \sum_{i\in\mathcal{S}} x_i \leq 1 \ \forall\ \mathcal{S} \in I_G, 0 \leq x_i \leq 1,\ \forall i \in V\}$, where I_G is the set of all independent sets of G. Then $\omega(G) \leq z \leq \chi(G)$.*

Proof:

$$\begin{aligned}
\omega(G) &= \max_x\{\sum_{i\in V} x_i \mid \sum_{i\in\mathcal{S}} x_i \leq 1 \ \forall\ \mathcal{S} \in I_G,\ x_i \in \{0,1\},\ \forall i \in V\} && (9.3)\\
&\leq \max_x\{\sum_{i\in V} x_i \mid \sum_{i\in\mathcal{S}} x_i \leq 1 \ \forall\ \mathcal{S} \in I_G,\ 0 \leq x_i \leq 1,\ \forall i \in V\} && (9.4)\\
&= \min_y\{\sum_{\mathcal{S}\in I_G} y_{\mathcal{S}} \mid \sum_{\substack{\mathcal{S}\in I_G\\ s.t.\ i\in\mathcal{S}}} y_{\mathcal{S}} \geq 1 \ \forall\ i \in V,\ y_{\mathcal{S}} \geq 0, \forall\mathcal{S} \in I_G\} && (9.5)\\
&= \min_y\{\sum_{\mathcal{S}\in I_G} y_{\mathcal{S}} \mid \sum_{\substack{\mathcal{S}\in I_G\\ s.t.\ i\in\mathcal{S}}} y_{\mathcal{S}} \geq 1 \ \forall\ i \in V,\ 0 \leq y_{\mathcal{S}} \leq 1, \forall\mathcal{S} \in I_G\} && (9.6)\\
&\leq \min_y\{\sum_{\mathcal{S}\in I_G} y_{\mathcal{S}} \mid \sum_{\substack{\mathcal{S}\in I_G\\ s.t.\ i\in\mathcal{S}}} y_{\mathcal{S}} \geq 1 \ \forall\ i \in V,\ y_{\mathcal{S}} \in \{0,1\}, \forall\mathcal{S} \in I_G\} && (9.7)\\
&= \chi(G) && (9.8)
\end{aligned}$$

Equation (9.3) is the initial IP formulation used in (9.2). In (9.4) we relax the integrality constraints which may increase the optimal value. (9.5) is the dual formulation of the previous line, and both terms have the same optimal values due to strong duality.

In the next step, we add $y_{\mathcal{S}} \leq 1$ and claim that the optimal value does not change: Assume, there is an $y_{\mathcal{S}} = \alpha > 1$ in an optimal solution to (9.5). Any constraint met by $y_{\mathcal{S}} = \alpha$ will also be met by $y_{\mathcal{S}} = 1$. Furthermore, since we minimize the sum over all $y_{\mathcal{S}}$ decreasing $y_{\mathcal{S}}$ down to 1 will improve the objective. This contradicts our assumption that we started with an optimal solution.

Finally, we introduce integrality again in (9.7) and obtain the column generation description of the chromatic number problem (see Mehrotra and Trick [150]). ∎

Corollary 5 *Whenever the largest independent set in G contains two nodes at most, (9.1) and (9.2) are identical. In that case, the LP relaxation of (9.1) is at least as good as the chromatic number bound on that graph.*

Corollary 6 *If G is a perfect graph, i.e. $\omega(G) = \chi(G)$, then the optimal solution value of the LP relaxation of (9.2) as well as the optimal IP value are equal to $\omega(G)$.*

As mentioned before, on general graphs the LP relaxation of (9.2) is NP-hard (Grötschel et al. [98]). Its exponentially many constraints make this formulation a good candidate for branch-and-cut approaches. Branch-and-cut is designed to exploit this special structure without necessarily generating the entire IP model (see e.g. [19, 69, 123, 147]).

9.4.2 Combinatorial Bounds

To introduce some combinatorial bounds, we assume the setting presented in Sec. 9.2: C is the clique currently under construction, $P = \bigcap_{v\in C} N(v)$ the corresponding candidate set. The following lemma lists some well-known upper bounds $\mathcal{U}_i(C, P)$ for a maximal clique $C^* \supseteq C$ (see also Bomze et al. [37]):

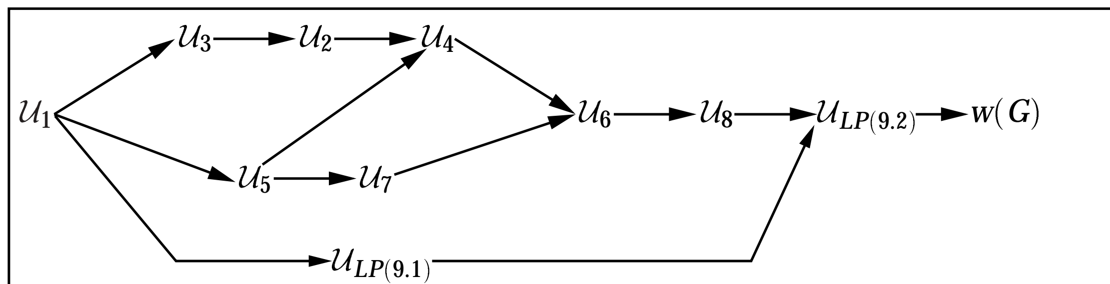
Lemma 8 (Upper Bounds for MC) *The following upper bounds hold for MC:*

1. *Only nodes from the candidate set can extend* C*:* $\mathcal{U}_1(C,P) = |C| + |P|$.
2. *A node with a maximum degree in* P *limits the size of any extension of* C *to a maximal clique:* $\mathcal{U}_2(C,P) = |C| + \max\{\delta_P(v) + 1 \mid v \in P\}$.
3. *Only one connected component in* P *can extend* C*:* $\mathcal{U}_3(C,P) = |C| + |I^{max}|$*, where* I^{max} *denotes the node set of the largest connected component in* $\mathcal{Z}(G_P)$
4. *A* k*-clique requires* k *nodes with a degree of at least* $k-1$*:*
$$\mathcal{U}_4(C,P) = |C| + \max\{k \mid \exists\ v_1 < \cdots < v_k \in P,\ \delta_P(v_i) \geq k-1\}$$
5. *A* k*-clique has* $k(k-1)/2$ *edges:* $\mathcal{U}_5(C,P) = |C| + \max\{k \in \mathbb{N} \mid \frac{k(k-1)}{2} \leq |E_P|\}$*, where* E_P *is the edge set of the graph* $G_P = (P, E_P)$ *induced by* P.
6. *Apply* $\mathcal{U}_4(C,P)$ *only to connected components of* G_P*:*
$$\mathcal{U}_6(C,P) = |C| + \max_{I\in\mathcal{Z}(G_P)} \max\{k \mid \exists\ v_1 < \cdots < v_k \in I,\ \delta_P(v_i) \geq k-1\}$$
7. *Apply* $\mathcal{U}_5(C,P)$ *only to connected components of* G_P*:*
$$\mathcal{U}_7(C,P) = |C| + \max_{I\in\mathcal{Z}(G_P)} \max\{k \in \mathbb{N} \mid \frac{k(k-1)}{2} \leq |E_I|\},$$
where E_I *is the edge set of the graph* $G_I = (P, E_I)$ *induced by* I.
8. *Any* k*-clique needs* k *colors in a vertex coloring:* $\mathcal{U}_8(C,P) = |C| + \chi(G_P)$*, where* $\chi(G_P)$ *denotes the (vertex) chromatic number of the graph induced by* P.

(Correctness of these bounds follows directly from their description. For applications of these bounds and further information we refer to [48] for $\mathcal{U}_1$*, to [16, 212] for* $\mathcal{U}_8$*, and [37] for some bound similar to* $\mathcal{U}_5$*. The other bounds are simple extensions of* $\mathcal{U}_1$*.)*

Some of these bounds are obviously contained in others. We will present a taxonomy of these combinatorial bounds and the LP bounds of the previous section. To our knowledge, such a taxonomy has not been presented before, though some relations are obvious and have probably been used earlier.

Lemma 9 (Taxonomy of Bounds) *Let* $G = (V,E)$ *be a graph,* C *a (not necessarily maximal) clique in* G*, and* $P = \bigcap_{v\in C} N(v)$ *the corresponding candidate set. With* $\mathcal{U}_1 - \mathcal{U}_8$*, we refer to the bounds mentioned in Lemma 8,* $\omega(G)$ *is the clique number of* G*, and* $\mathcal{U}_{LP(9.1)}$ *and* $\mathcal{U}_{LP(9.2)}$ *refer to the respective LP relaxation discussed in section 9.4.1. Whenever the best value obtainable by a bound is never smaller than the best value of another bound (*$\mathcal{U}_i(C,P) \geq \mathcal{U}_j(C,P)$*), we note that as* $\mathcal{U}_i \to \mathcal{U}_j$*. Then we get the following picture:*



(we have left out transitive dominance, e.g. $\mathcal{U}_3(C,P) \geq \mathcal{U}_6(C,P)$*)*

Proof:

$\mathcal{U}_1(C,P) \geq \mathcal{U}_3(C,P)$**:** For any connected component G_I of P it holds $|P| \geq |I|$.

$\mathcal{U}_3(C,P) \geq \mathcal{U}_2(C,P)$**:** Let $k := \max\{\delta_P(v)+1 \mid v \in P\}$. Then there exists a connected component with at least k nodes $\Rightarrow \ |I^{max}| \geq k$.

$\mathcal{U}_2(C,P) \geq \mathcal{U}_4(C,P)$**:** Let $k' := \max\{k \mid \exists\ v_1 < \cdots < v_k \in P,\ \delta_P(v_i) \geq k-1\}$
$\Rightarrow \max\{\delta_P(v)+1 \mid v \in P\} \geq (k'-1)+1 = k'$.

$\mathcal{U}_4(C,P) \geq \mathcal{U}_6(C,P)$**:** $\max\{k \mid \exists\ v_1 < \cdots < v_k \in P,\ \delta_P(v_i) \geq k-1\}$
$\geq \max_{I \in \mathcal{Z}(G_P)} \max\{k \mid \exists\ v_1 < \cdots < v_k \in I,\ \delta_P(v_i) \geq k-1\}$

$\mathcal{U}_6(C,P) \geq \mathcal{U}_8(C,P)$**:** Let $k' := \max_{I \in \mathcal{Z}(G_P)} \{\max\{k \mid \exists\ v_1 < \cdots < v_k \in I,\ \delta_P(v_i) \geq k-1\}\}$. Assume, $\chi(G_P) = k'' > k'$. As different connected components can be independently colored, an $I \in \mathcal{Z}(G_P)$ exists, such that $\chi(G_I) = \chi(G_P) = k''$. Then G_I contains $k'' > k'$ nodes of degree $k''-1 > k'-1$, hence k' is not maximal — a contradiction.

$\mathcal{U}_8(C,P) \geq \mathcal{U}_{LP(9.2)}(C,P)$ and $\mathcal{U}_{LP(9.2)}(C,P) \geq \omega(G)$**:** follows directly from Lemma 7.

$\mathcal{U}_1(C,P) \geq \mathcal{U}_5(C,P)$**:** Let $G_P = (P, E_P) \Rightarrow |E_P| \leq \frac{|P|(|P|-1)}{2} \Rightarrow \max\{k \in \mathbb{N} \mid \frac{k(k-1)}{2} \leq |E_P|\} \leq \max\{k \in \mathbb{N} \mid \frac{k(k-1)}{2} \leq \frac{|P|(|P|-1)}{2}\} = |P|$.

$\mathcal{U}_5(C,P) \geq \mathcal{U}_7(C,P)$**:** $\max\{k \in \mathbb{N} \mid \frac{k(k-1)}{2} \leq |E_P|\} \geq \max_{I \in \mathcal{Z}(G_P)} \max\{k \in \mathbb{N} \mid \frac{k(k-1)}{2} \leq |E_I|\}$

$\mathcal{U}_5(C,P) \geq \mathcal{U}_4(C,P)$**:** Let $k' := \max\{k \mid \exists\ v_1 < \cdots < v_k \in P,\ \delta_P(v_i) \geq k-1\}$. Then $|E_P| \geq \frac{k'(k'-1)}{2} \Rightarrow \max\{l \in \mathbb{N} \mid \frac{l(l-1)}{2} \leq |E_P|\} \geq k'$.

$\mathcal{U}_7(C,P) \geq \mathcal{U}_6(C,P)$**:** Let I be the node set, where the maximum of bound $\mathcal{U}_6(C,P) =: k$ is found. On a connected component, $\mathcal{U}_6(C,P)$ and $\mathcal{U}_4(C,P)$, as well as $\mathcal{U}_7(C,P)$ and $\mathcal{U}_5(C,P)$ are the same. As proved above $\mathcal{U}_5(C,P) \geq \mathcal{U}_4(C,P)$. Therefore, $\mathcal{U}_7(C,P) \geq \mathcal{U}_6(C,P)$ on that component and (since k is optimal for P) also on P.

$\mathcal{U}_1(C,P) \geq \mathcal{U}_{LP(9.1)}(C,P)$**:** We apply the LP relaxation of (9.1) to the graph induced by C, P. This LP contains $|C|+|P|$ variables. These variables have a value between zero and one. Hence, $\mathcal{U}_1 = |C|+|P| \geq \max_x\{|C| + \sum_{i \in P} x_i \mid x_i + x_j \leq 1\ \forall\ \{i,j\} \in \overline{E}_P,\ x_i \in \{0, \frac{1}{2}, 1\},\ \forall i \in P\}$.

$\mathcal{U}_{LP(9.1)}(C,P) \geq \mathcal{U}_{LP(9.2)}(C,P)$**:** follows directly from the definition of both LPs (all constraints of (9.1) are contained in (9.2)).

The following calculations show that there is no general dominance than those described above:

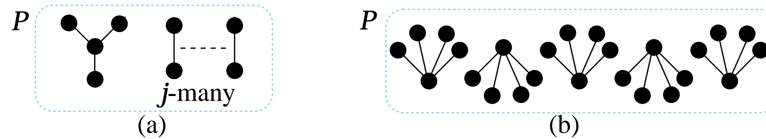


Figure 9.2: *(a) $\mathcal{U}_2$ versus $\mathcal{U}_5$, (b) $\mathcal{U}_4$ versus $\mathcal{U}_7$*

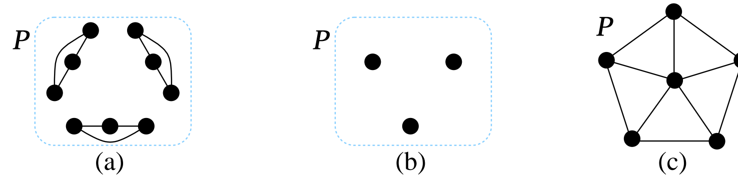


Figure 9.3: *Three graphs showing that $\mathcal{U}_{LP(9.1)}$ can be (a) worse than $\mathcal{U}_3$, (b) worse than $\mathcal{U}_5$, or (c) better than $\mathcal{U}_8$, respectively.*

$\mathcal{U}_2(C,P) \lessgtr \mathcal{U}_5(C,P)$ and $\mathcal{U}_3(C,P) \lessgtr \mathcal{U}_5(C,P)$: Consider the candidate set of Fig. 9.2(a). There, $\mathcal{U}_2(C,P) = |C| + 4$ and $\mathcal{U}_3(C,P) = |C| + 4$ because of the "star" in P. We have $|E_P| = 3 + j$, so we get $\mathcal{U}_5(C,P) = |C| + \max\{k \in \mathbb{N} \mid \frac{k(k-1)}{2} \leq 3 + j\}$. If $j = 1$ we have $\mathcal{U}_5(C,P) = |C| + 3 < \mathcal{U}_2(C,P), \mathcal{U}_3(C,P)$. If $j = 7$ we have $\mathcal{U}_5(C,P) = |C| + 5 > \mathcal{U}_2(C,P), \mathcal{U}_3(C,P)$. Hence, there is no general dominance between $\mathcal{U}_5(C,P)$ and $\mathcal{U}_2(C,P), \mathcal{U}_3(C,P)$.

$\mathcal{U}_4(C,P) \lessgtr \mathcal{U}_7(C,P)$: Consider the candidate set of Fig. 9.2(b). There, $\mathcal{U}_4(C,P) = |C| + 5$ and $\mathcal{U}_7(C,P) = |C| + \max\{k \in \mathbb{N} \mid \frac{k(k-1)}{2} \leq 4\} = |C| + 3$. In this case, $\mathcal{U}_4(C,P) \geq \mathcal{U}_7(C,P)$. Now, consider some P that only contains a single connected component. Then $\mathcal{U}_7(C,P) \equiv \mathcal{U}_5(C,P)$ and as proved above: $\mathcal{U}_5(C,P) \geq \mathcal{U}_4(C,P)$. Again, we cannot show general dominance between two bounds.

$\mathcal{U}_2(C,P) \lessgtr \mathcal{U}_7(C,P)$ and $\mathcal{U}_3(C,P) \lessgtr \mathcal{U}_7(C,P)$: In the single connected component case it holds $\mathcal{U}_7(C,P) \equiv \mathcal{U}_5(C,P)$. But as proved earlier: $\mathcal{U}_2(C,P) \lessgtr \mathcal{U}_5(C,P)$ and $\mathcal{U}_3(C,P) \lessgtr \mathcal{U}_5(C,P)$, hence $\mathcal{U}_2(C,P) \lessgtr \mathcal{U}_7(C,P)$. ■

Before leaving this section we would like to give two examples of graphs where $\mathcal{U}_{LP(9.1)}$ is weaker than bounds $\mathcal{U}_3$ and $\mathcal{U}_5$, respectively, and another graph, where $\mathcal{U}_{LP(9.1)}$ is better than $\mathcal{U}_8$. In other words:

Remark 3 *There is no tighter classification of $\mathcal{U}_{LP(9.1)}$ than the one given in Lemma 9.*

Proof: Let us start with Fig. 9.3(a): $\mathcal{U}_3(C,P) = |C| + 3$, since the largest connected component in P contains three nodes. A feasible solution to (9.1) is to assign $\frac{1}{2}$ to all nodes in P, thus $\mathcal{U}_{LP(9.1)}(C,P) \geq |C| + \frac{9}{2} > \mathcal{U}_3(C,P)$. Analogously, in Fig. 9.3(b) the LP approach can assign $\frac{1}{2}$ to all three nodes, resulting in an LP based bound of $|C| + \frac{3}{2}$. $\mathcal{U}_5$ will choose $k = 1$ as the maximal possible number that meets $\frac{k(k-1)}{2} \leq |E_P| = 0$. As a result, $\mathcal{U}_{LP(9.1)}(C,P) > |C| + 1 = \mathcal{U}_5(C,P)$.

On other graph structures LP (9.1) can provide fairly tight bounds. The graph in Fig. 9.3(c) needs at least four[1] colors for a feasible node coloring. For LP (9.1) there is no constraint involving the variable assigned to the middle node. Therefore, this variable will be set to one in an optimal LP. The remaining nodes will get $\frac{1}{2}$, resulting in an LP value of $3\frac{1}{2}$. Using Lemma 2 it is easy to see that this is indeed the optimal LP value. In this case $\mathcal{U}_{LP(9.1)}(C,P) = |C| + 3\frac{1}{2} < \mathcal{U}_8(C,P) = 4$. ■

9.4.3 Upper Bounds Dominated by Domain Filtering

We are now able to prove our main result on the connection between domain filtering and upper bounds: We show that the domain filtering described in Sec. 9.3 dominates bounds $\mathcal{U}_1$ – $\mathcal{U}_7$. I.e. after applying Alg. 22 to (C,P) and obtaining a new (C',P'), those bounds cannot prune the current part of the subtree.

Our dominance model is thus the ability of either technique to shrink the search space — either by using bounds and pruning useless parts of the search tree, or by domain filtering, and fixing variables. We believe that a direct comparison of the pruning-power of OR bounds with the effectiveness of cost based filtering techniques is a good model in this context.

Theorem 4 *Let $G = (V,E)$ be a graph, C be a clique on G, and P be a candidate set, i.e. $P = \bigcap_{v \in C} N(v)$. Furthermore, let $\sigma \in \mathbb{N}_0$ be a lower bound on the size of a maximum clique in G. Let $|C| + |P| > \sigma$ and let C' and P' be the node sets after applying the domain filtering of Algorithm 22. Then, $\mathcal{U}_i(C',P') > \sigma$ for $i = 1,\dots,7$.*

Proof: According to lemma 9 it is sufficient to prove $\mathcal{U}_6(C',P') > \sigma$. So let us assume, $\mathcal{U}_6(C',P') \leq \sigma$, i.e. $|C'| + \max_{I \in \mathcal{Z}(G_{P'})} \max\{k \mid \exists\, v_1 < \cdots < v_k \in I,\ \delta_{P'}(v_i) \geq k-1\} \leq \sigma$.
After applying Alg. 22 it holds: $\forall\, v \in P' :\ (\sigma - |C'|) \leq \delta_{P'}(v)$ (according to Lemma 6(*ii*))
$\Rightarrow\ \forall\, v \in P' :\ |C'| + \delta_{P'}(v) \geq \sigma \geq |C'| + \max_{I \in \mathcal{Z}(G_{P'})} \max\{k \mid \exists\, v_1 < \cdots < v_k \in I,\ \delta_{P'}(v_i) \geq k-1\}$
$\Rightarrow\ \forall\, v \in P' :\ \delta_{P'}(v) \geq \max_{I \in \mathcal{Z}(G_{P'})} \max\{k \mid \exists\, v_1 < \cdots < v_k \in I,\ \delta_{P'}(v_i) \geq k-1\} =: k'$

So the degree of all nodes in P' is at least k'. Especially, as there is a connected component in P' having at least $k'+1$ nodes with degree larger than k'. This contradicts the maximality of $\mathcal{U}_6(C',P')$. ■

9.4.4 Upper Bounds Not Dominated by Domain Filtering

Bound $\mathcal{U}_8$ is not included in the above theorem. The following examples show that depending on the situation domain filtering may be as good as this bound, or the bound may be more accurate than filtering:

Assume, the candidate set after domain filtering is given as sketched in the Fig. 9.4(a). Furthermore, we know a lower bound of $|C^*| = 3$, thus, $\sigma = 4$ and we have already $|C'| = 1$.

[1] Color the middle node with color 0, the remaining nodes form a ring with an odd number of nodes. Hence they need three colors, and since all nodes on the ring are adjacent to the middle node these three colors have to be different from color 0.

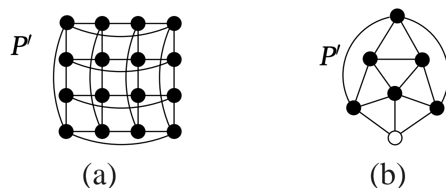


Figure 9.4: *Graphs used for illustrating the connection between domain filtering and $\mathcal{U}_8$.*

P' cannot be reduced further by domain filtering, as $\forall\ v \in P' :\ \underbrace{\sigma - |C'|}_{=3} \le \underbrace{\delta_{P'}(v)}_{=4} < \underbrace{|P'| - 1}_{=15}$.
Now, as $\chi(G_{P'}) = 2$ we get $\mathcal{U}_8(C', P') = 3 < \sigma$. Consequently, applying the coloring bound here can prune parts of the search tree which would still be considered when using filtering alone.

Things change when considering Fig. 9.4(b): When having $\sigma = 5$ and $|C'| = 1$, domain filtering can remove all nodes in P' as $\sigma - |C'| = 4$, but the degree of the white node in (b) is only 3. First, the white node is removed, decreasing the degree of its neighbors to 3. These two nodes disappear next, and in a last step the remaining 4 nodes are eliminated as well. Domain filtering has removed all potential nodes, and search backtracks.

Exact coloring bounds are not as successful: Because $\chi(G_{P'}) = 4$, we obtain $\mathcal{U}_8(C', P') = 5 = \sigma$ showing that in this case the coloring bound cannot prune the remaining subtree. A branch-and-bound algorithm would start new branches on this subproblem.

9.4.5 Computational Complexity

Better accuracy of bounds usually requires higher running time when calculating these bounds. As benchmark graphs for cliques are typically rather dense, we assume that the graph G is stored as an adjacency matrix. Also, we assume, that $|C'|$ can be determined in $O(|C'|)$. $\mathcal{U}_1$ requires the size of P, which can be determined in $O(|P|)$ within a reasonable data structure. Also the max. degree, required by $\mathcal{U}_2$, can be determined in that time if degree-information is available (otherwise: $O(|P|^2)$).

A largest connected component in P can be detected by a DFS in a dense graph in time $O(|P|^2)$. However, $\mathcal{U}_4$ being tighter than $\mathcal{U}_3$ can be computed in time $O(|P| \log |P|)$ if degree information is available (or in quadratic time otherwise). Similarly, $\mathcal{U}_6$ and $\mathcal{U}_7$ base their information on connected components, and need therefore time $O(|P|^2)$. If degree information is at hand, bound $\mathcal{U}_5$ can be determined in $O(|P|)$, otherwise this bound needs to check G completely, giving time complexity $O(|P|^2)$.

Exact vertex coloring, needed for $\mathcal{U}_8$ is in general NP-hard. For bound calculation, however, heuristics are used. They visit each node and each edge a constant number of time, giving running time $O(|P|^2)$ in a dense graph (see e.g. [40, 16, 212]). As a drawback, heuristic values may not be optimal, and the resulting bounds are not as tight as in the theoretical analysis. We will elaborate more on coloring heuristics in Section 10.1.

Linear programming bounds are typically solved using a simplex algorithm. This approach is known to have an exponential worst case running time (Klee and Minty [129]). Interior point methods (Karmarkar [126]) overcome this weakness, and can solve polynomially sized

linear programs in polynomial time.[2] $\mathcal{U}_{LP(9.2)}$ still requires exponential running time compared to the original input G unless some structure in G allows the building of a model that uses only a polynomial number of constraints (see e.g. Grötschel et al. [99]).

We proved the running time of our domain filtering in lemma 6*(iv)* to be $O(|P|^2)$. Thus we can conclude: Domain filtering is at least as accurate as bounds $\mathcal{U}_1$–$\mathcal{U}_7$, and asymptotically does not require more running time than the more effective of these seven bounds. The only bounds that outdo domain filtering in certain situation require higher running times.

9.5 Conclusions

We presented some simple domain filtering that tightens the candidate set used in algorithms for solving maximum clique problems. Using a taxonomy of linear programming and combinatorial bounds for MC we were able to prove that the tightened candidate set is in a sense as least as tight as seven of these bounds. To our knowledge, the taxonomy of bounds was not presented before. A brief complexity analysis furthermore showed, that the asymptotic running time of domain filtering is comparable to the more elaborated bounds.

The quality of domain filtering strategies is often measured by the notion of consistency. We believe that a direct comparison to the pruning-power of OR bounds is more appropriate in the case of cost based filtering techniques. Our proposal thus is to compare OR bounds and cost based filtering techniques by their ability to prune the search tree.

We showed in Sec. 9.4.4 that domain filtering can outdo $\mathcal{U}_8$ and Lemma 9 showed that $\mathcal{U}_{LP(9.2)}$ is as least as good as $\mathcal{U}_8$. As an open question it remains to investigate the relation between domain filtering and $\mathcal{U}_{LP(9.2)}$, i.e. is $\mathcal{U}_{LP(9.2)}$ at least as good as domain filtering or are there any counterexamples?

9.5.1 Extensions to the Weighted Case

The *Weighted Maximum Clique Problem (WMC)* is a canonical extension of MC where each node $v \in V$ is assigned a weight $w(v)$, where $w : V \to \mathbb{N}_0$ is a weight function.

We can extend our domain filtering quite easily to the weighted case. Instead of counting the number of neighbors (degree), we have to use the sum of weights of all neighbors. Most proofs can be easily adapted. The taxonomy of bounds, and the relation between bounds and domain filtering, however, needs some more care, as some dominance relations change.

[2] The original algorithm of Karmarkar requires $O(n^{3.5}L)$ arithmetic operations on $O(L)$ bit numbers, where n is the number of variables and L is the number of bits in the input (see [126]).

10

Two Adapted Branch-and-Bound Algorithms for Maximum Clique

The results of the previous chapter suggest the beneficial use of domain filtering in an enumeration algorithm for MC. As it does not always dominate the coloring bound $\mathcal{U}_8$ we may improve convergence further by introducing some coloring heuristics that provide us with a valid upper bound. In so doing, we can also get a lower bound heuristic "almost for free". We will discuss these two coloring based ingredients first.

10.1 Upper and Lower Bounds by Coloring Heuristics

Determining the chromatic number is NP-hard on general graphs, and various proposals for heuristics and exact algorithms were made in literature. We refer to Johnson and Trick [122] for a recent overview, and mention two important approaches developed in the meantime: The exact branch-and-price algorithm of Mehrotra and Trick [150], and the fractional coloring heuristic of Balas and Xue [15]. Both solve the column generation model for the chromatic number problem (9.7) which was already mentioned in the proof of lemma 7.

In our approach, we use two different and fast running heuristics for the coloring problem. The first one follows the idea of the DSATUR approach of Brélaz [40], and was used in the clique algorithms of Babel [11], Babel and Tinhofer [12], Balas and Xue [15], and Wood [212]. It chooses a node with a maximum number of colored neighbors and assigns the smallest unused color to that node (see Algorithm 23). Line 5 allows different tie breaking strategies. The one originally proposed is to choose a node with a highest degree in case of a tie. We found in our experiments that on some instances, the opposite strategy i.e. choosing a node with a lowest degree is more helpful. To embrace both situations, we ran the algorithm twice, changing the tie breaking strategy in between both runs.

In the root node, we additionally use a randomized tie breaking strategy. In both sorting orders, we pick a node uniformly at random from those with the same number of colored

Algorithm 23 DSATUR greedy method

function ColorBound(graph $G = (V, E)$)
1: **for all** $v \in V$ **do**
2: $\quad$ $color[v] \leftarrow 0$ // *initialize colors*
3: $\quad$ $coloredNeighbors[v] \leftarrow 0$ // *initialize colored neighbors counter*
4: **while** (not all nodes colored) **do**
5: $\quad$ $v \leftarrow \arg\max\{coloredNeighbors[u] \mid color[u] = 0,\ u \in V\}$
6: $\quad$ $color[v] \leftarrow \min\{k \mid k \geq 1 \ \wedge\ k \neq color[u]\ \forall \{u, v\} \in E\}$
7: $\quad$ // *Update colored neighbors info*
8: $\quad$ **for all** $\{u, v\} \in E$ **do**
9: $\quad\quad$ $coloredNeighbors[v] \leftarrow coloredNeighbors[v] + 1$
10: **return** $\max\{color[v] \mid v \in V\}$

neighbors. We run the resulting algorithms $|V|/100$ times in order to obtain a good initial bound. We found that we can usually improve the incumbent upper bound once or twice. Unfortunately, we cannot use this approach in other nodes of the search tree as it is too time consuming.

The other coloring method was developed by Biggs [33], and has also been used in the clique solvers of Balas and Xue [15] and Wood [212]. It heuristically partitions the given graph into a small number of maximal independent sets. Assigning each set a different color yields an upper bound on the chromatic number. The algorithm starts with an empty set I_1 and includes nodes in this set until no more nodes can be included. Each time a set I_k cannot be extended, a new set I_{k+1} is opened and the algorithm continues until all nodes have been accounted for (see Alg. 24).

Algorithm 24 Bigg's Greedy Method for Graph Coloring

function ColorBound(graph $G = (V, E)$)
1: **for all** $v \in V$ **do**
2: $\quad$ $color[v] \leftarrow 0$ // *initialize colors*
3: $k \leftarrow 0$
4: **while** (not all nodes colored) **do**
5: $\quad$ $k \leftarrow k + 1$ // *we need one more color*
6: $\quad$ $W \leftarrow \{v \in V \mid color[v] = 0\}$
7: $\quad$ $I_k \leftarrow \emptyset$ // *next independent set*
8: $\quad$ **for** $v \in W$ in predetermined order **do**
9: $\quad\quad$ $color[v] \leftarrow k$
10: $\quad\quad$ $I_k \leftarrow I_k \cup \{v\}$
11: $\quad\quad$ $W \leftarrow W \setminus (N_G(v) \cup \{v\})$
12: **return** k

The order in which nodes are arranged has some impact on the bounds' quality. Therefore, in line 8 of algorithm 24 we apply nodes in increasing and in decreasing order of degrees, and obtain two different variants of the heuristic.

Both algorithms can be implemented such that their running time is $O(|V|^2)$ which is optimal on dense graphs. The final upper bound is taken as the minimum number found by any

of the four variants sketched above.

Findings of Balas and Xue [15] and Wood [212] show that Alg. 23 outperforms Alg. 24 with respect to quality in many cases, but is slower than the latter one. Within a branch-and-bound approach, and in combination with other methods, the interference is more complicated. Our experiments showed that using both methods costs more running time in some cases, but in the majority of cases it helps to improve convergence.

10.1.1 Lower Bounds

Algorithm 23 can also produce lower bounds, i.e. primal solutions to the clique problem. The idea is that each time we have to open a new color class (lines 5–7), the nodes in W are in conflict with all their neighbors.

Starting with color class 1, we store those nodes opening the next color class that are also adjacent to all other nodes stored so far. All additional checks for the upper bound can be performed in $O(|V|^2)$, and the overall complexity of Alg. 23 remains unchanged. As a result, a lower bound on a maximum clique is found. This idea was already used by Wood [212].

10.2 Improving the Algorithms

10.2.1 Adapting the Carraghan and Pardalos Algorithm

Using the domain filtering introduced in chapter 9 and the coloring heuristics discussed before, we are able to improve the Carraghan and Pardalos algorithm:

Just before branching (line 8 of the Alg. 20) we call our domain filtering. We know that the next improvement has to be better than the best clique found so far. Hence, we use $(|C^*|+1)$ as a lower bound for the necessary checks. If still $P' \neq \emptyset$ we calculate the coloring bounds (both, upper and lower) as described above. If the upper bound prunes the current part of the search tree, we skip the next recursive call.

10.2.2 Adapting the Östergård Algorithm

Similarly, we can adapt Östergård's method: We use domain filtering just before line 11 in Alg. 21 and use coloring bounds afterwards if P still contains some elements. However, it is not straightforward to use lower bound information in Östergård's approach. In fact, Östergård addresses this as a problem in [158] but does not come up with a solution (nor do we). The critical point is the fact that the values $\beta[v]$ will loose their significance when using additional lower bounds: Assume we found a clique of size k by some heuristic. Now, if we stop searching for all subgraphs of a smaller or equal size than k the corresponding bounds $\beta[v]$ do not contain helpful information for subsequent iterations, and convergence slows down.

10.2.3 The Implementation

An efficient implementation makes use of some more "tricks" than those visible in the algorithmic descriptions 20 and 21. The gains are constants only. But neglecting those might

make the difference between solving or not solving a problem within a time limit.

When determining $P' = P \cap N(v)$, we use a loop running over all elements $p \in P$. If $\{p, v\} \in E$, p is copied to P'. For non-adjacent elements we decrease a counter m that contains the number of elements still needed in order to obtain $|C'| + |P'| > |C^*|$. Should m ever be decremented below zero we immediately skip that recursion-level. (We refer to [7] for more details).

For the domain filtering algorithm we also update degree information on the subgraph induced by P. That is, we decrease the degree of a node $p' \in P'$ if an adjacent node $p \in P$ is not copied to P'. By keeping a copy of P in each recursion level we can easily perform incremental updates on the degrees before entering the next recursion level.

10.3 Numerical Results on Benchmark Graphs

Obviously, as the tighter bounds need some more computational effort, there is a trade-off between the effectiveness of pruning techniques and the overall efficiency of the approach. In this section we examine the empirical behavior of the new approaches and we compare our results with those of some recent solvers for MC.

We use the well-established DIMACS benchmark set [122, 67] for our comparison. It consists of 66 instances, ranging from 28 – 3361 nodes, and 420 – 11 million edges. The density of the underlying graphs ranges from 3.5% – 99.8%. More details are given in Appendix B.1. To our knowledge optimal solutions have not yet been proved for some of these instances. A second set of numerical tests was performed on random graphs.

All algorithms were coded in C++ and compiled by the GNU g++ 2.95.3 compiler using full optimization. Our benchmark tests were run on a Dual-Pentium III-933 PC with 512MB RAM operating Linux 2.4.19. We stopped each run after 6 hours (21 600 sec). Only one processor per machine was used during the tests.

Definition 14 *We say that on a graph G an algorithm A is better than an algorithm B if*

(i) A found a larger clique than B, or

(ii) cliques found have equal size, but A terminates faster than B, or

(iii) clique size and runtime for A and B are identical, but A found a best solution earlier than B.

The latter case usually occurs if both approaches do not terminate within the given time limit. All runtimes are measured as process times. Regarding the remaining timing problems (see Sec. 2.4.3) we use (2.23) in the comparisons.

10.3.1 Domain Filtering

Our first set of experiments only considers the effects of domain filtering and coloring bounds. In tables 10.1 and 10.2 we compare our implementation[1] of `dfmax` using bound $\mathcal{U}_1$ (Alg. 20)

[1] For a fair comparison, we decided to use identical subroutines for all implementations. We adapted `dfmax` to our settings resulting in a slightly faster algorithm than the original `dfmax`

only, an implementation using coloring bounds only (χ), another using domain filtering only (DF), and a fourth using both ($\chi + DF$).

As expected, the number of choice points decreased considerably when domain filtering was applied. The savings range from a factor of 3.6 (MANN_a9) to a factor of 30.4 (p_hat1500_1) when comparing pure `dfmax` to `dfmax`+DF. In contrast, in many cases the running time of the current code is twice as slow as of the `dfmax` code. Here, domain filtering is computationally too expensive compared to the simple $\mathcal{U}_1$ bound: It is still cheaper to traverse large useless parts of the search tree than to detect these parts. These observations are confirmed by numerical results of Alg. 21.[2] Though the numbers vary, the general effects are the same: The number of choice points is considerably reduced. On the other hand, the original approach is usually faster than the same approach plus domain filtering. Interestingly, Table B.2 shows that the pure approach alone as well as the pure approach enhanced by either coloring or domain filtering finds a total clique size of about 3840. Only when combining Alg. 21 with coloring bounds and domain filtering the accumulated clique size goes up to over 4000 (this improvement is due to small improvements on several instances and thus is not a singular effect). Detailed tables are given in Appendix B.2.

The c-fatxxx instances play a special role in this context. They are quite easy to solve using domain filtering. Each instance requires only 5 choice points to prove optimality, with running time being negligible. Algorithm 20 alone uses dramatically more choice points (a factor of more than one million in the case of c-fat200-5, and more than 10^9 for c-fat500-10). Coloring bounds reduce the number of choice points in these instances as well, but are not as efficient as domain filtering. This corresponds to the findings of Wood [212] who also reports only a small number of choice points (1–27) for his approach on those instances. For algorithm 21 the number of choice points decreases, when using domain filtering, and for c-fat200-2 at least, the resulting gain is substantial.

10.3.2 Coloring Bound $\mathcal{U}_8$

The use of vertex color bounds significantly helps to detect useless parts of the search tree. Out of the 66 benchmark instances, 46 could be solved to optimality when using coloring bounds, whereas only 33 can be finalized by Alg. 20, and 34 by Alg. 21, respectively. Again, it turns out that DF helps to reduce choice points. Additionally, a positive impact on running time is observed. Since the coloring bounds are rather expensive to compute, any reduction in the remaining graph improves the overall running time. As a result, $DF + \chi$ finds larger cliques in three cases, and in 44 out of 66 cases domain filtering shows improvements over those which do not use domain filtering (according to Def. 14). We applying Östergård's idea plus $DF + \chi$ two larger cliques are found than by running that setting without domain filtering. All in all, 40 instances benefit from additional domain filtering (measured as described above).

It should be noted that for simple instances especially running time is negatively affected by more sophisticated approaches (e.g. some of the p_hatxx instances). For those cases a control mechanism that switches off expensive techniques should be used.

[2] We used our own implementation in the tests. Östergård provides his implementation of Alg. 21 only for the more general case of *weighted* cliques.

Instance	Alg. 20			Alg. 20+DF			Alg. 20+χ			Alg. 20+DF+χ		
	$\lvert C^*\rvert$	time	BB nodes	$\lvert C^*\rvert$	time	BB nodes	$\lvert C^*\rvert$	time	BB nodes	$\lvert C^*\rvert$	time	BB nodes
brock200_1	**21**	**26.58**	**38043497**	21	54.60	3350353	21	83.36	32847	21	73.99	30811
brock200_2	**12**	**0.04**	**54314**	12	0.17	3539	12	0.49	323	12	0.52	285
brock200_3	**15**	**0.37**	**453265**	15	1.04	32170	15	3.15	1418	15	2.68	1356
brock200_4	**17**	**1.61**	**2161909**	17	3.87	167750	17	9.73	4442	17	10.09	4169
brock400_1	**≥27**	**≥21600**	**23223752963**	≥25	≥21600	1022982977	≥25	≥21600	6477006	≥25	≥21600	5677245
brock400_2	**≥29**	**≥21600**	**20089180565**	≥29	≥21600	760111221	≥29	≥21600	4976220	≥29	≥21600	4429451
brock400_3	**≥31**	**≥21600**	**25770803689**	≥24	≥21600	1057635922	≥24	≥21600	6672788	≥24	≥21600	5830338
brock400_4	**33**	**19773.90**	**20152485865**	≥25	≥21600	935193147	≥25	≥21600	5851408	≥25	≥21600	5153284
brock800_1	**≥21**	**≥21600**	**21101899632**	≥21	≥21600	546659121	≥21	≥21600	6705439	≥21	≥21600	5862067
brock800_2	**≥21**	**≥21600**	**23496231361**	≥20	≥21600	636709653	≥20	≥21600	7261032	≥20	≥21600	6381751
brock800_3	**≥21**	**≥21600**	**22124014822**	≥20	≥21600	602902311	≥20	≥21600	7089985	≥20	≥21600	6112439
brock800_4	**≥26**	**≥21600**	**21662133367**	≥21	≥21600	606259112	≥20	≥21600	7239813	≥20	≥21600	6176011
c-fat200-1	12	0.01	52	**12**	**0.01**	**5**	12	0.01	34	**12**	**0.01**	**5**
c-fat200-2	24	0.01	444	**24**	**0.01**	**5**	24	0.01	70	**24**	**0.01**	**5**
c-fat200-5	58	1207.72	268435599	**58**	**0.01**	**5**	58	0.08	172	**58**	**0.01**	**5**
c-fat500-1	**14**	**0.01**	**71**	14	0.05	5	14	0.05	40	14	0.05	5
c-fat500-10	≥124	≥21600	15858231004	**126**	**0.07**	**5**	126	0.68	376	**126**	**0.07**	**5**
c-fat500-2	**26**	**0.01**	**683**	26	0.05	5	26	0.06	76	26	0.07	5
c-fat500-5	64	4.92	5703074	64	0.06	5	64	0.15	190	**64**	**0.03**	**5**
hamming10-2	≥512	≥21600	3366474297	≥512	≥21600	770355321	512	0.76	513	**512**	**0.67**	**257**
hamming10-4	≥32	≥21600	36567495385	≥32	≥21600	2815903831	≥33	≥21600	5237411	**≥33**	**≥21600**	**4495000**
hamming6-2	32	0.01	42787	32	0.01	1891	32	0.01	33	**32**	**0.01**	**17**
hamming6-4	4	0.01	221	4	0.01	47	4	0.01	129	**4**	**0.01**	**25**
hamming8-2	≥128	≥21600	14850398859	≥128	≥21600	1375703771	128	0.01	129	**128**	**0.01**	**65**
hamming8-4	16	4.06	3742143	16	7.89	252418	**16**	**3.29**	**585**	16	5.44	570
johnson16-2-4	**8**	**1.54**	**4177630**	8	3.62	904446	8	11.21	123282	8	17.73	57912
johnson32-2-4	**≥16**	**≥21600**	**49208418864**	≥16	≥21600	4929133669	≥16	≥21600	228095779	≥16	≥21600	70631873
johnson8-2-4	4	0.01	90	4	0.01	21	4	0.01	50	**4**	**0.01**	**11**
johnson8-4-4	14	0.01	12544	14	0.01	1205	14	0.03	27	**14**	**0.01**	**17**
keller4	**11**	**0.74**	**1275236**	11	1.56	107086	11	2.58	1598	11	3.65	1476
keller5	≥24	≥21600	26405722472	≥24	≥21600	787089454	≥26	≥21600	2844759	**≥26**	**≥21600**	**2423520**
keller6	≥43	≥21600	19876767579	≥43	≥21600	574383274	≥43	≥21600	1695034	**≥43**	**≥21600**	**1341291**
MANN_a27	≥116	≥21600	17111883459	≥116	≥21600	22543342098	126	3627.65	31978	**126**	**3046.67**	**31240**
MANN_a45	≥220	≥21600	12427538213	≥234	≥21600	22952851665	≥334	≥21600	26007	**≥336**	**≥21600**	**24794**

Table 10.1: *Algorithm 20 on DIMACS Instances (Part I). We compare the number of choice points and running time in seconds of our implementation of* `dfmax`*, our algorithm using coloring bounds only (χ), our algorithm using domain filtering only (DF), and a fourth using both ($\chi + DF$). Boldface numbers indicate best runs.* *(see also summary on page 192).*

Instance	Alg. 20			Alg. 20+DF			Alg. 20+χ			Alg. 20+DF+χ		
	$\lvert C^* \rvert$	time	BB nodes	$\lvert C^* \rvert$	time	BB nodes	$\lvert C^* \rvert$	time	BB nodes	$\lvert C^* \rvert$	time	BB nodes
MANN_a81	≥438	≥21600	6433965301	≥467	≥21600	22372923762	≥998	≥21600	33067	**≥998**	**≥21600**	**9637**
MANN_a9	16	0.09	329235	16	0.11	92210	16	0.01	46	**16**	**0.01**	**28**
p_hat1000-1	**10**	**2.05**	**1801019**	10	9.74	81573	10	27.45	14082	10	35.28	13303
p_hat1000-2	≥42	≥21600	20151659983	≥41	≥21600	1095205574	≥44	≥21600	1003624	**≥44**	**≥21600**	**1341279**
p_hat1000-3	≥49	≥21600	22757998877	≥47	≥21600	1426408001	≥50	≥21600	921339	**≥52**	**≥21600**	**1274139**
p_hat1500-1	**12**	**18.28**	**13825580**	12	92.85	454914	12	292.45	104668	12	380.57	101895
p_hat1500-2	≥46	≥21600	24297112095	≥46	≥21600	1848437250	≥52	≥21600	673044	**≥52**	**≥21600**	**783463**
p_hat1500-3	≥53	≥21600	27789610940	≥53	≥21600	2097394574	≥58	≥21600	983802	**≥58**	**≥21600**	**1148829**
p_hat300-1	**8**	**0.01**	**11634**	8	0.07	450	8	0.19	237	8	0.36	212
p_hat300-2	**25**	**1.00**	**1553140**	25	2.67	182168	25	2.38	647	25	3.61	438
p_hat300-3	36	1492.43	1960518988	36	2729.49	251846120	36	508.07	79793	**36**	**482.81**	**71863**
p_hat500-1	**9**	**0.08**	**89908**	9	0.51	4978	9	1.21	493	9	1.46	475
p_hat500-2	36	240.41	292274682	36	502.24	35324032	36	128.09	16235	**36**	**128.32**	**15101**
p_hat500-3	≥44	≥21600	25719412910	≥43	≥21600	1873805007	≥49	≥21600	1052581	**≥49**	**≥21600**	**1218440**
p_hat700-1	**11**	**0.36**	**343857**	11	2.00	20630	11	3.96	871	11	5.14	815
p_hat700-2	44	9742.35	10969888234	44	18281.20	1394776054	44	1129.18	89869	**44**	**1091.35**	**79390**
p_hat700-3	≥50	≥21600	24323367924	≥49	≥21600	1554061118	≥54	≥21600	801496	**≥56**	**≥21600**	**943799**
san1000	≥10	≥21600	12277896744	≥9	≥21600	179990506	**15**	**895.33**	**35573**	15	1141.11	31477
san200_0.7_1	30	5466.91	14265131069	30	9273.97	3805796652	30	0.90	389	**30**	**0.89**	**268**
san200_0.7_2	≥18	≥21600	31159815462	≥18	≥21600	4588848753	18	0.58	277	**18**	**0.60**	**176**
san200_0.9_1	≥48	≥21600	47701178923	≥48	≥21600	14808252136	70	30.97	5308	**70**	**24.54**	**4291**
san200_0.9_2	≥41	≥21600	48882914444	≥41	≥21600	3462650501	60	628.92	85085	**60**	**600.46**	**72365**
san200_0.9_3	≥36	≥21600	2192973297	≥44	≥21600	1651744796	**44**	**57.87**	**12622**	44	71.55	11830
san400_0.5_1	13	916.73	1414713059	13	2255.23	155275559	**13**	**2.59**	**2470**	13	3.64	739
san400_0.7_1	≥22	≥21600	87494021010	≥22	≥21600	16652377204	**40**	**123.09**	**7438**	40	150.44	8576
san400_0.7_2	≥17	≥21600	61966653095	≥17	≥21600	7525272755	**30**	**44.96**	**6497**	30	62.94	10129
san400_0.7_3	≥22	≥21600	26018343618	≥17	≥21600	1972475939	**22**	**290.52**	**65466**	22	367.05	84602
san400_0.9_1	≥49	≥21600	54964009156	≥49	≥21600	6047721579	100	2310.99	144664	**100**	**2033.07**	**120402**
sanr200_0.7	**18**	**5.83**	**7985123**	18	12.15	665013	18	27.65	12985	18	28.05	12246
sanr200_0.9	≥40	≥21600	29331657711	≥40	≥21600	2144307525	42	11884.40	1733080	**42**	**10263.50**	**1421500**
sanr400_0.5	**13**	**4.06**	**4283999**	13	12.16	251463	13	41.43	18084	13	44.22	17439
sanr400_0.7	**21**	**4587.91**	**5624332491**	21	9373.51	408188818	21	14585.70	5067837	21	14900.60	4783079

Table 10.2: *Algorithm 20 on DIMACS Instances (Part II). We compare the number of choice points and running time in seconds of our implementation of* `dfmax`*, our algorithm using coloring bounds only (χ), our algorithm using domain filtering only (DF), and a fourth using both ($\chi + DF$). Boldface numbers indicate best runs.* *(see also summary on page 192).*

10.3.3 Primal Heuristics

Compared to the original algorithm lower bound heuristics improve in 52 out of 66 cases. The gain in runtime, though, is not very dramatic (197 hours versus 217 hours, accumulated for all 66 instances). Using primal heuristics, benchmark instances c-fat500-10, hamming10-2, and hamming8-2 can be solved in in a fraction of a second. The heuristics find the optimal clique size already in the root node. The original approach cannot finalize its runs within a time limit of 6 hours. Instance p_hat700-2, on the other hand, is solved in less than 10 000 seconds by the original approach, but cannot be solved to optimality when using lower bound heuristics.

In 43 cases using the primal heuristic is best, in 23 not using it is best (according to Def. 14). Improvements stem from two facts: Primal heuristics provide a good primal solution and thus support pruning. They also provide tight bounds for being used in domain filtering, thus making the filtering more effective.

By combining primal heuristics with domain filtering and coloring (see table 10.3) we can produce the best results.

10.3.4 Comparison to other Algorithms

We also compare our results to some recent results presented in literature. We compare the results of Wood [212], Balas and Xue [15], and Östergård [158] with our χ+DF+primal heuristics results (Table 10.3). Since the results in the first column of tables 10.1 and 10.2 correspond to the Carraghan/Pardalos algorithm, we have already shown, that we outdo that algorithm in many cases.

Unfortunately, different machines as well as different time limits were used for the other approaches. Hence, we can only try to show a general tendency, rather than comparing details. Also, the papers mentioned do not present results on all 66 instances, whereas we have good results on some of the omitted instances. Using the ingredients described above, e.g., best clique size found increases from 220 to 342 for MANN_a45 and from 438 to 1096 for MANN_a81, respectively. Instances sanr200_0.9 and san200_0.9_3 cannot be finalized by the original algorithm, whereas our combined approach proves optimality within a short space of time. It would therefore be interesting to see the performance of other approaches on the omitted instances as well.

Wood's approach is a branch-and-bound using fractional coloring and lower bound heuristics. Results for 38 DIMACS instances are presented and solved within running times of up to 19 hours on a SUN S10. In 30 cases our approach uses fewer choice points than his fastest method (MC_C), the typical factor being 4–8. In 4 cases we use more, in another 4 cases we need the same number of choice points. When considering MC_D (the approach using the least number of choice points), our approach is better in 19 cases, MC_D has less choice points in 9 cases, the remaining 10 cases all solve the problem in the root node. However, in at least 28 cases the running time is higher than ours.[3] In 6 cases we need more computing time.

Balas and Xue developed a heuristics to determine fractional coloring bounds within a branch-and-bound framework. They used a DEC Alpha 300–400 AXP and time limit of 5 hours. In 29 out of 48 cases we use fewer choice points than their approach. Assuming that their computer is about 4 times slower than ours, we outperform that algorithm in 11 cases.

[3] A SUN Sparc 10/51 is about 10 times slower than our computer, and we transformed the running times by this factor

Instance	Alg. 20+ DF+χ+Heuristic			
	$\lvert C^* \rvert$	time	BB nodes	time best
brock200_1	21	83.81	30694	16.10
brock200_2	**12**	**0.54**	**272**	**0.14**
brock200_3	15	3.26	1356	0.01
brock200_4	**17**	**10.02**	**4123**	**7.29**
brock400_1	≥25	≥21600	6032026	16099.20
brock400_2	≥29	≥21600	4583263	8746.22
brock400_3	≥24	≥21600	6074490	52.50
brock400_4	≥25	≥21600	5309119	1245.55
brock800_1	≥21	≥21600	6108652	12905.80
brock800_2	≥20	≥21600	6699015	1662.42
brock800_3	≥20	≥21600	6504352	560.13
brock800_4	≥20	≥21600	6571649	190.12
c-fat200-1	**12**	**0.01**	**0**	**0.01**
c-fat200-2	**24**	**0.01**	**0**	**0.01**
c-fat200-5	**58**	**0.01**	**1**	**0.01**
c-fat500-1	**14**	**0.01**	**0**	**0.01**
c-fat500-10	**126**	**0.01**	**0**	**0.01**
c-fat500-2	**26**	**0.01**	**0**	**0.01**
c-fat500-5	**64**	**0.01**	**0**	**0.01**
hamming10-2	**512**	**0.04**	**0**	**0.03**
hamming10-4	≥33	≥21600	4692037	17301.70
hamming6-2	**32**	**0.01**	**0**	**0.01**
hamming6-4	**4**	**0.01**	**22**	**0.01**
hamming8-2	**128**	**0.01**	**0**	**0.01**
hamming8-4	**16**	**3.75**	**556**	**0.01**
johnson16-2-4	**8**	**12.39**	**57881**	**0.01**
johnson32-2-4	**≥16**	**≥21600**	**72262348**	**0.01**
johnson8-2-4	**4**	**0.01**	**11**	**0.01**
johnson8-4-4	**14**	**0.01**	**11**	**0.01**
keller4	**11**	**2.66**	**1457**	**0.03**
keller5	**≥26**	**≥21600**	**2490605**	**18977.10**
keller6	**≥43**	**≥21600**	**1344053**	**796.79**
MANN_a27	**126**	**2984.11**	**30458**	**283.11**

Instance	Alg. 20+ DF+χ+Heuristic			
	$\lvert C^* \rvert$	time	BB nodes	time best
MANN_a45	**≥342**	**≥21600**	**11883**	**0.05**
MANN_a81	**≥1096**	**≥21600**	**462**	**0.43**
MANN_a9	**16**	**0.01**	**18**	**0.01**
p_hat1000-1	**10**	**32.57**	**13282**	**2.71**
p_hat1000-2	**≥44**	**≥21600**	**1263289**	**4680.78**
p_hat1000-3	**≥57**	**≥21600**	**658102**	**0.08**
p_hat1500-1	**12**	**341.15**	**101886**	**67.74**
p_hat1500-2	**≥54**	**≥21600**	**616341**	**0.11**
p_hat1500-3	**≥75**	**≥21600**	**266476**	**0.12**
p_hat300-1	**8**	**0.24**	**207**	**0.15**
p_hat300-2	**25**	**2.41**	**333**	**1.59**
p_hat300-3	**36**	**487.80**	**71384**	**484.66**
p_hat500-1	**9**	**1.46**	**462**	**0.18**
p_hat500-2	**36**	**127.90**	**14585**	**127.88**
p_hat500-3	**≥49**	**≥21600**	**1108105**	**9059.61**
p_hat700-1	11	5.69	802	5.43
p_hat700-2	44	1111.40	76079	1048.25
p_hat700-3	**≥56**	**≥21600**	**577935**	**14274.40**
san1000	15	1476.19	31290	1476.17
san200_0.7_1	**30**	**0.58**	**154**	**0.58**
san200_0.7_2	**18**	**0.51**	**107**	**0.51**
san200_0.9_1	**70**	**7.13**	**465**	**7.12**
san200_0.9_2	60	651.72	71274	651.70
san200_0.9_3	44	86.09	11602	80.90
san400_0.5_1	13	4.95	694	4.89
san400_0.7_1	40	168.33	8449	168.31
san400_0.7_2	**30**	**63.87**	**9062**	**63.86**
san400_0.7_3	22	403.95	84522	403.95
san400_0.9_1	**100**	**2011.49**	**85997**	**2010.11**
sanr200_0.7	18	29.23	12198	8.69
sanr200_0.9	42	11082.70	1421874	7516.34
sanr400_0.5	13	45.72	17424	23.78
sanr400_0.7	21	15357.30	4783053	43.39

Table 10.3: *Comparison of Algorithm 20 plus domain filtering, coloring bounds, and primal heuristics. Boldface numbers indicate best runs in comparison to the same setting without primal heuristics (fourth column in tables 10.1, 10.2).*

Balas and Xue's approach is faster in 29 cases. A closer analysis indicates that their primal heuristic is much better than ours.

Östergård's approach was already described in Sec. 9.2. We compare our results to those published in Östergård [158], since results of our own implementation deviates from those given there. As no numbers of choice points are given, we can only compare running times. The computer used is a 500 MHz PC, roughly 2 times slower than ours. Östergård considers 38 DIMACS instances, of which we can solve 17 faster than his approach, whereas his approach is faster in 16 cases. On a majority of instances our own implementation of his idea is up to an order of magnitude slower than Östergård's original code. However the speed-up when combined with coloring bounds and domain filtering is significant. Thus, we expect to speed-up the original code of Östergård by combining it with coloring bounds and domain filtering.

10.3.5 Statistics on Large Sample Sets

In order to get a better idea of the impact of different techniques we also performed some statistical evaluation based on the entire data found when running our different algorithms on the DIMACS benchmark set. I.e. we accumulate data using a certain technique and compare it to a collection of data which does not use that technique. Detailed tables for all tests are given in Appendix B. Data stems from the tests presented in this thesis as well as from 8 tests using different node orderings. Node reordering follows an idea presented in Fahle [70]. In a more intensive study it turned out to have only a minor impact on the overall convergence when combined with additional techniques.[4]

As usual, some care is needed when interpreting statistics like these, since the experimental setting (time limit, good improvement on single instances only, etc.) influences results.

10.3.5.1 Domain Filtering versus no Domain Filtering

There are ten settings using domain filtering, and another ten not using it. This results in a sample of 660 experiments for either test class. The table in Appendix B.4 shows no clear difference in runtime or accumulated clique size, when these two categories are compared (using DF gains ≈ 25h, but on some instances worse solutions are found). A few more best results are found by DF (639 vs. 606), and DF clearly needs fewer branch and bound nodes for these results (in 418 cases, whereas not using DF is better in 88 cases). Overall, DF is better in 426 cases, not using DF pays off in 274 cases.[5]

10.3.5.2 Coloring Bounds versus no Coloring Bounds

Coloring bounds have a stronger influence on the numerical results than domain filtering. In Appendix B.4 the difference in runtime and clique size is obvious and is due to a significant decrease in running time for long running instances. On short running instances, however, coloring bounds have a negative impact — the best clique sizes are more often found by algorithms not using coloring bounds (481 vs. 292 cases). On the other hand, the number of

[4] Therefore, we do not present details on this idea here. Appendix B.3 contains some more insight.

[5] In some cases, both strategies produce identical numbers which explains that the sum of these values is larger than 660.

branch-and-bound nodes decreases. Overall, coloring bounds find better results in 424 cases, whereas not using them proves to be better in 304 cases.

10.3.5.3 Domain Filtering plus Coloring Bounds versus Either of Both

The results presented before suggest the investigation of the combination of DF and coloring bounds. 330 instances either use both or exactly one of the techniques in question, and the results are presented in Appendix B.4. Clearly, coloring bounds turn out to be the most important aspects. When not using them runtime goes down from 1010.65 hours to 649.49 hours. Also, quality of the solutions increases. However, not using domain filtering reduces the number of cases, where fewer branch-and-bound nodes are needed. More importantly, only 126 of the best results are found by using coloring bounds without domain filtering, whereas 226 are found using both techniques.

10.3.5.4 Lower Bound Heuristics

When analyzing the 528 tests using or not using heuristics, respectively, it turns out that the overall runtime does not differ much (Appendix B.4). The quality of solutions, on the other hand, benefits from using heuristics. 389 of the best solutions are found using heuristics, but only 142 when not applying them. Slightly more benchmark instances terminate earlier without primal heuristics showing that heuristics produce a significant runtime overhead. All in all, lower bound heuristics are helpful, if they find a good or optimal solution rather early. In other cases, they slow-down convergence. Also, the positive impact of lower bound heuristics diminishes when other techniques are combined with the approach. Using domain filtering and bound $\mathcal{U}_8$ in addition, lower bound heuristics can only improve clique size in five cases, whereas they slow others down. Using the heuristics in that setting is still better in 43 cases, but accumulated runtime is slightly lower, when heuristics are not used (see tables in AppendixB.3).

10.4 Numerical Results on Random Graphs

By running experiments on random graphs of varying density, we can perform scaling tests. This allows a different view of the characteristics of algorithmic techniques. In particular, for each $d \in \{0, 2, 4, \dots, 100\}$ we ran our algorithms on 50 different random graphs of density d.[6] By drawing the arithmetical mean of running time and branch-and-bound nodes, respectively, we can illustrate the characteristics of the techniques discussed before.

We present some results for graphs with 100 and 150 nodes, respectively.[7] For each instance a maximum of 90 000 seconds was allowed, and some dense graphs could not be solved within

[6] The random graphs were constructed by a simple scheme: We ran over all possible $\frac{n(n-1)}{2}$ edges and installed an edge with probability d.

[7] A larger sample size, as well as larger graphs would be desirable for this test. Unfortunately, running times for graphs with the 150 node were considerably large already. Ten computers took about two months of computing time each. Some initial experiments with graphs having 200 nodes each gave running time up to 12 times larger than for the small graphs. Thus, about 120 computers would be needed to finalize that experiment in a reasonable amount of time.

these 25 hours. In that case, we used 90 000 seconds, and the number of branch-and-bound nodes used so far in the statistics.

The upper diagram in figure 10.1 shows runtime and the number of branch-and-bound nodes for random graphs having 100 nodes each. There are two general tendencies: Methods not using coloring bounds have a high peak on graphs with density $d = 96$. Using coloring bounds, runtime grows slower, and the peak of that curve is at about $d = 90$. For graphs having density of about 85% or less, overall runtime is lower when not calculating coloring bounds. In other words: Determining coloring bounds (on these graphs) pays off only, if they can help to reduce the number of branch-and-bound nodes by more than a factor of 100 – 1000 (lower diagram). The same diagram also shows a significant difference for approaches using simple bounds together with domain filtering and those not using domain filtering. Also conspicuous is the impact of domain filtering on low density graphs. Approaches not using domain filtering do use a higher amount of branch-and-bound nodes for graphs having density $d \leq 15$. The pure coloring approach, as well as the "coloring+heuristics" approach show an interesting up-and-down of the number of branch-and-bound nodes needed. These effects also occur for larger graphs (see figures 10.2,10.3). The reason is that on sparse graphs wrong decisions made by the coloring heuristics cannot be balanced later.

Graphs with 150 nodes (Figures 10.2,10.3) show the effects described above, too. For a better readability, we reduced the number of approaches not using coloring bounds. The upper part shows two diagrams corresponding to the 100 node graphs. The vertical line for the Östergård approaches is due to the fact that Alg. 21 performs a search for each node in the graph. Thus, at least 150 branch-and-bound nodes are required for these approaches. On sparse graphs, 150 – 200 branch-and-bound nodes are also sufficient to finalize a complete search.

In the lower part, runtime and number of branch-and-bound, respectively for the faster approaches is drawn on a linear-scaled y-axis. This allows us to investigate the slight differences between the faster approaches. Our implementation of Östergård approaches seems to be fastest if being combined with coloring bounds and domain filtering. Dfmax + domain filtering and coloring is as good as the same combination using lower bound heuristics in addition on random graphs.

For Figures 10.1 and 10.2, detailed diagrams for each method are given in Appendix B.5.

10.5 Conclusions

In an experimental study we investigated the impact of domain filtering on two different branch-and-bound approaches proposed for MC. We showed significant reduction in the number of choice points when using the new approach. As a drawback, we observed that domain filtering is time consuming and does rarely pay off when being applied alone. In combination with tight upper bounds, however, it is an important technique. Domain filtering then supports finding best solutions and accelerates search: Any node fixed by domain filtering needs not to be colored by the heuristics. As their running time is $O(|P|^2)$ this directly impacts on the overall running time. Moreover, any node fixed cannot be assigned a wrong color, thus the heuristics tend to produce tighter bounds which improves the search.

Using primal (lower bound) heuristics further improves the algorithm, and the combination of tight lower and upper bounds, and domain filtering is the most promising approach tested

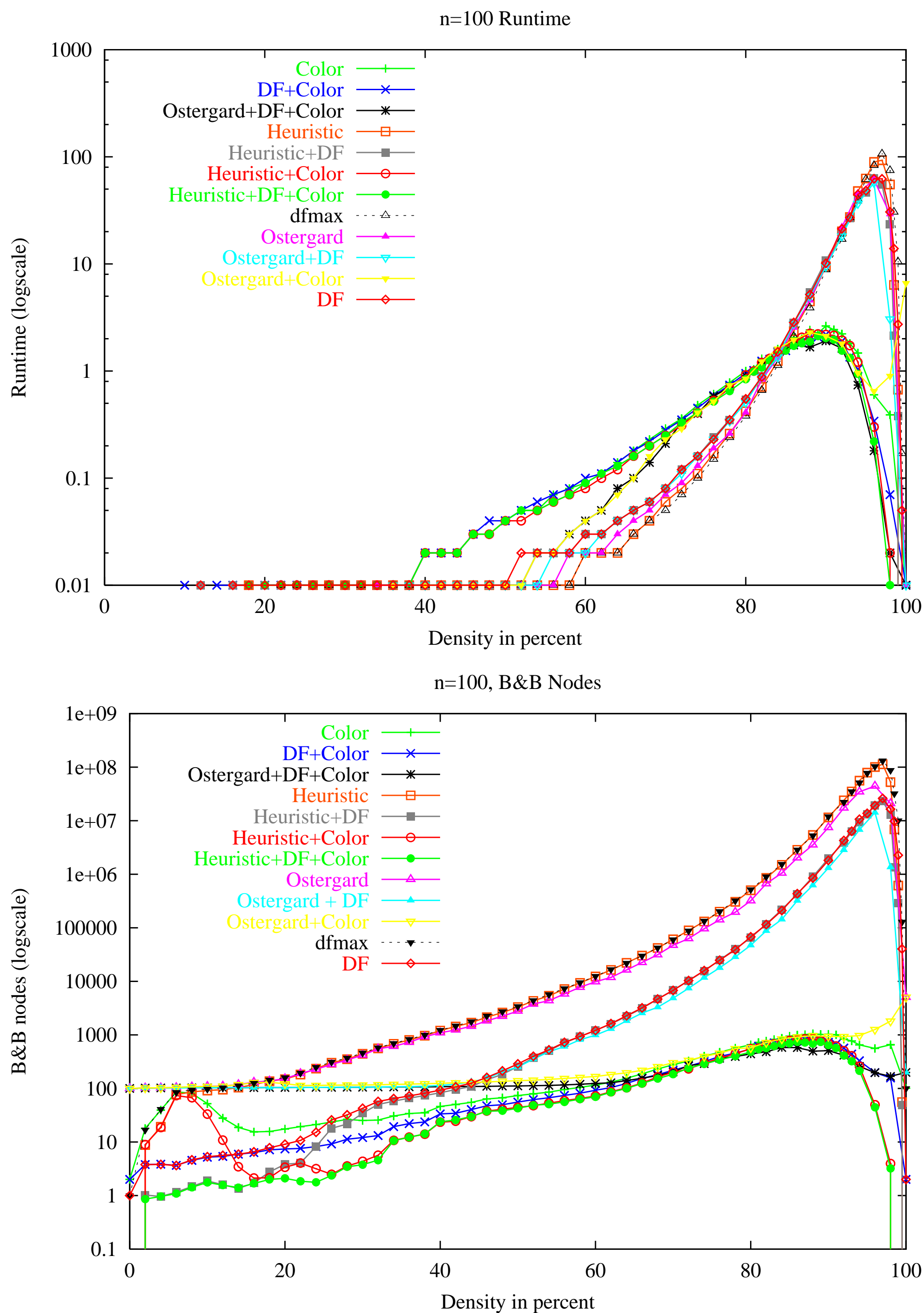


Figure 10.1: *Random graphs with 100 nodes. Runtime and the number of branch-and-bound nodes, respectively, given on the y-axis in logarithmic scale. On the x-axis, graph density varies from 0% – 100%. Lines connecting measuring points are given for easier readability and do not mark intermediate values. (see also Appendix B.5.)*

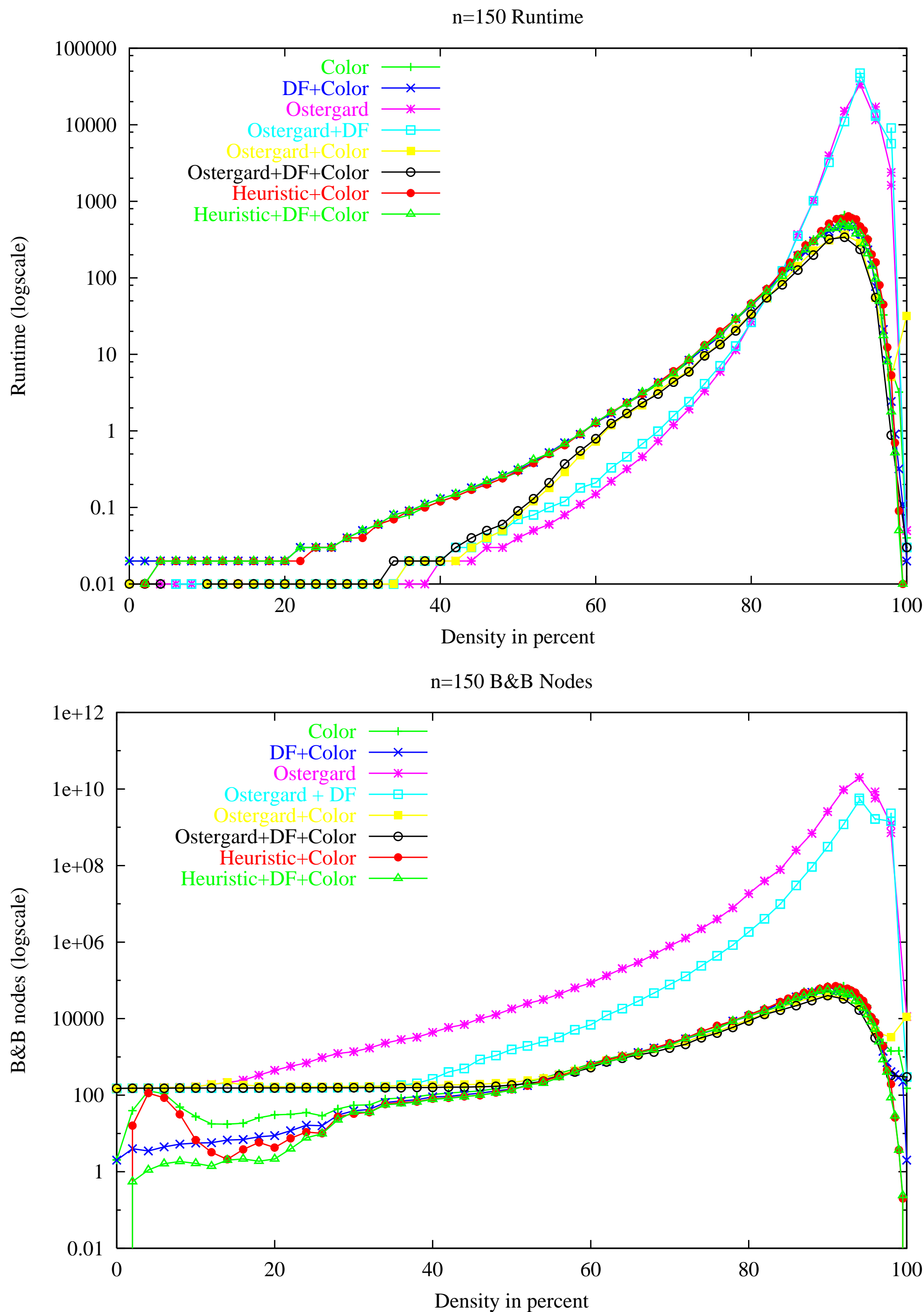


Figure 10.2: *Random graphs with 150 nodes. The y-axis gives runtime and number of branch-and-bound nodes, respectively. The y-axis is scaled logarithmic On the x-axis, graph density varies from 0% – 100%. Each sample point represents an average value of 50 instances. (see also Appendix B.5.)*

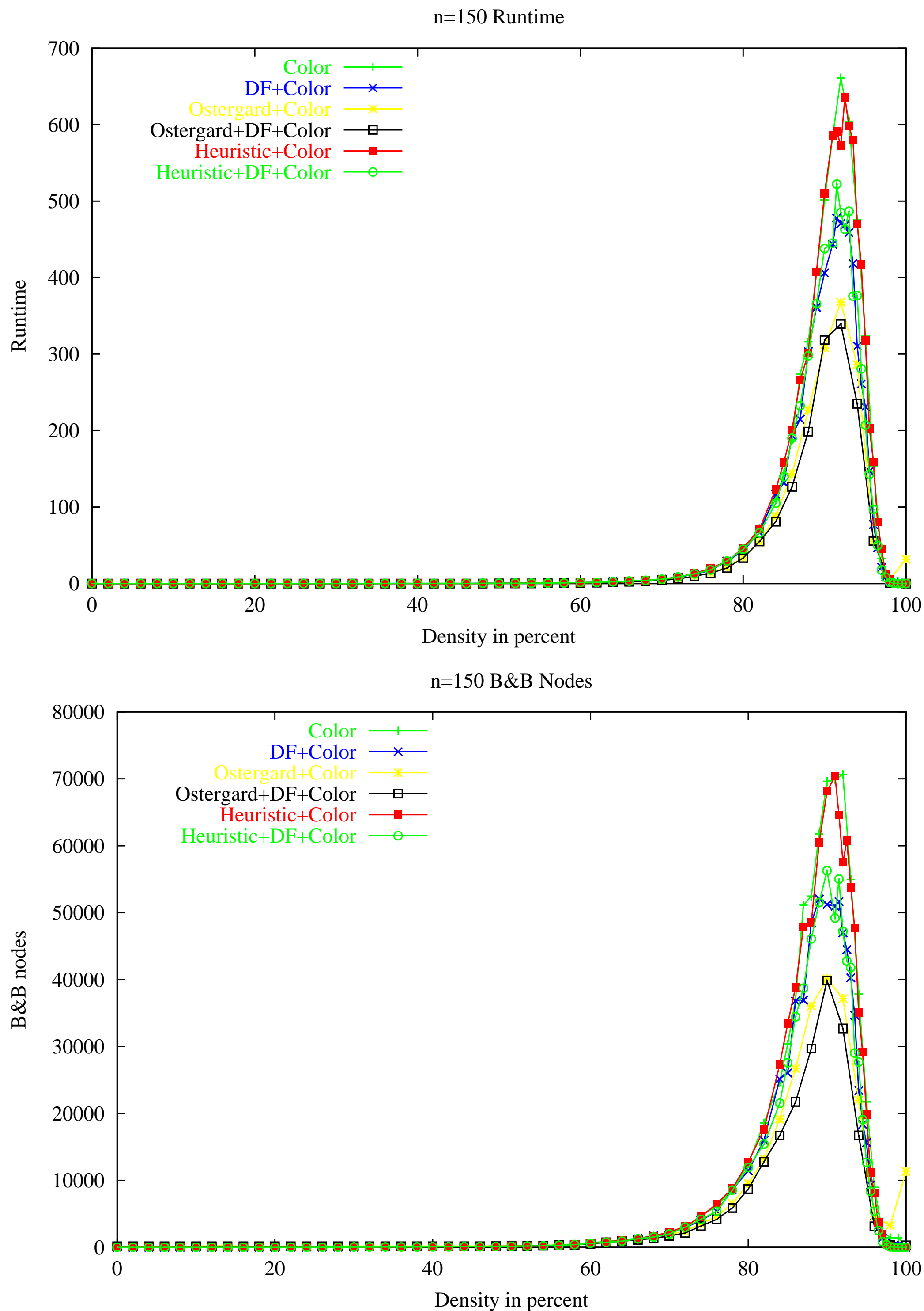


Figure 10.3: *Random graphs with 150 nodes. The y-axis gives runtime and number of branch-and-bound nodes, respectively. The y-axis is scaled linearly. On the x-axis, graph density varies from 0% – 100%. Each sample point represents an average value of 50 instances.*

in this thesis. The combination of a simple branch-and-bound, domain filtering, and lower and upper coloring bounds hence defines a simple and fast solution algorithm for MC.

Primal heuristics though seem to be the weakest part in the tests. It remains as an open question whether applying lower bound heuristics only in selected nodes of the branch-and-bound tree still preserves the positive impact while in general speeding up convergence. Also, we may improve the algorithms by using the stronger primal heuristic proposed by Balas and Xue [15].

Tests on random graphs indicate that domain filtering stabilizes the search for low degree graphs. A combination of domain filtering and coloring bounds is the best approach to high degree random graphs. As regards runtime, the more sophisticated techniques pay off only on high degree graphs. For random graphs with 100 or 150 nodes, respectively, a density of more than 85% is sufficient to justify the additional effort of filtering and bounding.

11

Conclusions and Open Questions

The integration of concepts from Constraint Programming and Operations Research algorithms has emerged in recent years. In this thesis we contributed some new techniques to the field. We developed integrated CP and OR techniques and applied them to

- the Airline Crew Rostering Problem,
- the Home Health Care Problem,
- the Automatic Recording Problem and
- the Maximum Clique Problem.

11.1 Conclusions

Solution approaches to the *Airline Crew Rostering Problem* benefit from CP based column generation. Complex airline rules and regulations can be treated efficiently by this framework. Also, we showed that the pure approach can be accelerated further by additional CP heuristics.

Carmen Systems, our industrial partner in PARROT, use some ideas from CP based column generation in their crew assignment software. They observed significant reductions in runtime when using improved IP heuristics, better constraint checking and the ideas from CP based column generation.

The concept of the CP based column generation framework was also used by Rousseau et al. [177]. They successfully applied it to the vehicle routing problem with time window constraints. Also their hybrid approach improved on a pure CP as well as on a pure column generation approach.

We conclude that CP based column generation is a very promising extension of the traditional column generation framework. Whenever column generation is appropriate for a problem, and when in addition the subproblem is described by complex constraints CP techniques

should be considered. When using the shortest path constraint or the knapsack constraint presented in previous chapters of this thesis quite a lot of subproblems can be modeled.

For the *Home Health Care Problem* we proposed a generic mathematical model. Based on this model we developed a combined CP and Tabu Search approach. CP provides feasible solutions quickly, whereas Tabu Search can easily improve them. In order to solve the sequencing subproblems we applied a combined CP and LP approach. CP searches for valid orderings of tasks and the LP assigns the optimal start time to each task in that ordering.

All in all the combined approach was better than Tabu Search or CP alone: Whereas Tabu Search had difficulties in finding an initial solution, a pure CP approach was hardly able to improve one. Thus, the mixture of techniques accelerated convergence in this real-world application.

For the *Automatic Recording Problem* we used CP based Lagrangian relaxation. When a given problem is composed of different substructures for which efficient domain filtering techniques are known, CP based Lagrangian relaxation allows us to reformulate the problem such that these filtering techniques can be efficiently applied. The concept was evaluated in Chapter 7. There, CP based Lagrangian relaxation was superior to pure Lagrangian relaxation and also to an approach that used the domain filtering without separating the substructures. The succeeding chapter, however, showed that the clear mathematical structure of the ARP can be tackled more efficiently by other OR approaches. With growing insight into the ARP's structure new approaches are likely to outdo the latter approaches as well.

Finally, we considered a prominent problem taken from computer science. We proposed domain filtering techniques for the *Maximum Clique Problem* and we were able to present a model in which domain filtering from CP and bounding techniques from OR can be compared. Using two different branch-and-bound methods from literature we were able to demonstrate the efficiency of domain filtering for maximum clique problem. Enhanced by additional techniques, a fast branch-and-bound algorithm was found and numerically investigated.

11.2 Open Questions

Goethe's quote on page iii is not only appropriate for politics. It is also valid in science. New insights into problems, new techniques or new numerical evaluations raise new questions. We mentioned some specific ones already in preceding chapters. Furthermore, we see two additional challenges:

- A deeper experimental foundation for the generic techniques presented, and
- a general theory for understanding the relation between CP domain filtering and OR bounding techniques.

CP based column generation as well as CP based Lagrangian relaxation provides some promising extensions of their classical roots. Using domain filtering while solving the resulting subproblem is likely to speed-up convergence. Using CP in the subproblems allows us to use a broad class of constraints. Finally, from a software engineering perspective, CP encapsulates logical constraints in simple building blocks, whereas IP approaches must model them via linear inequalities that are hard to maintain if requirements change.

Despite these advantages, both techniques need some more experimental evaluation from different application fields before declaring them superior or inferior to the standard approaches.

What is currently missing in the field of CP and OR integration is some generic description of effects resulting from the respective techniques. In polyhedral theory e.g. a notion of cut dominance is known and used to classify cuts. CP uses different levels of consistency to describe dominance of filtering techniques. However, there is — to our knowledge — no comprehensive theory for comparing domain filtering and bounds.

It might be helpful to interpret domain filtering for IPs as a kind of local cuts and then use methods from polyhedral theory for a classification. It is not clear, though, whether this is a successful path to follow. For the simple structure of the clique problem we were able to compare domain filtering and bounds and it is subject of further research whether these ideas can be applied to more complex settings.

Appendix

A

Numerical Results for the ARP

We give numerical results for all ARP benchmark classes tested. In section 7.3.1 we describe how these instances have been generated, and we also explain the different parameters for each class. Numbers given in boldface mark the approach that has been fastest on the corresponding instance. We use average runtime for 50 instances per set set for this comparison (respective numbers given in italic).

Whenever the runtime for a single instance took more than 200 hours (720 000 sec), the entire test set was not considered further. This only happened for tests on CP the based Lagrangian Relaxation. Therefore, results for

(5CU / 72h / 50ch) (5TSC / 24h / 50ch) (5TSC / 72h / 20ch) (5TSC / 72h / 50ch)
(5TWC / 72h / 50ch) (7CU / 72h / 50ch) (7TSC / 72h / 50ch) (7TWC / 72h / 50ch)

are missing in section A.1. Nevertheless, they appear in section A.2 where all results were found within this time limit. In fact, neither of the prevailing approaches (*DP*, DP^* and *BC*) was ever using more than 55 000 secs in our tests.

The fastest approach for a test class (measured over all seven approaches) is typed boldface. If numbers differ in less than $\frac{2}{200}$ sec. both numbers are given in boldface (see section 2.4.3 for a more detailed explanation).

A.1 CP based Lagrangian Relaxation

Instance		***LG-0***		***LG-1***		***LG-2***		***LG-3***	
		time	**ch pts**	**time**	**ch pts**	**time**	**ch pts**	**time**	**ch pts**
6h 5ch 3 CU	avg	*0.1*	*27.2*	*0.1*	*15.6*	***0.0***	*14.6*	***0.0***	*10.9*
	min	0.0	2.0	0.0	2.0	0.0	2.0	0.0	2.0
	max	0.3	136.0	0.2	70.0	0.1	55.0	0.1	32.0
	std	0.1	22.1	0.0	12.7	0.0	9.9	0.0	6.1

Instance		LG-0		LG-1		LG-2		LG-3	
		time	ch pts	time	ch pts	time	ch pts	time	ch pts
6h 20ch 3 CU	avg	*0.4*	*49.9*	*0.2*	*32.3*	*0.1*	*24.2*	*0.2*	*17.3*
	min	0.0	8.0	0.0	6.0	0.0	6.0	0.1	6.0
	max	3.8	409.0	1.1	127.0	0.6	172.0	0.7	66.0
	std	0.6	63.2	0.2	28.6	0.1	26.7	0.1	10.6
6h 50ch 3 CU	avg	*1.1*	*52.5*	*0.8*	*37.3*	*0.3*	*25.7*	*0.5*	*22.1*
	min	0.2	8.0	0.2	8.0	0.1	8.0	0.1	8.0
	max	5.4	267.0	3.8	208.0	1.6	92.0	1.9	89.0
	std	0.9	47.8	0.6	33.7	0.3	17.9	0.4	16.1
6h 100ch 3 CU	avg	*2.9*	*96.7*	*2.1*	*67.9*	*0.8*	*36.7*	*1.2*	*30.0*
	min	0.3	10.0	0.3	10.0	0.2	10.0	0.4	10.0
	max	10.5	726.0	7.9	210.0	2.9	131.0	5.0	119.0
	std	2.4	108.0	1.5	52.8	0.6	26.7	0.9	19.8
12h 5ch 3 CU	avg	*0.1*	*69.3*	*0.1*	*33.3*	*0.1*	*26.0*	*0.1*	*17.6*
	min	0.0	7.0	0.0	7.0	0.0	7.0	0.0	7.0
	max	1.7	982.0	0.3	128.0	0.2	99.0	0.2	42.0
	std	0.2	137.9	0.1	25.4	0.0	18.9	0.0	7.8
12h 20ch 3 CU	avg	*0.6*	*91.7*	*0.4*	*69.0*	*0.3*	*36.5*	*0.4*	*29.4*
	min	0.1	9.0	0.1	9.0	0.1	9.0	0.1	9.0
	max	3.2	517.0	2.1	403.0	0.9	121.0	1.0	68.0
	std	0.6	95.9	0.4	74.3	0.2	20.4	0.2	11.3
12h 50ch 3 CU	avg	*4.2*	*305.5*	*2.4*	*166.6*	*1.3*	*58.7*	*2.0*	*52.3*
	min	0.3	18.0	0.4	16.0	0.3	16.0	0.6	16.0
	max	47.8	4144.0	18.4	1503.0	6.2	234.0	6.4	182.0
	std	7.2	612.6	3.1	248.2	1.1	43.4	1.3	37.2
12h 100ch 3 CU	avg	*18.8*	*680.8*	*9.1*	*326.3*	*5.1*	*108.6*	*7.6*	*95.5*
	min	0.8	25.0	0.9	25.0	1.1	23.0	1.4	23.0
	max	109.4	4748.0	78.4	3158.0	30.5	512.0	34.6	437.0
	std	27.9	1128.4	13.1	529.6	5.7	112.9	7.9	97.4
24h 5ch 3 CU	avg	*0.8*	*130.4*	*0.7*	*98.2*	*0.2*	*48.0*	*0.4*	*38.5*
	min	0.1	21.0	0.1	20.0	0.1	17.0	0.1	17.0
	max	11.3	1211.0	9.9	1164.0	1.4	331.0	2.0	186.0
	std	1.6	182.3	1.4	165.3	0.2	46.0	0.3	25.7
24h 20ch 3 CU	avg	*7.9*	*479.6*	*5.3*	*306.1*	*1.6*	*103.0*	*2.5*	*81.9*
	min	0.8	32.0	0.8	28.0	0.5	34.0	0.9	32.0
	max	90.3	9362.0	47.9	4638.0	11.1	938.0	18.6	640.0
	std	13.8	1312.1	7.5	663.0	1.8	138.3	2.8	95.9
24h 50ch 3 CU	avg	*37.8*	*1461.1*	*19.5*	*734.4*	*11.1*	*187.9*	*13.8*	*178.5*
	min	1.1	34.0	1.4	34.0	1.5	34.0	2.7	34.0
	max	551.7	24461.0	281.2	11580.0	173.5	2411.0	177.8	2367.0
	std	85.1	3711.4	42.8	1768.6	24.5	344.1	25.6	336.9
24h 100ch 3 CU	avg	*291.0*	*5351.2*	*108.2*	*2028.1*	*55.6*	*530.2*	*71.0*	*439.9*
	min	3.1	50.0	4.0	50.0	4.2	43.0	6.8	43.0
	max	1798.5	34034.0	813.7	15199.0	626.3	8796.0	1020.4	6259.0
	std	466.0	8758.5	187.9	3606.2	102.0	1260.6	149.9	922.3
72h 5ch 3 CU	avg	*4.7*	*508.2*	*6.0*	*327.3*	*2.0*	*138.9*	*3.1*	*121.6*
	min	0.4	53.0	0.8	53.0	0.7	53.0	1.3	53.0
	max	23.6	2213.0	41.0	1678.0	5.4	592.0	14.1	563.0
	std	4.9	520.0	6.8	338.5	1.2	99.0	2.1	80.6
72h 20ch 3 CU	avg	*177.4*	*3003.1*	*111.7*	*1897.2*	*36.6*	*468.4*	*42.9*	*401.2*
	min	5.1	111.0	5.9	105.0	5.6	104.0	11.4	104.0
	max	1596.4	31675.0	1260.9	29942.0	543.3	5335.0	577.4	5335.0
	std	310.0	5677.8	200.2	4363.8	81.5	827.3	84.3	792.6

Instance		*LG-0*		*LG-1*		*LG-2*		*LG-3*	
		time	ch pts	time	ch pts	time	ch pts	time	ch pts
72h 50ch 3 CU	avg	*12227.8*	*149877.2*	*5647.5*	*67864.5*	*1191.5*	*4744.5*	*1197.3*	*4703.5*
	min	11.8	110.0	15.1	110.0	17.3	110.0	34.9	110.0
	max	142593.9	1750902.0	72763.9	869643.0	14733.4	53173.0	14733.8	53171.0
	std	24875.2	304985.5	12170.8	147372.8	2503.1	9484.5	2494.3	9480.3
120h 5ch 3 CU	avg	*64.0*	*3272.2*	*43.5*	*2705.4*	*17.5*	*878.0*	*17.9*	*434.5*
	min	1.6	114.0	2.0	114.0	2.5	114.0	4.5	113.0
	max	1305.4	52821.0	774.6	46513.0	380.2	23323.0	308.5	8464.0
	std	217.1	9746.8	144.2	8810.8	55.5	3344.4	44.5	1201.9
120h 20ch 3 CU	avg	*5210.3*	*60839.2*	*1734.4*	*30676.4*	*455.9*	*3433.9*	*490.1*	*2945.1*
	min	14.4	182.0	14.5	182.0	16.0	167.0	34.8	166.0
	max	120587.0	1163526.0	24954.8	465090.0	7518.4	46040.0	7314.1	46044.0
	std	17395.6	180210.1	4642.7	85554.5	1203.2	8133.8	1202.1	7594.8
12h 20ch 3 SC	avg	***0.1***	*19.7*	***0.1***	*11.9*	***0.1***	*12.0*	*0.2*	*11.9*
	min	0.1	5.0	0.0	5.0	0.0	5.0	0.1	5.0
	max	0.2	41.0	0.2	23.0	0.1	23.0	0.3	23.0
	std	0.0	8.7	0.0	3.9	0.0	3.9	0.0	3.9
12h 50ch 3 SC	avg	***0.3***	*17.1*	***0.3***	*10.6*	***0.2***	*10.6*	*0.4*	*10.6*
	min	0.1	5.0	0.1	5.0	0.1	5.0	0.2	5.0
	max	0.7	51.0	0.5	23.0	0.4	23.0	0.7	23.0
	std	0.1	11.0	0.1	4.3	0.1	4.3	0.1	4.3
24h 20ch 3 SC	avg	***0.3***	*22.0*	***0.3***	*16.2*	***0.3***	*16.2*	*0.5*	*16.2*
	min	0.1	9.0	0.2	9.0	0.1	9.0	0.2	9.0
	max	0.5	48.0	0.4	27.0	0.4	27.0	0.7	27.0
	std	0.1	8.4	0.1	3.9	0.1	3.9	0.1	3.9
24h 50ch 3 SC	avg	***0.7***	*19.4*	***0.8***	*15.7*	***0.7***	*15.7*	*1.2*	*15.7*
	min	0.3	7.0	0.4	7.0	0.3	7.0	0.6	7.0
	max	1.4	50.0	1.2	31.0	1.2	31.0	2.1	31.0
	std	0.2	8.0	0.2	4.6	0.2	4.6	0.3	4.6
72h 20ch 3 SC	avg	*1.6*	*39.1*	*1.9*	*35.5*	*2.0*	*35.5*	*3.5*	*35.5*
	min	1.1	25.0	1.3	25.0	1.4	25.0	2.7	25.0
	max	3.1	85.0	2.9	57.0	3.2	57.0	5.1	57.0
	std	0.4	12.1	0.3	6.7	0.4	6.7	0.6	6.7
72h 50ch 3 SC	avg	*4.8*	*39.5*	*5.9*	*36.0*	*6.1*	*36.0*	*10.7*	*36.0*
	min	2.8	23.0	3.8	22.0	3.6	22.0	6.7	22.0
	max	15.0	131.0	12.3	79.0	10.8	79.0	18.5	79.0
	std	1.7	14.9	1.2	8.0	1.1	8.0	2.0	8.0
12h 20ch 3 TSC	avg	*0.4*	*52.1*	*0.3*	*44.0*	*0.2*	*25.6*	*0.3*	*17.7*
	min	0.0	4.0	0.0	4.0	0.0	3.0	0.1	3.0
	max	1.6	225.0	1.4	183.0	0.9	131.0	1.6	67.0
	std	0.3	45.9	0.3	38.6	0.2	22.6	0.3	13.7
12h 50ch 3 TSC	avg	*1.7*	*92.7*	*1.5*	*81.9*	*0.9*	*45.0*	*1.3*	*30.5*
	min	0.1	3.0	0.1	3.0	0.1	3.0	0.1	3.0
	max	5.7	306.0	5.2	296.0	3.9	158.0	5.3	127.0
	std	1.3	70.7	1.1	65.6	0.7	35.4	1.1	26.9
24h 20ch 3 TSC	avg	*5.8*	*613.3*	*2.6*	*194.3*	*0.9*	*60.4*	*1.3*	*40.2*
	min	0.3	21.0	0.3	17.0	0.2	11.0	0.3	10.0
	max	142.6	17894.0	9.9	1111.0	4.9	228.0	5.1	154.0
	std	20.0	2514.6	2.5	214.2	0.9	51.5	1.2	34.5
24h 50ch 3 TSC	avg	*25.6*	*974.6*	*15.8*	*479.2*	*6.3*	*125.5*	*8.1*	*93.9*
	min	0.4	10.0	0.5	10.0	0.5	10.0	0.7	10.0
	max	193.3	10215.0	112.7	4594.0	42.9	788.0	46.8	582.0
	std	39.6	1896.1	21.1	760.7	7.9	138.6	9.8	115.1

Instance		LG-0		LG-1		LG-2		LG-3	
		time	ch pts	time	ch pts	time	ch pts	time	ch pts
72h 20ch 3 TSC	avg	*77.7*	*2334.1*	*56.8*	*1432.7*	*12.5*	*197.8*	*16.4*	*153.8*
	min	0.8	24.0	1.1	24.0	1.4	24.0	2.3	24.0
	max	567.0	19530.0	307.5	9434.0	56.7	740.0	58.8	617.0
	std	118.9	4147.6	60.6	1769.1	12.3	172.9	14.3	136.5
72h 50ch 3 TSC	avg	*3186.4*	*37460.3*	*1401.0*	*13926.7*	*211.1*	*949.2*	*248.8*	*815.7*
	min	46.1	359.0	46.9	346.0	12.0	64.0	18.4	54.0
	max	77889.6	968689.0	19507.8	232819.0	1667.3	9940.0	2286.0	8280.0
	std	11252.4	141270.4	2957.2	34516.9	337.7	1619.8	421.6	1451.7
12h 20ch 3 TWC	avg	*0.3*	*41.5*	*0.2*	*30.1*	*0.2*	*19.4*	*0.2*	*13.4*
	min	0.0	5.0	0.1	4.0	0.0	3.0	0.1	3.0
	max	2.1	377.0	0.9	125.0	0.5	52.0	0.7	34.0
	std	0.3	52.9	0.2	23.7	0.1	13.2	0.2	7.8
12h 50ch 3 TWC	avg	*0.7*	*37.3*	*0.6*	*30.3*	*0.4*	*20.2*	*0.6*	*15.2*
	min	0.1	4.0	0.1	4.0	0.1	4.0	0.1	4.0
	max	2.2	154.0	2.1	106.0	2.0	121.0	4.4	91.0
	std	0.6	33.4	0.5	26.0	0.3	19.6	0.7	13.9
24h 20ch 3 TWC	avg	*1.0*	*77.7*	*0.9*	*64.8*	*0.5*	*30.2*	*0.7*	*22.3*
	min	0.1	9.0	0.1	8.0	0.1	8.0	0.2	7.0
	max	3.9	275.0	4.3	271.0	2.3	91.0	2.5	81.0
	std	0.8	60.9	0.7	52.8	0.4	18.8	0.5	13.4
24h 50ch 3 TWC	avg	*6.0*	*190.8*	*4.5*	*128.9*	*2.1*	*50.6*	*3.1*	*36.0*
	min	0.4	10.0	0.5	9.0	0.4	9.0	0.7	9.0
	max	30.0	1198.0	14.4	527.0	5.8	146.0	8.8	104.0
	std	6.8	265.7	3.5	117.0	1.3	33.3	2.0	22.7
72h 20ch 3 TWC	avg	*102.4*	*3707.0*	*32.9*	*1014.0*	*8.2*	*147.6*	*11.9*	*113.4*
	min	0.9	23.0	1.2	23.0	1.4	23.0	2.4	22.0
	max	3167.4	122104.0	306.4	12033.0	54.6	1711.0	111.5	1231.0
	std	442.1	17081.6	58.0	2322.7	8.9	243.1	16.1	172.2
72h 50ch 3 TWC	avg	*274.1*	*3206.3*	*116.9*	*1094.9*	*22.0*	*150.6*	*32.3*	*110.8*
	min	5.8	50.0	6.3	49.0	4.1	25.0	7.2	25.0
	max	5825.0	76170.0	714.6	8129.0	92.1	1031.0	179.4	730.0
	std	828.6	10816.2	151.8	1699.4	19.1	157.3	28.6	111.1
12h 20ch 5 CU	avg	*2.4*	*238.3*	*1.3*	*129.9*	*1.4*	*108.6*	*2.1*	*89.9*
	min	0.1	5.0	0.1	5.0	0.1	5.0	0.1	5.0
	max	25.4	2216.0	15.1	1531.0	12.9	1269.0	24.2	1045.0
	std	4.2	396.8	2.3	243.6	2.5	206.5	4.0	173.5
12h 50ch 5 CU	avg	*16.5*	*741.9*	*8.2*	*370.1*	*9.9*	*272.0*	*14.2*	*250.5*
	min	0.1	3.0	0.2	3.0	0.2	3.0	0.2	3.0
	max	167.3	7058.0	82.6	3615.0	154.1	2664.0	156.5	2664.0
	std	30.8	1377.6	14.5	658.9	22.9	506.9	26.8	465.0
24h 20ch 5 CU	avg	*9.5*	*519.4*	*5.2*	*295.5*	*7.0*	*198.5*	*8.6*	*184.0*
	min	0.5	21.0	0.4	13.0	0.3	10.0	0.5	10.0
	max	87.5	4416.0	59.6	3762.0	93.4	2094.0	98.1	2067.0
	std	15.2	829.9	9.3	587.6	17.2	377.8	17.7	374.9
24h 50ch 5 CU	avg	*1104.9*	*24301.4*	*585.2*	*14219.3*	*883.3*	*8371.9*	*921.5*	*8286.8*
	min	0.8	12.0	1.0	12.0	0.7	9.0	1.1	9.0
	max	31045.5	675235.0	15625.2	368440.0	33281.3	292753.0	31573.5	292753.0
	std	4448.4	97121.0	2272.6	54288.6	4662.0	41139.4	4441.2	41121.2
72h 20ch 5 CU	avg	*2627.7*	*40901.5*	*1786.7*	*27662.0*	*920.4*	*6674.7*	*990.9*	*6514.7*
	min	2.0	29.0	2.4	29.0	3.0	29.0	5.5	29.0
	max	32751.9	460350.0	30520.3	412421.0	11766.0	90397.0	13724.7	89589.0
	std	5514.7	85325.8	4543.1	65188.4	1996.8	14515.9	2189.7	14379.4

Instance		*LG-0*		*LG-1*		*LG-2*		*LG-3*	
		time	ch pts	time	ch pts	time	ch pts	time	ch pts
12h 20ch 5 SC	avg	*0.2*	*23.1*	*0.2*	*15.2*	*0.2*	*15.2*	*0.3*	*15.2*
	min	0.1	4.0	0.1	4.0	0.1	4.0	0.1	4.0
	max	0.7	88.0	0.5	47.0	0.5	48.0	1.0	47.0
	std	0.1	17.0	0.1	7.5	0.1	7.6	0.1	7.5
12h 50ch 5 SC	avg	*0.5*	*18.8*	*0.5*	*13.9*	*0.5*	*13.9*	*0.8*	*13.9*
	min	0.1	4.0	0.2	4.0	0.2	4.0	0.3	4.0
	max	1.3	55.0	0.9	28.0	0.9	28.0	1.6	28.0
	std	0.2	10.0	0.2	4.6	0.1	4.6	0.3	4.6
24h 20ch 5 SC	avg	*0.5*	*26.9*	*0.6*	*21.9*	*0.6*	*21.9*	*1.0*	*21.9*
	min	0.3	10.0	0.3	10.0	0.3	10.0	0.6	10.0
	max	1.4	69.0	1.2	50.0	1.1	50.0	1.9	50.0
	std	0.2	11.6	0.2	7.0	0.2	7.0	0.3	7.0
24h 50ch 5 SC	avg	***1.6***	*29.1*	*1.8*	*23.2*	*1.7*	*23.2*	*3.0*	*23.2*
	min	0.6	9.0	0.8	9.0	0.8	9.0	1.3	9.0
	max	4.5	90.0	3.9	55.0	3.3	55.0	5.9	55.0
	std	0.9	18.8	0.7	10.4	0.6	10.4	1.2	10.4
72h 20ch 5 SC	avg	*3.5*	*53.4*	*4.6*	*51.7*	*5.0*	*51.7*	*8.7*	*51.7*
	min	2.4	35.0	2.9	35.0	3.2	35.0	5.7	35.0
	max	6.0	99.0	6.3	72.0	7.0	72.0	13.2	72.0
	std	0.8	12.0	0.9	9.3	0.9	9.3	1.7	9.3
72h 50ch 5 SC	avg	*11.0*	*54.4*	*14.4*	*52.8*	*15.4*	*52.8*	*27.2*	*52.8*
	min	6.2	32.0	8.8	32.0	9.4	32.0	16.4	32.0
	max	21.9	107.0	23.6	89.0	24.0	89.0	41.0	89.0
	std	3.1	15.6	3.1	12.0	3.4	12.0	5.7	12.0
12h 20ch 5 TSC	avg	*10.9*	*1573.0*	*4.7*	*597.4*	*1.6*	*144.2*	*3.3*	*125.3*
	min	0.1	9.0	0.1	8.0	0.1	8.0	0.2	8.0
	max	415.2	62816.0	127.5	18607.0	26.4	3236.0	78.9	3031.0
	std	57.9	8767.4	17.7	2588.8	4.0	454.4	11.1	426.3
12h 50ch 5 TSC	avg	*5667.7*	*346435.2*	*5119.1*	*306369.3*	*332.2*	*6292.3*	*372.3*	*5839.0*
	min	0.3	12.0	0.4	12.0	0.4	12.0	0.6	9.0
	max	157597.0	9718051.0	135791.1	8752271.0	7229.1	114820.0	7209.1	114747.0
	std	23764.0	1463794.5	21251.3	1318189.3	1051.8	17169.1	1070.1	17081.8
24h 20ch 5 TSC	avg	*441.8*	*40155.7*	*67.2*	*4525.8*	*18.0*	*696.5*	*28.4*	*575.6*
	min	0.3	18.0	0.4	18.0	0.4	18.0	0.7	18.0
	max	19371.5	1829155.0	1255.8	98915.0	226.7	9207.0	364.8	7888.0
	std	2706.0	255669.8	184.6	14343.6	43.3	1789.2	75.7	1555.7
12h 20ch 5 TWC	avg	*1.4*	*167.1*	*0.9*	*95.0*	*0.4*	*37.2*	*0.7*	*28.7*
	min	0.1	8.0	0.1	8.0	0.1	6.0	0.1	6.0
	max	11.1	1615.0	5.2	543.0	2.5	279.0	6.5	243.0
	std	1.9	265.6	1.0	109.5	0.4	42.5	0.9	35.7
12h 50ch 5 TWC	avg	*130.5*	*8749.1*	*62.1*	*3966.5*	*9.1*	*217.2*	*11.3*	*187.3*
	min	0.2	9.0	0.3	9.0	0.3	9.0	0.5	9.0
	max	2884.7	198292.0	1225.0	83713.0	211.8	3579.0	221.3	3579.0
	std	466.4	32541.8	197.8	13705.2	29.6	531.7	32.2	529.2
24h 20ch 5 TWC	avg	*15.5*	*1136.1*	*12.0*	*802.6*	*6.2*	*339.9*	*9.5*	*321.2*
	min	0.2	14.0	0.3	14.0	0.3	14.0	0.6	14.0
	max	201.8	18359.0	142.0	12113.0	185.0	12579.0	301.8	12376.0
	std	39.4	3236.1	29.8	2246.5	25.8	1751.9	42.0	1724.8
24h 50ch 5 TWC	avg	*790.0*	*23226.4*	*283.0*	*7796.7*	*62.9*	*788.8*	*79.1*	*687.9*
	min	2.2	46.0	2.6	46.0	1.3	20.0	2.1	17.0
	max	9770.2	272945.0	4505.2	126052.0	981.6	15246.0	1412.5	12997.0
	std	1964.3	58445.8	703.3	20049.1	165.7	2320.4	219.5	2006.4

Instance		*LG-0*		*LG-1*		*LG-2*		*LG-3*	
		time	ch pts	time	ch pts	time	ch pts	time	ch pts
72h 20ch 5 TWC	avg	*11600.1*	*293386.8*	*1526.8*	*35718.4*	*261.0*	*3683.6*	*411.7*	*3134.5*
	min	2.5	45.0	3.4	45.0	3.8	43.0	6.8	43.0
	max	231048.7	5883904.0	19584.2	537840.0	2977.7	55820.0	5509.4	45286.0
	std	38598.8	991575.0	3349.5	87056.2	617.1	10463.8	1078.7	8618.2
12h 20ch 7 CU	avg	*1.3*	*134.3*	*1.1*	*119.8*	*0.6*	*48.7*	*1.0*	*44.2*
	min	0.3	32.0	0.3	32.0	0.4	32.0	0.6	32.0
	max	5.8	546.0	4.6	537.0	1.3	73.0	1.9	58.0
	std	1.1	101.0	0.8	95.9	0.2	8.5	0.3	5.5
12h 50ch 7 CU	avg	*7.9*	*305.6*	*6.6*	*278.5*	*2.1*	*68.3*	*3.7*	*62.3*
	min	1.3	53.0	1.3	53.0	1.5	53.0	2.8	50.0
	max	63.4	2283.0	55.2	2272.0	5.3	111.0	6.5	97.0
	std	10.7	397.5	9.5	390.6	0.8	13.8	0.7	9.0
24h 20ch 7 CU	avg	*8.0*	*449.0*	*6.7*	*375.6*	*2.4*	*97.6*	*4.5*	*88.8*
	min	1.1	70.0	1.4	70.0	1.6	70.0	2.9	69.0
	max	69.8	4570.0	40.4	2198.0	4.5	177.0	6.7	125.0
	std	11.1	684.0	7.5	429.5	0.7	23.8	0.9	12.6
24h 50ch 7 CU	avg	*67.0*	*1451.9*	*46.6*	*1034.3*	*10.8*	*162.4*	*19.3*	*141.7*
	min	5.5	109.0	5.8	109.0	6.5	109.0	14.0	106.0
	max	771.2	18584.0	313.5	7290.0	35.1	331.0	47.0	281.0
	std	115.5	2731.6	61.8	1374.4	5.2	56.2	6.6	39.9
72h 20ch 7 CU	avg	*689.3*	*13135.8*	*490.2*	*9272.8*	*60.4*	*426.1*	*81.1*	*387.5*
	min	12.1	234.0	15.6	234.0	19.4	228.0	43.5	222.0
	max	18704.6	358404.0	12983.2	243556.0	655.5	3093.0	646.1	3018.0
	std	2670.4	51441.3	1868.2	35342.8	98.4	476.6	95.8	443.3
12h 20ch 7 SC	avg	*0.2*	*18.4*	*0.2*	*12.7*	***0.1***	*12.7*	*0.3*	*12.7*
	min	0.1	5.0	0.1	5.0	0.1	5.0	0.1	5.0
	max	0.6	76.0	0.4	39.0	0.4	40.0	0.6	39.0
	std	0.1	13.6	0.1	5.9	0.1	6.1	0.1	5.9
12h 50ch 7 SC	avg	*0.5*	*18.7*	*0.5*	*13.1*	***0.4***	*13.1*	*0.7*	*13.1*
	min	0.2	8.0	0.2	6.0	0.2	6.0	0.4	6.0
	max	0.9	46.0	0.8	26.0	0.8	26.0	1.4	26.0
	std	0.2	7.9	0.1	4.0	0.1	4.0	0.2	4.0
24h 20ch 7 SC	avg	*0.5*	*26.3*	*0.5*	*21.2*	*0.5*	*21.2*	*0.9*	*21.2*
	min	0.2	11.0	0.2	7.0	0.2	7.0	0.3	7.0
	max	1.5	93.0	1.1	53.0	1.0	53.0	1.7	53.0
	std	0.2	15.2	0.2	8.2	0.2	8.2	0.3	8.2
24h 50ch 7 SC	avg	***1.2***	*22.0*	*1.3*	*18.5*	*1.3*	*18.5*	*2.3*	*18.5*
	min	0.7	12.0	0.8	11.0	0.8	11.0	1.3	11.0
	max	2.8	59.0	2.9	43.0	2.7	43.0	4.3	43.0
	std	0.4	9.2	0.4	5.4	0.3	5.4	0.6	5.4
72h 20ch 7 SC	avg	*3.2*	*52.3*	*4.1*	*50.1*	*4.4*	*50.1*	*7.6*	*50.1*
	min	2.3	39.0	2.8	37.0	2.8	37.0	4.7	37.0
	max	4.7	75.0	5.2	65.0	5.6	65.0	10.1	65.0
	std	0.6	8.9	0.6	6.6	0.7	6.6	1.2	6.6
72h 50ch 7 SC	avg	*9.8*	*52.8*	*12.5*	*50.0*	*13.6*	*50.0*	*23.9*	*50.0*
	min	6.6	35.0	9.6	35.0	9.5	35.0	17.4	35.0
	max	16.6	93.0	18.4	69.0	18.7	69.0	35.0	69.0
	std	2.0	11.7	1.7	7.1	1.8	7.1	3.4	7.1
12h 20ch 7 TSC	avg	*2.7*	*377.2*	*1.6*	*183.6*	*0.6*	*47.7*	*0.8*	*32.8*
	min	0.1	7.0	0.1	7.0	0.1	7.0	0.1	7.0
	max	38.2	6599.0	12.3	1609.0	3.3	167.0	3.4	158.0
	std	6.1	1015.6	2.1	280.9	0.6	37.1	0.7	28.3

Instance		*LG-0*		*LG-1*		*LG-2*		*LG-3*	
		time	ch pts	time	ch pts	time	ch pts	time	ch pts
12h 50ch 7 TSC	avg	*235.8*	*15278.3*	*112.1*	*6169.5*	*28.5*	*655.5*	*32.6*	*496.0*
	min	0.3	14.0	0.4	14.0	0.3	10.0	0.5	8.0
	max	4529.1	342700.0	1218.6	76353.0	538.3	8619.0	540.5	8614.0
	std	688.1	50867.3	227.6	13576.3	80.0	1486.5	83.6	1309.1
24h 20ch 7 TSC	avg	*29.6*	*2335.5*	*22.7*	*1596.4*	*3.7*	*132.3*	*4.9*	*105.2*
	min	0.3	19.0	0.4	19.0	0.4	19.0	0.8	17.0
	max	470.5	42780.0	393.8	30321.0	36.2	854.0	36.6	847.0
	std	80.4	7243.9	64.1	5129.7	6.2	179.0	6.7	150.0
24h 50ch 7 TSC	avg	*8349.0*	*250367.1*	*4066.3*	*105572.6*	*403.4*	*6235.4*	*533.0*	*4219.1*
	min	0.9	20.0	1.2	20.0	1.2	18.0	2.2	18.0
	max	156559.0	5194747.0	44659.5	1196233.0	8794.1	198714.0	14067.3	104481.0
	std	24549.2	786477.6	9903.9	247724.9	1290.3	27833.6	1990.2	14959.6
72h 20ch 7 TSC	avg	*4229.7*	*96482.8*	*3430.5*	*74021.0*	*297.8*	*2121.0*	*327.9*	*2062.8*
	min	2.9	54.0	3.7	54.0	4.6	54.0	8.3	54.0
	max	41811.7	1027702.0	28476.6	671850.0	4476.5	28015.0	4889.7	28003.0
	std	8760.7	208251.1	7012.0	161117.5	708.7	4603.9	768.2	4605.1
12h 20ch 7 TWC	avg	*0.7*	*77.7*	*0.7*	*68.8*	*0.3*	*27.0*	*0.5*	*20.6*
	min	0.1	6.0	0.1	6.0	0.1	6.0	0.1	6.0
	max	5.0	621.0	4.8	557.0	2.1	131.0	2.4	80.0
	std	1.0	108.8	0.9	101.4	0.3	22.9	0.5	16.0
12h 50ch 7 TWC	avg	*20.9*	*1338.9*	*9.0*	*447.0*	*2.7*	*84.0*	*4.0*	*66.3*
	min	0.3	10.0	0.3	10.0	0.3	10.0	0.5	10.0
	max	467.3	35241.0	82.3	5948.0	21.4	797.0	36.4	650.0
	std	68.5	5143.9	15.4	976.1	4.1	128.1	6.4	107.6
24h 20ch 7 TWC	avg	*26.9*	*2371.1*	*8.2*	*561.7*	*2.0*	*87.8*	*3.3*	*71.8*
	min	0.2	11.0	0.2	11.0	0.2	11.0	0.5	11.0
	max	1022.3	96320.0	196.6	15804.0	13.8	1012.0	39.3	966.0
	std	142.5	13440.0	27.3	2194.8	2.5	147.9	5.7	137.0
24h 50ch 7 TWC	avg	*172.3*	*5066.3*	*117.1*	*3086.5*	*17.1*	*193.6*	*21.2*	*168.9*
	min	0.8	18.0	1.1	18.0	1.1	18.0	2.1	18.0
	max	1532.9	50612.0	1092.9	34787.0	127.6	840.0	120.1	804.0
	std	338.4	10792.1	235.8	6788.0	21.4	182.7	21.9	169.9
72h 20ch 7 TWC	avg	*569.2*	*13333.6*	*384.8*	*8518.1*	*55.4*	*430.8*	*58.2*	*417.1*
	min	1.8	40.0	2.6	40.0	3.1	40.0	5.9	40.0
	max	13906.8	323678.0	9552.0	218748.0	763.9	5057.0	650.1	5052.0
	std	1979.2	46478.2	1361.7	31389.7	123.4	851.1	111.6	852.4

A.2 Dynamic Programming and Branch-and-Cut

Instance		*DP* time	*DP** time	*BC* time	*BC* BBnodes	*BC* 1st solution time	*BC* 1st solution gap
6h 5ch 3 CU	avg	***0.00***	***0.00***	***0.02***	*3.40*	*0.01*	*0.43*
	min	0.00	0.00	0.00	0.00	0.00	0.00
	max	0.01	0.01	0.04	24.00	0.03	3.40
	std	0.00	0.00	0.01	5.38	0.01	0.88
6h 20ch 3 CU	avg	*0.04*	***0.01***	*0.05*	*5.84*	*0.04*	*0.50*
	min	0.02	0.00	0.02	0.00	0.02	0.00
	max	0.06	0.01	0.12	61.00	0.07	3.24
	std	0.01	0.00	0.03	10.38	0.01	0.84
6h 50ch 3 CU	avg	*0.27*	***0.03***	*0.17*	*8.26*	*0.13*	*0.42*
	min	0.18	0.02	0.06	0.00	0.06	0.00
	max	0.43	0.05	0.41	84.00	0.19	2.65
	std	0.05	0.01	0.07	16.37	0.03	0.67
6h 100ch 3 CU	avg	*1.26*	***0.12***	*0.53*	*15.38*	*0.39*	*0.68*
	min	0.88	0.08	0.22	0.00	0.21	0.00
	max	1.82	0.16	2.12	184.00	0.56	3.47
	std	0.19	0.02	0.28	35.75	0.07	0.86
12h 5ch 3 CU	avg	***0.02***	***0.01***	***0.02***	*6.28*	*0.02*	*0.16*
	min	0.01	0.00	0.00	0.00	0.00	0.00
	max	0.04	0.01	0.07	76.00	0.03	1.76
	std	0.01	0.00	0.02	15.01	0.01	0.36
12h 20ch 3 CU	avg	*0.44*	***0.05***	*0.11*	*12.82*	*0.08*	*0.25*
	min	0.30	0.03	0.04	0.00	0.04	0.00
	max	0.64	0.06	0.39	236.00	0.12	1.30
	std	0.07	0.01	0.07	34.94	0.02	0.37
12h 50ch 3 CU	avg	*3.37*	***0.26***	*0.47*	*16.42*	*0.32*	*0.25*
	min	2.59	0.21	0.19	0.00	0.18	0.00
	max	4.23	0.34	1.25	83.00	0.42	1.25
	std	0.35	0.03	0.23	22.51	0.06	0.32
12h 100ch 3 CU	avg	*14.63*	***0.94***	*1.40*	*17.38*	*1.05*	*0.35*
	min	12.79	0.78	0.70	0.00	0.70	0.00
	max	16.91	1.09	3.22	140.00	1.33	1.16
	std	1.23	0.08	0.55	30.64	0.14	0.38
24h 5ch 3 CU	avg	*0.24*	***0.03***	*0.06*	*14.80*	*0.02*	*0.22*
	min	0.16	0.02	0.00	0.00	0.01	0.00
	max	0.35	0.05	0.19	100.00	0.05	1.35
	std	0.04	0.01	0.04	24.83	0.01	0.31
24h 20ch 3 CU	avg	*4.43*	*0.30*	***0.24***	*12.92*	*0.16*	*0.12*
	min	3.71	0.24	0.07	0.00	0.09	0.00
	max	5.71	0.37	0.62	140.00	0.21	0.80
	std	0.44	0.03	0.12	24.26	0.03	0.15

Instance		*DP* time	*DP** time	*BC* time	*BC* BBnodes	*BC* 1st solution time	*BC* 1st solution gap
24h 50ch 3 CU	avg	*30.32*	*1.47*	***1.03***	*18.86*	*0.68*	*0.12*
	min	25.71	1.24	0.45	0.00	0.44	0.00
	max	36.25	2.05	2.87	146.00	0.91	0.56
	std	2.60	0.15	0.62	35.87	0.13	0.16
24h 100ch 3 CU	avg	*134.49*	*5.23*	***2.81***	*13.74*	*2.16*	*0.14*
	min	117.56	4.53	1.48	0.00	1.51	0.00
	max	152.79	5.90	10.20	228.00	2.72	0.64
	std	8.50	0.37	1.47	34.41	0.30	0.20
72h 5ch 3 CU	avg	*7.18*	*0.45*	***0.17***	*22.68*	*0.07*	*0.11*
	min	5.46	0.36	0.03	0.00	0.03	0.00
	max	8.68	0.54	0.46	113.00	0.13	0.49
	std	0.82	0.04	0.10	26.31	0.02	0.11
72h 20ch 3 CU	avg	*127.99*	*4.29*	***1.04***	*24.58*	*0.57*	*0.06*
	min	105.52	3.64	0.33	0.00	0.34	0.00
	max	148.24	4.98	3.12	260.00	0.73	0.25
	std	10.47	0.33	0.64	48.70	0.09	0.06
72h 50ch 3 CU	avg	*925.62*	*20.05*	***3.74***	*22.52*	*2.36*	*0.05*
	min	807.64	17.76	1.63	0.00	1.65	0.00
	max	1125.99	22.85	19.51	392.00	3.01	0.30
	std	67.74	1.30	3.37	66.10	0.42	0.07
120h 5ch 3 CU	avg	*34.36*	*1.59*	***0.28***	*22.92*	*0.13*	*0.05*
	min	29.02	1.37	0.05	0.00	0.05	0.00
	max	44.09	1.86	1.22	170.00	0.17	0.59
	std	3.32	0.12	0.23	37.33	0.03	0.09
120h 20ch 3 CU	avg	*630.21*	*14.21*	***1.92***	*18.28*	*1.02*	*0.04*
	min	521.38	12.23	0.65	0.00	0.64	0.00
	max	723.36	15.83	7.56	180.00	1.28	0.12
	std	45.47	0.90	1.36	32.13	0.15	0.03
12h 20ch 3 SC	avg	*0.44*	***0.05***	*0.12*	*12.92*	*0.07*	*0.50*
	min	0.32	0.03	0.05	0.00	0.03	0.00
	max	0.62	0.06	0.25	64.00	0.09	1.83
	std	0.08	0.01	0.05	13.84	0.01	0.54
12h 50ch 3 SC	avg	*3.42*	***0.26***	*0.39*	*8.20*	*0.27*	*0.49*
	min	2.54	0.20	0.18	0.00	0.16	0.00
	max	4.69	0.35	0.81	43.00	0.33	7.03
	std	0.44	0.03	0.18	11.73	0.04	1.04
24h 20ch 3 SC	avg	*4.53*	*0.31*	***0.28***	*13.78*	*0.16*	*0.25*
	min	3.40	0.25	0.11	0.00	0.11	0.00
	max	5.42	0.36	0.73	67.00	0.21	0.82
	std	0.49	0.03	0.15	17.54	0.02	0.25
24h 50ch 3 SC	avg	*30.10*	*1.47*	*1.02*	*10.46*	*0.61*	*0.16*
	min	24.15	1.19	0.38	0.00	0.38	0.00
	max	36.17	1.77	4.15	82.00	0.72	0.67
	std	2.52	0.12	0.71	16.58	0.08	0.21

Instance		*DP*	*DP**	*BC*		*BC* 1st solution	
		time	time	time	BBnodes	time	gap
72h 20ch 3 SC	avg	*127.02*	*4.28*	***0.95***	*9.28*	*0.61*	*0.07*
	min	103.21	3.57	0.35	0.00	0.37	0.00
	max	150.58	4.95	3.17	90.00	0.90	0.30
	std	9.28	0.28	0.61	18.51	0.09	0.10
72h 50ch 3 SC	avg	*916.12*	*20.00*	***3.70***	*8.42*	*2.29*	*0.04*
	min	771.82	17.60	1.53	0.00	1.53	0.00
	max	1091.52	23.13	11.67	65.00	3.09	0.20
	std	70.08	1.46	2.77	16.07	0.44	0.06
12h 20ch 3 TSC	avg	*0.44*	***0.05***	*0.09*	*10.18*	*0.06*	*0.98*
	min	0.32	0.03	0.04	0.00	0.04	0.00
	max	0.62	0.06	0.27	60.00	0.09	6.77
	std	0.08	0.01	0.05	17.09	0.01	1.71
12h 50ch 3 TSC	avg	*3.42*	***0.26***	*0.46*	*14.26*	*0.34*	*0.96*
	min	2.54	0.20	0.25	0.00	0.25	0.00
	max	4.67	0.35	1.13	101.00	0.47	5.08
	std	0.44	0.03	0.19	22.95	0.04	1.37
24h 20ch 3 TSC	avg	*4.55*	*0.31*	***0.24***	*24.70*	*0.14*	*0.95*
	min	3.41	0.25	0.10	0.00	0.10	0.00
	max	5.60	0.36	1.02	224.00	0.17	3.77
	std	0.51	0.03	0.14	36.06	0.01	0.95
24h 50ch 3 TSC	avg	*30.13*	*1.47*	***1.15***	*23.44*	*0.77*	*0.73*
	min	24.48	1.20	0.54	0.00	0.50	0.00
	max	36.40	1.78	2.77	135.00	0.94	3.38
	std	2.61	0.12	0.50	33.26	0.08	0.90
72h 20ch 3 TSC	avg	*127.12*	*4.29*	***0.90***	*33.50*	*0.46*	*0.24*
	min	103.65	3.62	0.32	0.00	0.32	0.00
	max	150.05	4.97	3.95	300.00	0.55	0.89
	std	9.23	0.29	0.65	57.21	0.05	0.26
72h 50ch 3 TSC	avg	*918.16*	*20.07*	***4.86***	*35.86*	*2.76*	*0.32*
	min	770.61	17.67	2.02	0.00	2.00	0.00
	max	1075.28	23.20	13.11	277.00	3.18	1.17
	std	72.07	1.44	2.66	57.24	0.20	0.30
12h 20ch 3 TWC	avg	*0.45*	***0.05***	*0.09*	*9.94*	*0.07*	*1.33*
	min	0.33	0.04	0.04	0.00	0.04	0.00
	max	0.57	0.06	0.22	58.00	0.09	8.03
	std	0.07	0.01	0.04	12.92	0.01	1.87
12h 50ch 3 TWC	avg	*3.39*	***0.26***	*0.40*	*5.38*	*0.34*	*0.61*
	min	2.50	0.20	0.25	0.00	0.24	0.00
	max	4.47	0.33	0.80	50.00	0.55	4.85
	std	0.39	0.03	0.12	10.15	0.05	1.20
24h 20ch 3 TWC	avg	*4.46*	*0.30*	***0.24***	*24.60*	*0.13*	*0.76*
	min	3.64	0.25	0.09	0.00	0.09	0.00
	max	5.64	0.35	1.31	212.00	0.18	3.07
	std	0.45	0.02	0.19	38.36	0.02	0.87

Instance		*DP* time	*DP** time	*BC* time	*BC* BBnodes	*BC* 1st solution time	*BC* 1st solution gap
24h 50ch 3 TWC	avg	*30.91*	*1.50*	***1.11***	*20.00*	*0.78*	*0.71*
	min	26.13	1.26	0.58	0.00	0.56	0.00
	max	37.75	1.73	3.39	130.00	0.95	2.73
	std	2.47	0.11	0.46	27.61	0.08	0.78
72h 20ch 3 TWC	avg	*126.74*	*4.30*	***1.06***	*52.86*	*0.46*	*0.33*
	min	105.23	3.73	0.33	0.00	0.29	0.00
	max	147.63	5.00	5.00	330.00	0.55	1.14
	std	9.54	0.32	0.74	65.17	0.04	0.32
72h 50ch 3 TWC	avg	*916.37*	*20.08*	***3.92***	*19.72*	*2.70*	*0.26*
	min	806.85	17.68	2.19	0.00	2.24	0.00
	max	1066.29	22.46	13.31	151.00	2.96	1.08
	std	66.17	1.29	2.05	37.20	0.18	0.30
12h 20ch 5 CU	avg	*1.16*	***0.08***	*0.21*	*37.82*	*0.11*	*0.12*
	min	0.69	0.05	0.06	0.00	0.06	0.00
	max	1.67	0.11	1.59	484.00	0.19	0.96
	std	0.20	0.01	0.28	99.55	0.02	0.20
12h 50ch 5 CU	avg	*8.44*	***0.44***	*1.08*	*103.98*	*0.51*	*0.12*
	min	6.34	0.35	0.36	0.00	0.36	0.00
	max	11.09	0.58	9.11	1312.00	0.68	0.78
	std	1.13	0.05	1.41	240.16	0.07	0.20
24h 20ch 5 CU	avg	*10.28*	***0.47***	*1.38*	*234.96*	*0.23*	*0.09*
	min	8.06	0.39	0.18	0.00	0.17	0.00
	max	13.48	0.57	43.64	7618.00	0.30	0.74
	std	1.08	0.04	6.06	1074.40	0.03	0.14
24h 50ch 5 CU	avg	*70.46*	***2.40***	*176.54*	*8851.52*	*1.15*	*0.16*
	min	56.41	1.94	0.86	0.00	0.88	0.00
	max	87.90	3.11	4357.75	235195.00	1.53	2.24
	std	7.36	0.26	715.71	37095.52	0.13	0.34
72h 20ch 5 CU	avg	*295.83*	***6.60***	*516.96*	*52576.38*	*0.82*	*0.05*
	min	256.36	5.69	0.54	0.00	0.55	0.00
	max	360.12	7.86	7650.35	1498720.00	1.00	0.24
	std	24.21	0.47	1658.57	223318.36	0.09	0.06
72h 50ch 5 CU	avg	*2213.63*	***32.81***	*4876.28*	*97985.80*	*3.91*	*0.04*
	min	1893.86	28.84	3.17	0.00	3.21	0.00
	max	2510.56	36.75	54911.78	507614.00	5.20	0.14
	std	169.45	2.21	12616.56	193458.00	0.40	0.04
12h 20ch 5 SC	avg	*1.16*	***0.08***	*0.15*	*6.60*	*0.11*	*0.36*
	min	0.85	0.05	0.07	0.00	0.06	0.00
	max	1.55	0.10	0.41	63.00	0.14	1.18
	std	0.19	0.01	0.07	13.64	0.02	0.39
12h 50ch 5 SC	avg	*8.45*	***0.44***	*0.62*	*7.88*	*0.37*	*0.31*
	min	6.22	0.33	0.24	0.00	0.26	0.00
	max	10.62	0.68	2.32	74.00	0.49	1.54
	std	0.92	0.06	0.45	13.62	0.06	0.40

Instance		*DP*	*DP**	*BC*		*BC* 1st solution	
		time	time	time	BBnodes	time	gap
24h 20ch 5 SC	avg	*10.41*	*0.46*	***0.37***	*11.08*	*0.23*	*0.15*
	min	7.51	0.36	0.14	0.00	0.13	0.00
	max	12.85	0.57	1.23	89.00	0.31	0.60
	std	1.00	0.04	0.26	21.52	0.04	0.18
24h 50ch 5 SC	avg	*72.54*	*2.42*	***1.62***	*9.34*	*0.90*	*0.08*
	min	53.62	1.86	0.62	0.00	0.64	0.00
	max	93.84	2.92	6.85	78.00	1.18	0.43
	std	7.32	0.22	1.56	19.96	0.13	0.14
72h 20ch 5 SC	avg	*290.84*	*6.46*	***1.33***	*7.00*	*0.91*	*0.04*
	min	242.46	5.58	0.54	0.00	0.54	0.00
	max	347.07	7.60	3.98	74.00	1.35	0.17
	std	26.05	0.52	0.81	14.30	0.15	0.05
72h 50ch 5 SC	avg	*2232.33*	*32.54*	***6.84***	*9.94*	*3.51*	*0.04*
	min	1882.83	28.25	2.31	0.00	2.37	0.00
	max	2581.61	37.04	32.29	89.00	4.44	0.15
	std	178.88	2.30	6.45	18.98	0.56	0.05
12h 20ch 5 TSC	avg	*1.20*	***0.08***	***0.10***	*13.64*	*0.08*	*0.60*
	min	0.71	0.06	0.05	0.00	0.05	0.00
	max	1.86	0.13	0.28	149.00	0.10	3.27
	std	0.20	0.01	0.05	30.07	0.01	0.90
12h 50ch 5 TSC	avg	*8.43*	***0.45***	*0.81*	*73.40*	*0.41*	*0.50*
	min	6.88	0.37	0.30	0.00	0.30	0.00
	max	10.14	0.54	6.78	1164.00	0.47	2.96
	std	0.81	0.04	1.16	217.47	0.04	0.65
24h 20ch 5 TSC	avg	*10.44*	*0.47*	***0.27***	*30.52*	*0.16*	*0.36*
	min	8.80	0.40	0.10	0.00	0.11	0.00
	max	13.26	0.57	1.34	307.00	0.20	1.72
	std	1.09	0.04	0.22	63.06	0.02	0.40
24h 50ch 5 TSC	avg	*71.63*	*2.45*	***1.34***	*27.50*	*0.92*	*0.36*
	min	60.16	2.07	0.72	0.00	0.71	0.00
	max	84.99	2.92	3.93	330.00	1.06	2.29
	std	5.53	0.19	0.71	58.50	0.07	0.49
72h 20ch 5 TSC	avg	*297.37*	*6.63*	***1.07***	*35.34*	*0.56*	*0.16*
	min	246.82	5.77	0.36	0.00	0.37	0.00
	max	351.63	7.60	3.67	228.00	0.67	0.63
	std	24.14	0.46	0.72	56.89	0.06	0.18
72h 50ch 5 TSC	avg	*2226.48*	*32.85*	***6.85***	*110.54*	*3.15*	*0.13*
	min	1782.29	27.83	2.34	0.00	2.40	0.00
	max	2581.23	37.21	25.51	1083.00	3.62	0.52
	std	166.35	2.06	5.79	232.14	0.35	0.16
12h 20ch 5 TWC	avg	*1.16*	***0.08***	***0.10***	*11.16*	*0.07*	*0.63*
	min	0.71	0.05	0.05	0.00	0.04	0.00
	max	1.72	0.11	0.28	119.00	0.10	4.40
	std	0.20	0.01	0.04	22.40	0.01	1.07

Instance		*DP* time	*DP** time	*BC* time	*BC* BBnodes	*BC* 1st solution time	*BC* 1st solution gap
12h 50ch 5 TWC	avg	*8.44*	***0.45***	*0.57*	*27.34*	*0.42*	*0.70*
	min	6.37	0.35	0.29	0.00	0.30	0.00
	max	11.05	0.59	1.23	341.00	0.50	4.14
	std	1.11	0.05	0.21	57.50	0.04	0.90
24h 20ch 5 TWC	avg	*10.32*	*0.47*	***0.26***	*25.04*	*0.16*	*0.36*
	min	8.10	0.38	0.10	0.00	0.10	0.00
	max	13.59	0.58	1.10	258.00	0.20	1.60
	std	1.06	0.04	0.19	46.76	0.02	0.48
24h 50ch 5 TWC	avg	*71.09*	*2.41*	***1.28***	*26.44*	*0.91*	*0.37*
	min	57.22	1.96	0.67	0.00	0.67	0.00
	max	88.59	3.12	2.98	344.00	1.03	1.60
	std	7.35	0.26	0.60	58.96	0.08	0.49
72h 20ch 5 TWC	avg	*297.63*	*6.62*	***0.97***	*48.32*	*0.53*	*0.14*
	min	257.14	5.71	0.40	0.00	0.39	0.00
	max	364.60	7.85	5.41	785.00	0.64	0.68
	std	24.66	0.47	0.86	127.72	0.06	0.17
72h 50ch 5 TWC	avg	*2226.73*	*32.87*	***3.62***	*5.36*	*3.05*	*0.09*
	min	1891.13	28.79	2.35	0.00	2.37	0.00
	max	2512.74	36.93	12.10	90.00	3.49	0.59
	std	164.03	2.19	1.49	13.84	0.29	0.15
12h 20ch 7 CU	avg	*0.98*	***0.07***	*0.12*	*1.70*	*0.11*	*0.09*
	min	0.71	0.06	0.06	0.00	0.07	0.00
	max	1.21	0.10	0.19	13.00	0.15	0.65
	std	0.12	0.01	0.03	3.24	0.02	0.18
12h 50ch 7 CU	avg	*7.10*	***0.41***	*0.55*	*1.28*	*0.52*	*0.05*
	min	5.81	0.33	0.39	0.00	0.38	0.00
	max	8.32	0.50	0.84	17.00	0.72	0.71
	std	0.68	0.04	0.12	3.03	0.08	0.13
24h 20ch 7 CU	avg	*9.02*	*0.46*	***0.26***	*1.86*	*0.24*	*0.05*
	min	6.92	0.37	0.12	0.00	0.16	0.00
	max	11.25	0.56	0.46	28.00	0.34	0.40
	std	1.03	0.05	0.07	4.61	0.04	0.09
24h 50ch 7 CU	avg	*61.57*	*2.27*	***1.29***	*2.98*	*1.19*	*0.03*
	min	51.70	1.99	0.87	0.00	0.89	0.00
	max	75.76	2.73	1.92	31.00	1.39	0.28
	std	4.99	0.17	0.27	6.17	0.13	0.06
72h 20ch 7 CU	avg	*262.63*	*6.54*	***0.98***	*1.90*	*0.89*	*0.01*
	min	209.71	5.32	0.57	0.00	0.59	0.00
	max	322.56	7.46	1.67	19.00	1.14	0.10
	std	23.28	0.51	0.26	4.42	0.14	0.02
72h 50ch 7 CU	avg	*1951.64*	*32.02*	***4.69***	*4.98*	*4.02*	*0.03*
	min	1664.09	27.58	2.98	0.00	2.96	0.00
	max	2188.43	36.89	9.53	56.00	5.03	0.20
	std	135.84	2.22	1.32	10.17	0.52	0.06

Instance		*DP*	*DP**	*BC*		*BC* 1st solution	
		time	**time**	**time**	**BBnodes**	**time**	**gap**
12h 20ch 7 SC	avg	*1.00*	***0.08***	*0.17*	*11.02*	*0.10*	*0.34*
	min	0.62	0.06	0.07	0.00	0.06	0.00
	max	1.44	0.10	0.47	85.00	0.17	4.41
	std	0.16	0.01	0.10	17.60	0.02	0.65
12h 50ch 7 SC	avg	*7.36*	***0.43***	*0.61*	*10.42*	*0.37*	*0.30*
	min	6.09	0.35	0.24	0.00	0.25	0.00
	max	8.77	0.51	1.71	55.00	0.45	0.85
	std	0.61	0.04	0.36	14.53	0.05	0.32
24h 20ch 7 SC	avg	*8.99*	*0.45*	***0.29***	*7.16*	*0.20*	*0.12*
	min	7.32	0.36	0.12	0.00	0.12	0.00
	max	11.69	0.55	0.95	73.00	0.28	0.56
	std	0.98	0.04	0.16	13.19	0.04	0.16
24h 50ch 7 SC	avg	*61.89*	*2.32*	*1.66*	*11.44*	*0.90*	*0.13*
	min	50.96	1.94	0.60	0.00	0.59	0.00
	max	73.17	2.65	9.06	148.00	1.12	0.46
	std	5.01	0.18	1.44	23.06	0.13	0.15
72h 20ch 7 SC	avg	*257.46*	*6.53*	***1.45***	*9.54*	*0.89*	*0.04*
	min	216.54	5.54	0.54	0.00	0.55	0.00
	max	291.83	7.44	3.58	57.00	1.18	0.14
	std	20.20	0.48	0.77	14.48	0.13	0.05
72h 50ch 7 SC	avg	*1961.56*	*32.06*	***5.97***	*8.68*	*3.36*	*0.04*
	min	1759.07	28.65	2.28	0.00	2.27	0.00
	max	2194.47	35.92	47.87	129.00	4.20	0.15
	std	123.54	1.93	7.01	21.63	0.39	0.05
12h 20ch 7 TSC	avg	*1.01*	***0.08***	*0.12*	*13.30*	*0.09*	*0.69*
	min	0.75	0.06	0.04	0.00	0.06	0.00
	max	1.37	0.10	0.43	94.00	0.12	2.42
	std	0.14	0.01	0.07	23.73	0.02	0.78
12h 50ch 7 TSC	avg	*7.17*	***0.42***	*0.60*	*20.44*	*0.44*	*0.54*
	min	5.70	0.34	0.33	0.00	0.31	0.00
	max	8.60	0.51	1.94	247.00	0.52	3.87
	std	0.70	0.04	0.32	50.99	0.05	0.93
24h 20ch 7 TSC	avg	*9.12*	*0.46*	***0.29***	*27.90*	*0.18*	*0.37*
	min	7.02	0.38	0.12	0.00	0.11	0.00
	max	10.70	0.54	1.17	259.00	0.22	1.63
	std	0.85	0.04	0.20	56.31	0.03	0.43
24h 50ch 7 TSC	avg	*62.51*	*2.35*	***1.51***	*32.18*	*0.99*	*0.37*
	min	50.70	2.00	0.70	0.00	0.73	0.00
	max	76.67	2.86	4.88	370.00	1.23	1.29
	std	5.66	0.21	0.90	72.48	0.11	0.41
72h 20ch 7 TSC	avg	*261.83*	*6.61*	***1.11***	*30.22*	*0.62*	*0.15*
	min	226.57	5.70	0.44	0.00	0.43	0.00
	max	306.24	7.58	3.53	345.00	0.76	0.73
	std	20.69	0.49	0.78	62.37	0.08	0.18

Instance		*DP*	*DP**	*BC*		*BC* 1st solution	
		time	time	time	BBnodes	time	gap
72h 50ch 7 TSC	avg	*1910.72*	*31.54*	***6.75***	*61.22*	*3.43*	*0.18*
	min	1662.03	27.76	2.52	0.00	2.55	0.00
	max	2227.28	36.17	37.28	1267.00	4.00	0.54
	std	136.56	2.15	6.66	189.83	0.33	0.17
12h 20ch 7 TWC	avg	*0.99*	***0.08***	*0.12*	*14.74*	*0.08*	*0.67*
	min	0.72	0.06	0.05	0.00	0.06	0.00
	max	1.22	0.10	0.37	165.00	0.11	5.03
	std	0.12	0.01	0.07	30.58	0.01	0.98
12h 50ch 7 TWC	avg	*7.15*	***0.42***	*0.52*	*9.26*	*0.44*	*0.58*
	min	5.83	0.34	0.31	0.00	0.35	0.00
	max	8.47	0.52	1.16	116.00	0.53	2.99
	std	0.68	0.04	0.17	20.98	0.04	0.89
24h 20ch 7 TWC	avg	*9.07*	*0.46*	***0.27***	*21.02*	*0.17*	*0.39*
	min	6.94	0.37	0.11	0.00	0.10	0.00
	max	11.30	0.56	0.75	146.00	0.22	1.44
	std	1.07	0.05	0.14	36.31	0.02	0.44
24h 50ch 7 TWC	avg	*61.76*	*2.29*	***1.54***	*32.52*	*1.01*	*0.46*
	min	52.12	2.00	0.70	0.00	0.70	0.00
	max	75.72	2.76	4.11	276.00	1.15	1.83
	std	5.06	0.17	0.73	59.75	0.10	0.46
72h 20ch 7 TWC	avg	*259.47*	*6.57*	***1.00***	*21.54*	*0.60*	*0.15*
	min	209.39	5.34	0.42	0.00	0.42	0.00
	max	301.36	7.52	2.79	141.00	0.74	0.57
	std	21.65	0.51	0.59	35.72	0.08	0.16
72h 50ch 7 TWC	avg	*1956.43*	*32.21*	***4.85***	*17.44*	*3.32*	*0.12*
	min	1630.04	27.80	2.53	0.00	2.49	0.00
	max	2270.47	36.79	13.25	140.00	3.99	0.65
	std	138.10	2.20	2.69	32.68	0.35	0.16

B

Numerical Results for Maximum Clique

B.1 Characteristics of the DIMACS Benchmark Set

The DIMACS benchmark set for cliques [122, 67] consists of 66 instances, ranging from 28 – 3361 nodes and 420 – 11 million edges. The table below presents number of nodes ($|V|$), number of edges ($|E|$), resulting graph density in percent (density), and smallest and largest node degree (Δ_{min}, Δ_{max}), respectively, of the 66 instances in the DIMACS test set.

Instance	$\|V\|$	$\|E\|$	density	Δ_{min}	Δ_{max}
brock200_1	200	14834	74.54%	130	165
brock200_2	200	9876	49.63%	78	114
brock200_3	200	12048	60.54%	99	134
brock200_4	200	13089	65.77%	112	147
brock400_1	400	59723	74.84%	272	320
brock400_2	400	59786	74.92%	274	328
brock400_3	400	59681	74.79%	275	322
brock400_4	400	59765	74.89%	275	326
brock800_1	800	207505	64.93%	477	560
brock800_2	800	208166	65.13%	472	566
brock800_3	800	207333	64.87%	474	558
brock800_4	800	207643	64.97%	481	565
c-fat200-1	200	1534	7.71%	14	17
c-fat200-2	200	3235	16.26%	32	34
c-fat200-5	200	8473	42.58%	83	86
c-fat500-1	500	4459	3.57%	17	20
c-fat500-10	500	46627	37.38%	185	188
c-fat500-2	500	9139	7.33%	35	38
c-fat500-5	500	23191	18.59%	92	95
hamming10-2	1024	518656	99.02%	1013	1013
hamming10-4	1024	434176	82.89%	848	848
hamming6-2	64	1824	90.48%	57	57
hamming6-4	64	704	34.92%	22	22
hamming8-2	256	31616	96.86%	247	247
hamming8-4	256	20864	63.92%	163	163
johnson16-2-4	120	5460	76.47%	91	91
johnson32-2-4	496	107880	87.88%	435	435
johnson8-2-4	28	210	55.56%	15	15
johnson8-4-4	70	1855	76.81%	53	53
MANN_a9	45	918	92.73%	40	41
MANN_a27	378	70551	99.01%	364	374
MANN_a45	1035	533115	99.63%	1012	1031
MANN_a81	3321	5506380	99.88%	3280	3317

Instance	$\|V\|$	$\|E\|$	density	Δ_{min}	Δ_{max}
keller4	171	9435	64.91%	102	124
keller5	776	225990	75.15%	560	638
keller6	3361	4619898	81.82%	2690	2952
p_hat1000-1	1000	122253	24.48%	105	408
p_hat1000-2	1000	244799	49.01%	232	766
p_hat1000-3	1000	371746	74.42%	582	895
p_hat1500-1	1500	284923	25.34%	157	614
p_hat1500-2	1500	568960	50.61%	335	1153
p_hat1500-3	1500	847244	75.36%	912	1330
p_hat300-1	300	10933	24.38%	23	132
p_hat300-2	300	21928	48.89%	59	229
p_hat300-3	300	33390	74.45%	168	267
p_hat500-1	500	31569	25.31%	52	204
p_hat500-2	500	62946	50.46%	117	389
p_hat500-3	500	93800	75.19%	293	452
p_hat700-1	700	60999	24.93%	75	286
p_hat700-2	700	121728	49.76%	157	539
p_hat700-3	700	183010	74.80%	408	627
san1000	1000	250500	50.15%	445	550
san200_0.7_1	200	13930	70.00%	125	155
san200_0.7_2	200	13930	70.00%	103	164
san200_0.9_1	200	17910	90.00%	159	191
san200_0.9_2	200	17910	90.00%	169	188
san200_0.9_3	200	17910	90.00%	166	187
san400_0.5_1	400	39900	50.00%	174	225
san400_0.7_1	400	55860	70.00%	257	301
san400_0.7_2	400	55860	70.00%	257	304
san400_0.7_3	400	55860	70.00%	250	307
san400_0.9_1	400	71820	90.00%	341	374
sanr200_0.7	200	13868	69.69%	120	161
sanr200_0.9	200	17863	89.76%	166	189
sanr400_0.5	400	39984	50.11%	161	233
sanr400_0.7	400	55869	70.01%	252	310

B.2 Detailed Results for the DIMACS Benchmark Set

In the following two tables we summarize the major characteristics of Tables 10.1, 10.2 and Tables B.1, B.2, respectively. "Instances" gives the number of instances tackled, "within time limit" gives the number of those that could be solved within 21 600 sec. "best $|C^*|$" refers to the number of best clique sizes found among the four approaches. The next three rows show how often the best clique size was found fastest, or with fewest branch-and-bound nodes, or simultaneously fastest and with fewest branch-and-bound nodes, respectively. Row "overall best" rates the results according to Def. 14. Finally, "total time" accumulates the running time over all instances and "total size" accumulates the best clique sizes found.

Summary for Tables 10.1 and 10.2 (pages 158 and 159)

	Alg. 20	Alg. 20+DF	Alg. 20+$\mathcal{U}_8$	Alg. 20+DF+$\mathcal{U}_8$
Instances:	66	66	66	66
within timelimit:	33	33	46	46
best $\|C^*\|$:	46	41	57	60
fastest termination for best $\|C^*\|$:	42	33	23	26
best $\|C^*\|$ with fewest bb nodes:	5	22	0	45
fastest termination for best $\|C^*\|$ with fewest bb nodes:	0	3	0	25
fastest best $\|C^*\|$ found:	36	22	17	20
overall best:	26	4	7	33
total time :	210.08 h	209.84 h	130.21 h	129.72 h
total size :	3103	3121	3928	3934

Summary for Tables B.1 and B.2 (pages 193 and 194)

	Alg. 21	Alg. 21+DF	Alg. 21+$\mathcal{U}_8$	Alg. 21+DF+$\mathcal{U}_8$
Instances:	66	66	66	66
within timelimit:	34	34	46	46
best $\|C^*\|$:	49	48	62	64
fastest termination for best $\|C^*\|$:	54	42	29	30
best $\|C^*\|$ with fewest bb nodes:	4	25	5	44
fastest termination for best $\|C^*\|$ with fewest bb nodes:	0	12	3	28
fastest best $\|C^*\|$ found:	48	36	16	29
overall best:	21	12	11	34
total time :	197.75 h	199.01 h	127.46 h	127.13 h
total size :	3840	3841	3844	4077

Instance	Alg. 21			Alg. 21+DF			Alg. 21+χ			Alg. 21+DF+χ		
	$\|C^*\|$	time	BB nodes	$\|C^*\|$	time	BB nodes	$\|C^*\|$	time	BB nodes	$\|C^*\|$	time	BB nodes
brock200_1	**21**	**40.99**	**40763864**	21	52.65	3809437	21	91.26	42767	21	92.73	40856
brock200_2	**12**	**0.08**	**47946**	12	0.13	3088	12	0.24	377	12	0.27	312
brock200_3	**15**	**0.56**	**568162**	15	0.92	47432	15	3.49	2839	15	3.65	2694
brock200_4	**17**	**1.26**	**1058449**	17	1.99	82857	17	4.63	2457	17	4.92	2295
brock400_1	**≥25**	**≥21600**	**16744343445**	≥25	≥21600	1047235395	≥25	≥21600	5998687	≥25	≥21600	5705834
brock400_2	**≥25**	**≥21600**	**17624720710**	≥25	≥21600	1185423407	≥25	≥21600	6710250	≥25	≥21600	6428350
brock400_3	**≥24**	**≥21600**	**17960143939**	≥24	≥21600	1191320924	≥24	≥21600	6745388	≥24	≥21600	6504665
brock400_4	**≥25**	**≥21600**	**16569459718**	≥25	≥21600	1019550053	≥25	≥21600	5902546	≥25	≥21600	5702967
brock800_1	**≥21**	**≥21600**	**15763475452**	≥20	≥21600	694985151	≥20	≥21600	7531481	≥20	≥21600	7062587
brock800_2	**≥21**	**≥21600**	**15364807482**	≥21	≥21600	703074375	≥20	≥21600	7516642	≥20	≥21600	7019896
brock800_3	**≥22**	**≥21600**	**12984394338**	≥22	≥21600	499594513	≥22	≥21600	5747476	≥22	≥21600	5381786
brock800_4	**≥20**	**≥21600**	**16035660281**	≥20	≥21600	709682147	≥20	≥21600	7490461	≥20	≥21600	6979390
c-fat200-1	12	0.01	266	**12**	**0.01**	**211**	12	0.01	266	**12**	**0.01**	**211**
c-fat200-2	24	0.01	476	**24**	**0.01**	**223**	24	0.03	476	**24**	**0.01**	**223**
c-fat200-5	58	1822.34	268437253	**58**	**0.01**	**257**	58	0.53	1853	**58**	**0.01**	**257**
c-fat500-1	14	0.05	591	**14**	**0.05**	**513**	14	0.06	591	**14**	**0.04**	**513**
c-fat500-10	126	0.12	8375	**126**	**0.11**	**625**	126	9.65	8375	**126**	**0.13**	**625**
c-fat500-2	26	0.05	825	**26**	**0.05**	**525**	26	0.08	825	**26**	**0.06**	**525**
c-fat500-5	64	0.07	2516	**64**	**0.08**	**563**	64	0.80	2516	**64**	**0.06**	**563**
hamming10-2	512	6.19	131840	**512**	**5.16**	**1535**	512	2425.72	131840	**512**	**5.22**	**1535**
hamming10-4	≥32	≥21600	32259594986	≥32	≥21600	5396081984	≥32	≥21600	5939826	**≥32**	**≥21600**	**5795183**
hamming6-2	32	0.01	560	**32**	**0.01**	**95**	32	0.06	560	**32**	**0.01**	**95**
hamming6-4	4	0.01	230	**4**	**0.01**	**67**	4	0.01	230	**4**	**0.01**	**67**
hamming8-2	128	0.07	8384	**128**	**0.07**	**383**	128	10.29	8384	**128**	**0.07**	**383**
hamming8-4	16	5.80	3742273	16	7.85	252454	**16**	**2.75**	**862**	16	3.20	733
johnson16-2-4	**8**	**2.09**	**4177665**	8	3.63	904465	8	11.10	122781	8	12.23	58009
johnson32-2-4	≥16	≥21600	46937027805	≥16	≥21600	7962372028	≥16	≥21600	15120429	**≥16**	**≥21600**	**14886797**
johnson8-2-4	4	0.01	99	**4**	**0.01**	**31**	4	0.01	99	**4**	**0.01**	**31**
johnson8-4-4	14	0.01	8416	14	0.01	1084	14	0.01	164	**14**	**0.01**	**92**
keller4	**11**	**0.63**	**1003121**	11	0.92	133911	11	2.01	2947	11	1.99	3043
keller5	≥20	≥21600	34083725546	≥20	≥21600	5207232560	≥20	≥21600	13132294	**≥20**	**≥21600**	**13350366**
keller6	≥31	≥21600	40974572043	≥31	≥21600	23153877676	≥39	≥21600	1306103	**≥39**	**≥21600**	**1170365**
MANN_a27	≥117	≥21600	8925632960	≥117	≥21600	23181265259	**126**	**887.82**	**26396**	126	1833.36	25867
MANN_a45	≥330	≥21600	3033506091	≥330	≥21600	22952138384	≥342	≥21600	107901	**≥342**	**≥21600**	**17512**

Table B.1: *Algorithm 21 on DIMACS Instances (Part I). We compare the number of choice points and running time in seconds of our implementation of Östergård's idea, our algorithm using coloring bounds only (χ), our algorithm using domain filtering only (DF), and a fourth using both ($\chi + DF$). Boldface numbers indicate best runs.* *(see also summary on page 192).*

Instance	Alg. 21			Alg. 21+*DF*			Alg. 21+χ			Alg. 21+*DF*+χ		
	$\lvert C^*\rvert$	time	BB nodes	$\lvert C^*\rvert$	time	BB nodes	$\lvert C^*\rvert$	time	BB nodes	$\lvert C^*\rvert$	time	BB nodes
MANN_a81	≥1080	≥21600	907536555	≥1080	≥21600	23566687196	≥848	≥21600	360102	**≥1080**	**≥21600**	**5143**
MANN_a9	16	0.08	274543	16	0.13	138787	16	0.01	178	**16**	**0.01**	**129**
p_hat1000-1	**10**	**3.17**	**1686442**	10	8.45	75156	10	19.10	13404	10	22.46	12837
p_hat1000-2	≥46	≥21600	15833817626	≥46	≥21600	2235855371	≥46	≥21600	1379492	**≥46**	**≥21600**	**1386354**
p_hat1000-3	≥60	≥21600	19385985248	≥61	≥21600	8438845622	≥64	≥21600	1200137	**≥65**	**≥21600**	**1183638**
p_hat1500-1	**12**	**27.54**	**15753173**	12	84.23	703941	12	235.02	107530	12	274.43	106328
p_hat1500-2	≥61	≥21600	16053867267	≥62	≥21600	4575631546	≥65	≥21600	844836	**≥65**	**≥21600**	**835237**
p_hat1500-3	≥85	≥21600	13435662980	≥85	≥21600	11872713614	≥92	≥21600	443039	**≥92**	**≥21600**	**412249**
p_hat300-1	8	0.04	8428	8	0.05	363	8	0.04	328	**8**	**0.05**	**309**
p_hat300-2	25	1.23	944950	25	1.90	107966	**25**	**0.83**	**630**	25	0.92	361
p_hat300-3	36	1071.48	774525947	36	1365.33	95821409	36	165.33	18873	**36**	**166.16**	**17439**
p_hat500-1	**9**	**0.20**	**83943**	9	0.40	4821	9	0.29	560	9	0.44	529
p_hat500-2	36	159.23	100476737	36	225.26	11681596	**36**	**34.57**	**4163**	36	36.35	3579
p_hat500-3	≥50	≥21600	15316807664	≥50	≥21600	2396804695	≥50	≥21600	1183948	**≥50**	**≥21600**	**1134209**
p_hat700-1	**11**	**0.59**	**187935**	11	1.29	10591	11	0.73	758	11	1.26	721
p_hat700-2	44	7857.52	5022334663	44	9758.65	628055012	**44**	**447.60**	**30107**	44	456.30	27572
p_hat700-3	≥62	≥21600	14841334382	≥62	≥21600	3073251854	≥62	≥21600	941276	**≥62**	**≥21600**	**939722**
san1000	≥8	≥21600	40251583444	≥8	≥21600	3826623750	**15**	**101.81**	**8185**	15	142.59	9769
san200_0.7_1	≥15	≥21600	72167962008	≥15	≥21600	23884042229	30	0.43	675	**30**	**0.41**	**440**
san200_0.7_2	≥12	≥21600	73596061385	≥12	≥21600	21011103362	**18**	**0.26**	**358**	18	0.29	386
san200_0.9_1	≥45	≥21600	26762063204	≥45	≥21600	21781906220	70	1.88	2615	**70**	**0.90**	**726**
san200_0.9_2	≥35	≥21600	36870465389	≥35	≥21600	22185945440	60	2.06	2078	**60**	**1.64**	**827**
san200_0.9_3	≥24	≥21600	54510972529	≥24	≥21600	22523065742	44	449.27	69006	**44**	**447.97**	**66651**
san400_0.5_1	13	3257.59	4714133041	13	4876.80	450539488	**13**	**1.23**	**525**	13	1.61	714
san400_0.7_1	≥20	≥21600	63144249943	≥20	≥21600	22963713648	**40**	**13.14**	**3017**	40	14.24	2570
san400_0.7_2	≥15	≥21600	73952176568	≥15	≥21600	23623596407	**30**	**97.36**	**11358**	30	114.90	12762
san400_0.7_3	≥12	≥21600	76892809612	≥12	≥21600	21108189660	**22**	**606.96**	**145060**	22	766.37	178388
san400_0.9_1	≥50	≥21600	24266146323	≥50	≥21600	22766607890	100	534.17	35170	**100**	**493.40**	**27224**
sanr200_0.7	**18**	**7.37**	**6506241**	18	10.35	530953	18	21.38	9449	18	21.82	8887
sanr200_0.9	≥41	≥21600	17875107504	≥41	≥21600	2160419473	42	7010.62	1007007	**42**	**6853.37**	**859718**
sanr400_0.5	**13**	**3.80**	**2902155**	13	7.89	178928	13	25.41	12476	13	27.65	12097
sanr400_0.7	**21**	**6432.37**	**5226950345**	21	8831.31	376886905	21	13619.40	4762337	21	13877.50	4531210

Table B.2: *Algorithm 21 on DIMACS Instances (Part II). We compare the number of choice points and running time in seconds of our implementation of Östergård's idea, our algorithm using coloring bounds only (χ), our algorithm using domain filtering only (DF), and a fourth using both (χ + DF). Boldface numbers indicate best runs. (see also summary on page 192).*

B.3 Single Statistics

The following tables present the results of 20 different combinations of techniques and algorithms for the maximum clique problem. Boldface numbers indicate that the corresponding method found a best result (according to Def. 14) among these 20 competitors.

In Tables B.11 – B.18 we use *reordering* in addition to techniques described before. Node reordering follows an idea presented in Fahle [70]. In a more intensive study it turned out to have only a minor impact on the overall convergence when combined with other techniques.

For the same reasons that prevent the use of primal heuristics in Östergård's approach, reordering cannot be applied in Alg. 21. Thus, we only test it on Alg. 20.

Reordering is introduced in Alg. 20 just before the next recursion is called. There, we sort the nodes according to the remaining degree in P in decreasing order. In doing so, dynamic reordering ensures that the most interesting nodes are considered first for branching in each part of the search tree. Notice that the original algorithm sorts nodes according to their original degree in G only at the beginning and thus uses a static ordering of variables.

Reordering is done by a standard quicksort algorithm, or — if $|P'| < 100$ — by an insertion sort. This switching parameter was determined empirically and reflects the fact that insertion sort benefits from only a few elements changing their relative position after the update step. By switching the sorting algorithms, we often gain some running time.

Reordering has a lesser impact on the overall quality than the other techniques considered before. On selected instances, reordering can significantly improve convergence. Over the entire benchmark set, this effect diminishes, although reordering occurs alongside with some reduction in branch-and-bound nodes. Apparently, reordering nodes makes sense whenever the initial static ordering would give bad advice for branching. This is usually the case in the right part of the search tree. Since domain filtering and heuristics tend to support finding good solutions early and since sorting produces a time overhead the impact of reordering should not be overestimated. Indeed, the accumulated clique size is often smaller when comparing the experiments using reordering with their counterparts using only static ordering.

Table B.3: *Alg. 20*

Instance	$\lvert C^*\rvert$	time	BB node	time best
brock200_1	**21**	**26.58**	**38043497**	**4.46**
brock200_2	**12**	**0.04**	**54314**	**0.01**
brock200_3	**15**	**0.37**	**453265**	**0.01**
brock200_4	17	1.61	2161909	1.13
brock400_1	**≥27**	**≥21600.00**	**23223752963**	**15951.80**
brock400_2	≥29	≥21600.00	20089180565	4015.51
brock400_3	**≥31**	**≥21600.00**	**25770803689**	**18788.40**
brock400_4	33	19773.90	20152485865	13264.70
brock800_1	**≥21**	**≥21600.00**	**21101899632**	**2828.48**
brock800_2	≥21	≥21600.00	23496231361	18899.50
brock800_3	≥21	≥21600.00	22124014822	12254.90
brock800_4	**≥26**	**≥21600.00**	**21662133367**	**20245.20**
c-fat200-1	12	0.01	52	0.01
c-fat200-2	24	0.01	444	0.01
c-fat200-5	58	1207.72	268435599	1207.72
c-fat500-1	14	0.01	71	0.01
c-fat500-10	≥124	≥21600.00	15858231004	0.01
c-fat500-2	26	0.01	683	0.01
c-fat500-5	64	4.92	5703074	4.91
hamming10-2	≥512	≥21600.00	3366474297	0.20
hamming10-4	≥32	≥21600.00	36567495385	0.01
hamming6-2	32	0.01	42787	0.01
hamming6-4	4	0.01	221	0.01
hamming8-2	≥128	≥21600.00	14850398859	0.01
hamming8-4	16	4.06	3742143	0.01
johnson16-2-4	**8**	**1.54**	**4177630**	**0.01**
johnson32-2-4	≥16	≥21600.00	49208418864	0.01
johnson8-2-4	4	0.01	90	0.01
johnson8-4-4	14	0.01	12544	0.01
keller4	11	0.74	1275236	0.01
keller5	≥24	≥21600.00	26405722472	3151.67
keller6	≥43	≥21600.00	19876767579	11626.10
MANN_a27	≥116	≥21600.00	17111883459	11539.70
MANN_a45	≥220	≥21600.00	12427538213	1364.73
MANN_a81	≥438	≥21600.00	6433965301	42.34
MANN_a9	16	0.09	329235	0.01
p_hat1000-1	**10**	**2.05**	**1801019**	**0.16**
p_hat1000-2	≥42	≥21600.00	20151659983	11737.90
p_hat1000-3	≥49	≥21600.00	22757998877	17131.10
p_hat1500-1	**12**	**18.28**	**13825580**	**3.25**
p_hat1500-2	≥46	≥21600.00	24297112095	11607.90
p_hat1500-3	≥53	≥21600.00	27789610940	8036.53
p_hat300-1	8	0.01	11634	0.01
p_hat300-2	25	1.00	1553140	0.56
p_hat300-3	36	1492.43	1960518988	1469.14
p_hat500-1	**9**	**0.08**	**89908**	**0.01**
p_hat500-2	36	240.41	292274682	240.23
p_hat500-3	≥44	≥21600.00	25719412910	15486.00
p_hat700-1	**11**	**0.36**	**343857**	**0.34**
p_hat700-2	44	9742.35	10969888234	8534.67
p_hat700-3	≥50	≥21600.00	24323367924	21101.60
san1000	≥10	≥21600.00	12277896744	10265.60
san200_0.7_1	30	5466.91	14265131069	4713.39
san200_0.7_2	≥18	≥21600.00	31159815462	70.94
san200_0.9_1	≥48	≥21600.00	47701178923	1063.20
san200_0.9_2	≥41	≥21600.00	48882914444	12808.90
san200_0.9_3	≥36	≥21600.00	2192973297	657.81
san400_0.5_1	13	916.73	1414713059	49.51
san400_0.7_1	≥22	≥21600.00	87494021010	3512.73
san400_0.7_2	≥17	≥21600.00	61966653095	9542.01
san400_0.7_3	≥22	≥21600.00	26018343618	13896.70
san400_0.9_1	≥49	≥21600.00	54964009156	183.34
sanr200_0.7	**18**	**5.83**	**7985123**	**1.25**
sanr200_0.9	≥40	≥21600.00	29331657711	6979.29
sanr400_0.5	13	4.06	4283999	2.05
sanr400_0.7	**21**	**4587.91**	**5624332491**	**9.70**

Table B.4: *Alg. 20 + Domain Filtering*

Instance	$\|C^*\|$	time	BB node	time best
brock200_1	21	54.60	3350353	9.02
brock200_2	12	0.17	3539	0.03
brock200_3	15	1.04	32170	0.01
brock200_4	17	3.87	167750	2.72
brock400_1	≥25	≥21600.00	1022982977	14507.30
brock400_2	≥29	≥21600.00	760111221	7438.01
brock400_3	≥24	≥21600.00	1057635922	38.56
brock400_4	≥25	≥21600.00	935193147	955.69
brock800_1	≥21	≥21600.00	546659121	6339.46
brock800_2	≥20	≥21600.00	636709653	693.76
brock800_3	≥20	≥21600.00	602902311	227.53
brock800_4	≥21	≥21600.00	606259112	18028.90
c-fat200-1	12	0.01	5	0.01
c-fat200-2	24	0.01	5	0.01
c-fat200-5	58	0.01	5	0.01
c-fat500-1	14	0.05	5	0.05
c-fat500-10	126	0.07	5	0.07
c-fat500-2	26	0.05	5	0.05
c-fat500-5	64	0.06	5	0.06
hamming10-2	≥512	≥21600.00	770355321	0.48
hamming10-4	≥32	≥21600.00	2815903831	0.45
hamming6-2	32	0.01	1891	0.01
hamming6-4	4	0.01	47	0.01
hamming8-2	≥128	≥21600.00	1375703771	0.01
hamming8-4	16	7.89	252418	0.01
johnson16-2-4	8	3.62	904446	0.01
johnson32-2-4	≥16	≥21600.00	4929133669	0.06
johnson8-2-4	4	0.01	21	0.01
johnson8-4-4	14	0.01	1205	0.01
keller4	11	1.56	107086	0.01
keller5	≥24	≥21600.00	787089454	5057.18
keller6	≥43	≥21600.00	574383274	8096.65
MANN_a27	≥116	≥21600.00	22543342098	2926.39

Instance	$\|C^*\|$	time	BB node	time best
MANN_a45	≥234	≥21600.00	22952851665	6605.06
MANN_a81	≥467	≥21600.00	22372923762	2924.38
MANN_a9	16	0.11	92210	0.01
p_hat1000-1	10	9.74	81573	1.27
p_hat1000-2	≥41	≥21600.00	1095205574	13548.50
p_hat1000-3	≥47	≥21600.00	1426408001	5481.57
p_hat1500-1	12	92.85	454914	20.08
p_hat1500-2	≥46	≥21600.00	1848437250	17558.20
p_hat1500-3	≥53	≥21600.00	2097394574	7264.97
p_hat300-1	8	0.07	450	0.05
p_hat300-2	25	2.67	182168	1.61
p_hat300-3	36	2729.49	251846120	2689.14
p_hat500-1	9	0.51	4978	0.12
p_hat500-2	36	502.24	35324032	501.96
p_hat500-3	≥43	≥21600.00	1873805007	11546.10
p_hat700-1	11	2.00	20630	1.96
p_hat700-2	44	18281.20	1394776054	16148.50
p_hat700-3	≥49	≥21600.00	1554061118	11807.10
san1000	≥9	≥21600.00	179990506	407.10
san200_0.7_1	30	9273.97	3805796652	7670.94
san200_0.7_2	≥18	≥21600.00	4588848753	80.54
san200_0.9_1	≥48	≥21600.00	14808252136	743.49
san200_0.9_2	≥41	≥21600.00	3462650501	1123.33
san200_0.9_3	≥44	≥21600.00	1651744796	14325.50
san400_0.5_1	13	2255.23	155275559	102.52
san400_0.7_1	≥22	≥21600.00	16652377204	1554.65
san400_0.7_2	≥17	≥21600.00	7525272755	15565.60
san400_0.7_3	≥17	≥21600.00	1972475939	5246.27
san400_0.9_1	≥49	≥21600.00	6047721579	5.23
sanr200_0.7	18	12.15	665013	2.56
sanr200_0.9	≥40	≥21600.00	2144307525	10452.20
sanr400_0.5	13	12.16	251463	6.21
sanr400_0.7	21	9373.51	408188818	19.28

Table B.5: *Alg. 20 +* $\mathcal{U}_8$

Instance	$\lvert C^* \rvert$	time	BB node	time best
brock200_1	21	83.36	32847	16.10
brock200_2	12	0.49	323	0.13
brock200_3	15	3.15	1418	0.01
brock200_4	17	9.73	4442	7.13
brock400_1	≥25	≥21600.00	6477006	16084.70
brock400_2	≥29	≥21600.00	4976220	8808.90
brock400_3	≥24	≥21600.00	6672788	52.58
brock400_4	≥25	≥21600.00	5851408	1235.66
brock800_1	≥21	≥21600.00	6705439	12402.80
brock800_2	≥20	≥21600.00	7261032	1623.76
brock800_3	≥20	≥21600.00	7089985	547.95
brock800_4	≥20	≥21600.00	7239813	180.93
c-fat200-1	12	0.01	34	0.01
c-fat200-2	24	0.01	70	0.01
c-fat200-5	58	0.08	172	0.08
c-fat500-1	14	0.05	40	0.05
c-fat500-10	126	0.68	376	0.68
c-fat500-2	26	0.06	76	0.06
c-fat500-5	64	0.15	190	0.15
hamming10-2	512	0.76	513	0.75
hamming10-4	≥33	≥21600.00	5237411	17194.30
hamming6-2	32	0.01	33	0.01
hamming6-4	4	0.01	129	0.01
hamming8-2	128	0.01	129	0.01
hamming8-4	16	3.29	585	0.01
johnson16-2-4	8	11.21	123282	0.01
johnson32-2-4	≥16	≥21600.00	228095779	0.08
johnson8-2-4	4	0.01	50	0.01
johnson8-4-4	14	0.03	27	0.01
keller4	11	2.58	1598	0.03
keller5	≥26	≥21600.00	2844759	19070.30
keller6	≥43	≥21600.00	1695034	1269.69
MANN_a27	126	3627.65	31978	383.88

Instance	$\lvert C^* \rvert$	time	BB node	time best
MANN_a45	≥334	≥21600.00	26007	13816.70
MANN_a81	≥998	≥21600.00	33067	17009.30
MANN_a9	16	0.01	46	0.01
p_hat1000-1	10	27.45	14082	1.99
p_hat1000-2	≥44	≥21600.00	1003624	5044.39
p_hat1000-3	≥50	≥21600.00	921339	1552.40
p_hat1500-1	12	292.45	104668	57.33
p_hat1500-2	≥52	≥21600.00	673044	2849.27
p_hat1500-3	≥58	≥21600.00	983802	15063.80
p_hat300-1	8	0.19	237	0.11
p_hat300-2	25	2.38	647	1.71
p_hat300-3	36	508.07	79793	504.61
p_hat500-1	9	1.21	493	0.14
p_hat500-2	36	128.09	16235	128.07
p_hat500-3	≥49	≥21600.00	1052581	9766.99
p_hat700-1	11	3.96	871	3.79
p_hat700-2	44	1129.18	89869	1064.31
p_hat700-3	≥54	≥21600.00	801496	2854.44
san1000	15	895.33	35573	895.33
san200_0.7_1	30	0.90	389	0.90
san200_0.7_2	18	0.58	277	0.58
san200_0.9_1	70	30.97	5308	30.97
san200_0.9_2	60	628.92	85085	628.92
san200_0.9_3	**44**	**57.87**	**12622**	**53.09**
san400_0.5_1	13	2.59	2470	2.59
san400_0.7_1	40	123.09	7438	123.09
san400_0.7_2	30	44.96	6497	44.96
san400_0.7_3	22	290.52	65466	290.52
san400_0.9_1	100	2310.99	144664	2308.99
sanr200_0.7	18	27.65	12985	7.44
sanr200_0.9	42	11884.40	1733080	7932.39
sanr400_0.5	13	41.43	18084	21.63
sanr400_0.7	21	14585.70	5067837	40.76

Table B.6: *Alg. 20 + Domain Filtering +* $\mathcal{U}_8$

Instance	$\lvert C^* \rvert$	time	BB node	time best
brock200_1	21	73.99	30811	16.68
brock200_2	12	0.52	285	0.15
brock200_3	15	2.68	1356	0.01
brock200_4	17	10.09	4169	7.53
brock400_1	≥25	≥21600.00	5677245	16179.00
brock400_2	≥29	≥21600.00	4429451	8665.44
brock400_3	≥24	≥21600.00	5830338	53.22
brock400_4	≥25	≥21600.00	5153284	1239.64
brock800_1	≥21	≥21600.00	5862067	12967.00
brock800_2	≥20	≥21600.00	6381751	1646.71
brock800_3	≥20	≥21600.00	6112439	567.43
brock800_4	≥20	≥21600.00	6176011	183.07
c-fat200-1	12	0.01	5	0.01
c-fat200-2	24	0.01	5	0.01
c-fat200-5	58	0.01	5	0.01
c-fat500-1	14	0.05	5	0.05
c-fat500-10	126	0.07	5	0.07
c-fat500-2	26	0.07	5	0.07
c-fat500-5	64	0.03	5	0.03
hamming10-2	512	0.67	257	0.66
hamming10-4	**≥33**	**≥21600.00**	**4495000**	**17099.10**
hamming6-2	32	0.01	17	0.01
hamming6-4	4	0.01	25	0.01
hamming8-2	128	0.01	65	0.01
hamming8-4	16	5.44	570	0.03
johnson16-2-4	8	17.73	57912	0.01
johnson32-2-4	≥16	≥21600.00	70631873	0.11
johnson8-2-4	4	0.01	11	0.01
johnson8-4-4	14	0.01	17	0.01
keller4	11	3.65	1476	0.05
keller5	≥26	≥21600.00	2423520	19297.40
keller6	≥43	≥21600.00	1341291	879.56
MANN_a27	126	3046.67	31240	327.21

Instance	$\lvert C^* \rvert$	time	BB node	time best
MANN_a45	≥336	≥21600.00	24794	14973.50
MANN_a81	≥998	≥21600.00	9637	7482.24
MANN_a9	16	0.01	28	0.01
p_hat1000-1	10	35.28	13303	4.22
p_hat1000-2	≥44	≥21600.00	1341279	4794.00
p_hat1000-3	≥52	≥21600.00	1274139	17507.80
p_hat1500-1	12	380.57	101895	79.79
p_hat1500-2	≥52	≥21600.00	783463	2614.38
p_hat1500-3	≥58	≥21600.00	1148829	12700.10
p_hat300-1	8	0.36	212	0.22
p_hat300-2	25	3.61	438	2.58
p_hat300-3	36	482.81	71863	479.61
p_hat500-1	9	1.46	475	0.19
p_hat500-2	36	128.32	15101	128.30
p_hat500-3	≥49	≥21600.00	1218440	9000.27
p_hat700-1	11	5.14	815	4.96
p_hat700-2	44	1091.35	79390	1032.19
p_hat700-3	≥56	≥21600.00	943799	20078.10
san1000	15	1141.11	31477	1141.11
san200_0.7_1	30	0.89	268	0.89
san200_0.7_2	18	0.60	176	0.60
san200_0.9_1	70	24.54	4291	24.54
san200_0.9_2	60	600.46	72365	600.44
san200_0.9_3	44	71.55	11830	66.28
san400_0.5_1	13	3.64	739	3.64
san400_0.7_1	40	150.44	8576	150.44
san400_0.7_2	30	62.94	10129	62.94
san400_0.7_3	22	367.05	84602	367.05
san400_0.9_1	100	2033.07	120402	2031.89
sanr200_0.7	18	28.05	12246	7.57
sanr200_0.9	42	10263.50	1421500	6949.09
sanr400_0.5	13	44.22	17439	22.28
sanr400_0.7	21	14900.60	4783079	41.62

Table B.7: *Alg. 20 + Lower Bound Heuristics*

Instance	$\|C^*\|$	time	BB node	time best
brock200_1	21	42.15	38042992	6.98
brock200_2	12	0.08	54309	0.01
brock200_3	15	0.58	453265	0.01
brock200_4	17	2.47	2161899	1.75
brock400_1	≥25	≥21600.00	17509992712	11529.80
brock400_2	≥29	≥21600.00	13717772679	5983.55
brock400_3	≥24	≥21600.00	17849569194	31.30
brock400_4	≥33	≥21600.00	16095315583	20293.10
brock800_1	≥21	≥21600.00	14474127602	4156.91
brock800_2	≥20	≥21600.00	15934828100	464.66
brock800_3	≥21	≥21600.00	15528515213	18057.50
brock800_4	≥21	≥21600.00	15218342908	11858.00
c-fat200-1	**12**	**0.01**	**0**	**0.01**
c-fat200-2	**24**	**0.01**	**0**	**0.01**
c-fat200-5	58	1845.75	268435427	0.01
c-fat500-1	**14**	**0.01**	**0**	**0.01**
c-fat500-10	**126**	**0.01**	**0**	**0.01**
c-fat500-2	**26**	**0.01**	**0**	**0.01**
c-fat500-5	**64**	**0.01**	**0**	**0.01**
hamming10-2	**512**	**0.04**	**0**	**0.04**
hamming10-4	≥32	≥21600.00	26701433927	0.04
hamming6-2	**32**	**0.01**	**0**	**0.01**
hamming6-4	4	0.01	219	0.01
hamming8-2	**128**	**0.01**	**0**	**0.01**
hamming8-4	16	5.84	3742134	0.01
johnson16-2-4	8	2.05	4177628	0.01
johnson32-2-4	≥16	≥21600.00	39916006041	0.01
johnson8-2-4	4	0.01	88	0.01
johnson8-4-4	14	0.01	12536	0.01
keller4	11	1.05	1275235	0.01
keller5	≥24	≥21600.00	16696020061	4716.59
keller6	≥43	≥21600.00	13238117569	17796.50
MANN_a27	≥125	≥21600.00	10300735430	0.01

Instance	$\|C^*\|$	time	BB node	time best
MANN_a45	≥342	≥21600.00	3467302536	0.05
MANN_a81	≥1096	≥21600.00	972062172	0.39
MANN_a9	16	0.12	329210	0.01
p_hat1000-1	10	3.23	1800820	0.65
p_hat1000-2	≥42	≥21600.00	13536058039	17341.00
p_hat1000-3	≥57	≥21600.00	9165352377	0.04
p_hat1500-1	12	28.66	13825474	6.21
p_hat1500-2	≥54	≥21600.00	9513755020	0.11
p_hat1500-3	≥75	≥21600.00	6725356212	0.11
p_hat300-1	8	0.03	11602	0.03
p_hat300-2	25	1.63	1378757	0.88
p_hat300-3	36	2340.66	1958377489	2302.61
p_hat500-1	9	0.19	89900	0.08
p_hat500-2	36	382.84	290559086	382.56
p_hat500-3	≥44	≥21600.00	16480493753	21171.00
p_hat700-1	11	0.69	343852	0.68
p_hat700-2	≥41	≥21600.00	4241515523	1266.67
p_hat700-3	≥55	≥21600.00	10897648442	0.01
san1000	≥10	≥21600.00	8541223897	13781.40
san200_0.7_1	30	7569.35	14265131068	6432.11
san200_0.7_2	≥18	≥21600.00	26707614335	94.98
san200_0.9_1	≥48	≥21600.00	31846126072	100.23
san200_0.9_2	≥41	≥21600.00	36295036841	17862.90
san200_0.9_3	≥44	≥21600.00	15617446544	13812.50
san400_0.5_1	13	1407.06	1414712955	65.02
san400_0.7_1	≥22	≥21600.00	68294511591	4593.51
san400_0.7_2	≥17	≥21600.00	49132455738	11965.30
san400_0.7_3	≥22	≥21600.00	19318251314	20240.40
san400_0.9_1	≥49	≥21600.00	40400910354	230.02
sanr200_0.7	18	8.52	7985056	1.78
sanr200_0.9	≥40	≥21600.00	18830466170	10837.30
sanr400_0.5	13	5.69	4283902	2.90
sanr400_0.7	21	6871.77	5624332477	14.45

Table B.8: *Alg. 20 + Domain Filtering + Lower Bound Heuristics*

Instance	$\lvert C^* \rvert$	time	BB node	time best
brock200_1	21	55.66	3350276	9.17
brock200_2	12	0.17	3534	0.01
brock200_3	15	1.06	32169	0.01
brock200_4	17	3.92	167745	2.75
brock400_1	≥25	≥21600.00	802043069	15547.30
brock400_2	≥29	≥21600.00	578464091	7608.22
brock400_3	≥24	≥21600.00	642974556	33.85
brock400_4	≥25	≥21600.00	804732848	957.49
brock800_1	≥21	≥21600.00	438408850	6309.47
brock800_2	≥20	≥21600.00	483525994	711.95
brock800_3	≥20	≥21600.00	521912459	246.56
brock800_4	≥20	≥21600.00	507024368	76.96
c-fat200-1	**12**	**0.01**	**0**	**0.01**
c-fat200-2	**24**	**0.01**	**0**	**0.01**
c-fat200-5	**58**	**0.01**	**1**	**0.01**
c-fat500-1	**14**	**0.01**	**0**	**0.01**
c-fat500-10	**126**	**0.01**	**0**	**0.01**
c-fat500-2	**26**	**0.01**	**0**	**0.01**
c-fat500-5	**64**	**0.01**	**0**	**0.01**
hamming10-2	**512**	**0.04**	**0**	**0.04**
hamming10-4	≥32	≥21600.00	2102894087	0.04
hamming6-2	**32**	**0.01**	**0**	**0.01**
hamming6-4	4	0.01	45	0.01
hamming8-2	**128**	**0.01**	**0**	**0.01**
hamming8-4	16	8.06	252416	0.01
johnson16-2-4	8	3.69	904444	0.01
johnson32-2-4	≥16	≥21600.00	3905097678	0.01
johnson8-2-4	4	0.01	19	0.01
johnson8-4-4	14	0.01	1202	0.01
keller4	11	1.57	107084	0.01
keller5	≥24	≥21600.00	612982816	5156.58
keller6	≥43	≥21600.00	422477378	8293.01
MANN_a27	≥125	≥21600.00	6773339145	0.01
MANN_a45	≥342	≥21600.00	1584836740	0.03
MANN_a81	**≥1096**	**≥21600.00**	**509194829**	**0.35**
MANN_a9	16	0.13	92204	0.01
p_hat1000-1	10	10.13	81542	1.28
p_hat1000-2	≥41	≥21600.00	1123297576	11542.10
p_hat1000-3	≥57	≥21600.00	645646229	0.04
p_hat1500-1	12	97.35	454901	21.10
p_hat1500-2	≥54	≥21600.00	732702539	0.10
p_hat1500-3	≥75	≥21600.00	379259450	0.09
p_hat300-1	8	0.07	442	0.05
p_hat300-2	25	2.51	160836	1.40
p_hat300-3	36	2734.96	251602128	2693.99
p_hat500-1	9	0.53	4971	0.13
p_hat500-2	36	512.17	35122029	511.89
p_hat500-3	≥43	≥21600.00	1845348113	8498.38
p_hat700-1	11	2.13	20625	2.07
p_hat700-2	44	18369.00	1387618978	16213.60
p_hat700-3	≥55	≥21600.00	948549946	0.01
san1000	≥9	≥21600.00	161897446	403.41
san200_0.7_1	30	9238.11	3805796650	7685.40
san200_0.7_2	≥18	≥21600.00	5002801923	80.63
san200_0.9_1	≥48	≥21600.00	16362350308	60.72
san200_0.9_2	≥41	≥21600.00	3562188013	1103.38
san200_0.9_3	≥44	≥21600.00	1670536852	14220.30
san400_0.5_1	13	2252.10	155275555	102.37
san400_0.7_1	≥22	≥21600.00	17778266595	1551.15
san400_0.7_2	≥17	≥21600.00	8111510989	15057.30
san400_0.7_3	≥18	≥21600.00	2039144714	21269.30
san400_0.9_1	≥49	≥21600.00	6094343540	3.00
sanr200_0.7	18	12.36	665002	2.61
sanr200_0.9	≥40	≥21600.00	2194582435	10370.70
sanr400_0.5	13	12.52	251449	6.42
sanr400_0.7	21	9427.26	408188809	19.74

Table B.9: *Alg. 20 +* $\mathcal{U}_8$ *+ Lower Bound Heuristics*

Instance	$\|C^*\|$	time	BB node	time best
brock200_1	21	83.67	32765	16.19
brock200_2	12	0.49	289	0.13
brock200_3	15	3.15	1402	0.01
brock200_4	17	9.46	4373	6.86
brock400_1	≥25	≥21600.00	6566602	16035.30
brock400_2	≥29	≥21600.00	5024485	8736.53
brock400_3	≥24	≥21600.00	6629519	53.16
brock400_4	≥25	≥21600.00	5836585	1246.54
brock800_1	≥21	≥21600.00	6802768	12384.00
brock800_2	≥20	≥21600.00	7483625	1602.05
brock800_3	≥20	≥21600.00	7288740	544.57
brock800_4	≥20	≥21600.00	7351531	181.23
c-fat200-1	**12**	**0.01**	**0**	**0.01**
c-fat200-2	**24**	**0.01**	**0**	**0.01**
c-fat200-5	**58**	**0.01**	**1**	**0.01**
c-fat500-1	**14**	**0.01**	**0**	**0.01**
c-fat500-10	**126**	**0.01**	**0**	**0.01**
c-fat500-2	**26**	**0.01**	**0**	**0.01**
c-fat500-5	**64**	**0.01**	**0**	**0.01**
hamming10-2	**512**	**0.04**	**0**	**0.04**
hamming10-4	≥33	≥21600.00	5182900	17465.10
hamming6-2	**32**	**0.01**	**0**	**0.01**
hamming6-4	4	0.01	125	0.01
hamming8-2	**128**	**0.01**	**0**	**0.01**
hamming8-4	16	3.28	553	0.01
johnson16-2-4	8	11.27	123168	0.01
johnson32-2-4	≥16	≥21600.00	224623978	0.01
johnson8-2-4	4	0.01	52	0.01
johnson8-4-4	14	0.01	11	0.01
keller4	11	2.57	1556	0.03
keller5	≥26	≥21600.00	2837055	19125.70
keller6	≥43	≥21600.00	1602673	1220.17
MANN_a27	126	3545.72	30250	360.67

Instance	$\|C^*\|$	time	BB node	time best
MANN_a45	≥342	≥21600.00	9280	0.04
MANN_a81	≥1096	≥21600.00	344	0.40
MANN_a9	16	0.01	18	0.01
p_hat1000-1	10	27.34	14024	1.98
p_hat1000-2	≥44	≥21600.00	1391266	4590.68
p_hat1000-3	≥57	≥21600.00	725698	0.04
p_hat1500-1	12	290.96	104610	57.15
p_hat1500-2	≥54	≥21600.00	710845	0.11
p_hat1500-3	≥75	≥21600.00	293927	0.10
p_hat300-1	8	0.20	212	0.12
p_hat300-2	25	2.09	345	1.42
p_hat300-3	36	510.82	78958	507.28
p_hat500-1	9	1.21	458	0.16
p_hat500-2	36	126.73	15228	126.71
p_hat500-3	≥49	≥21600.00	1224756	9443.06
p_hat700-1	11	3.94	842	3.77
p_hat700-2	44	1137.19	85853	1071.19
p_hat700-3	≥56	≥21600.00	827848	14607.20
san1000	15	875.01	22118	875.00
san200_0.7_1	30	0.70	135	0.70
san200_0.7_2	18	0.50	102	0.50
san200_0.9_1	70	6.83	544	6.83
san200_0.9_2	60	659.15	82932	659.13
san200_0.9_3	44	74.23	12338	69.30
san400_0.5_1	13	2.49	403	2.48
san400_0.7_1	40	123.60	7219	123.60
san400_0.7_2	30	44.80	5894	44.79
san400_0.7_3	22	291.47	65118	291.46
san400_0.9_1	100	1984.76	100519	1982.67
sanr200_0.7	18	27.79	12930	7.49
sanr200_0.9	42	11804.90	1731560	8049.09
sanr400_0.5	13	41.48	18057	21.65
sanr400_0.7	21	14630.90	5067745	40.92

Table B.10: *Alg. 20 + Domain Filtering + $\mathcal{U}_8$+ Lower Bound Heuristics*

Instance	$\|C^*\|$	time	BB node	time best
brock200_1	21	83.81	30694	16.10
brock200_2	12	0.54	272	0.14
brock200_3	15	3.26	1356	0.01
brock200_4	17	10.02	4123	7.29
brock400_1	≥25	≥21600.00	6032026	16099.20
brock400_2	≥29	≥21600.00	4583263	8746.22
brock400_3	≥24	≥21600.00	6074490	52.50
brock400_4	≥25	≥21600.00	5309119	1245.55
brock800_1	≥21	≥21600.00	6108652	12905.80
brock800_2	≥20	≥21600.00	6699015	1662.42
brock800_3	≥20	≥21600.00	6504352	560.13
brock800_4	≥20	≥21600.00	6571649	190.12
c-fat200-1	**12**	**0.01**	**0**	**0.01**
c-fat200-2	**24**	**0.01**	**0**	**0.01**
c-fat200-5	**58**	**0.01**	**1**	**0.01**
c-fat500-1	**14**	**0.01**	**0**	**0.01**
c-fat500-10	**126**	**0.01**	**0**	**0.01**
c-fat500-2	**26**	**0.01**	**0**	**0.01**
c-fat500-5	**64**	**0.01**	**0**	**0.01**
hamming10-2	**512**	**0.04**	**0**	**0.03**
hamming10-4	≥33	≥21600.00	4692037	17301.70
hamming6-2	**32**	**0.01**	**0**	**0.01**
hamming6-4	4	0.01	22	0.01
hamming8-2	**128**	**0.01**	**0**	**0.01**
hamming8-4	16	3.75	556	0.01
johnson16-2-4	8	12.39	57881	0.01
johnson32-2-4	≥16	≥21600.00	72262348	0.01
johnson8-2-4	4	0.01	11	0.01
johnson8-4-4	14	0.01	11	0.01
keller4	11	2.66	1457	0.03
keller5	≥26	≥21600.00	2490605	18977.10
keller6	≥43	≥21600.00	1344053	796.79
MANN_a27	126	2984.11	30458	283.11

Instance	$\|C^*\|$	time	BB node	time best
MANN_a45	≥342	≥21600.00	11883	0.05
MANN_a81	≥1096	≥21600.00	462	0.43
MANN_a9	16	0.01	18	0.01
p_hat1000-1	10	32.57	13282	2.71
p_hat1000-2	≥44	≥21600.00	1263289	4680.78
p_hat1000-3	≥57	≥21600.00	658102	0.08
p_hat1500-1	12	341.15	101886	67.74
p_hat1500-2	≥54	≥21600.00	616341	0.11
p_hat1500-3	≥75	≥21600.00	266476	0.12
p_hat300-1	8	0.24	207	0.15
p_hat300-2	25	2.41	333	1.59
p_hat300-3	36	487.80	71384	484.66
p_hat500-1	9	1.46	462	0.18
p_hat500-2	36	127.90	14585	127.88
p_hat500-3	≥49	≥21600.00	1108105	9059.61
p_hat700-1	11	5.69	802	5.43
p_hat700-2	44	1111.40	76079	1048.25
p_hat700-3	≥56	≥21600.00	577935	14274.40
san1000	15	1476.19	31290	1476.17
san200_0.7_1	30	0.58	154	0.58
san200_0.7_2	18	0.51	107	0.51
san200_0.9_1	70	7.13	465	7.12
san200_0.9_2	60	651.72	71274	651.70
san200_0.9_3	44	86.09	11602	80.90
san400_0.5_1	13	4.95	694	4.89
san400_0.7_1	40	168.33	8449	168.31
san400_0.7_2	30	63.87	9062	63.86
san400_0.7_3	22	403.95	84522	403.95
san400_0.9_1	100	2011.49	85997	2010.11
sanr200_0.7	18	29.23	12198	8.69
sanr200_0.9	42	11082.70	1421874	7516.34
sanr400_0.5	13	45.72	17424	23.78
sanr400_0.7	21	15357.30	4783053	43.39

Table B.11: *Alg. 20 + Reordering*

Instance	$\lvert C^* \rvert$	time	BB node	time best
brock200_1	21	65.51	49899719	10.40
brock200_2	12	0.11	63604	0.03
brock200_3	15	0.83	538998	0.01
brock200_4	17	3.52	2615053	2.43
brock400_1	≥24	≥21600.00	15257627938	2647.41
brock400_2	≥29	≥21600.00	12580370063	9965.22
brock400_3	≥24	≥21600.00	15059690473	41.74
brock400_4	≥25	≥21600.00	13903077839	1291.18
brock800_1	≥21	≥21600.00	12784917881	6051.07
brock800_2	≥20	≥21600.00	14202981566	656.16
brock800_3	≥20	≥21600.00	13795314603	511.79
brock800_4	≥21	≥21600.00	13642131003	17512.80
c-fat200-1	12	0.01	110	0.01
c-fat200-2	24	0.01	1029	0.01
c-fat200-5	58	1844.93	268435599	1844.93
c-fat500-1	14	0.05	218	0.05
c-fat500-10	≥124	≥21600.00	5797086125	0.04
c-fat500-2	26	0.04	2019	0.04
c-fat500-5	64	6139.54	1050334958	6139.54
hamming10-2	≥512	≥21600.00	179480844	0.69
hamming10-4	≥32	≥21600.00	18984190137	0.42
hamming6-2	32	0.04	33963	0.01
hamming6-4	4	0.01	246	0.01
hamming8-2	≥128	≥21600.00	9104272282	0.01
hamming8-4	16	13.54	7376228	0.01
johnson16-2-4	8	3.03	5283608	0.01
johnson32-2-4	≥16	≥21600.00	36678118580	0.05
johnson8-2-4	4	0.01	120	0.01
johnson8-4-4	14	0.03	21362	0.01
keller4	11	1.67	1730662	0.01
keller5	≥24	≥21600.00	13816583527	8329.70
keller6	≥42	≥21600.00	14157491075	16329.10
MANN_a27	≥103	≥21600.00	42444002454	268.52
MANN_a45	≥173	≥21600.00	2908205213	12940.10
MANN_a81	≥303	≥21600.00	326992131	505.26
MANN_a9	16	0.37	914643	0.01
p_hat1000-1	10	3.54	1915766	0.58
p_hat1000-2	≥40	≥21600.00	13834984821	18592.50
p_hat1000-3	≥47	≥21600.00	13254199985	5961.59
p_hat1500-1	12	31.24	14911533	7.17
p_hat1500-2	≥45	≥21600.00	16486749297	18027.40
p_hat1500-3	≥53	≥21600.00	17135763325	19437.10
p_hat300-1	8	0.03	12580	0.03
p_hat300-2	25	2.72	2130446	1.70
p_hat300-3	36	5114.76	3803115648	5045.63
p_hat500-1	9	0.21	97060	0.08
p_hat500-2	36	760.18	517465338	744.70
p_hat500-3	≥43	≥21600.00	15547363258	17471.50
p_hat700-1	11	0.76	369843	0.74
p_hat700-2	≥43	≥21600.00	14317764838	19005.10
p_hat700-3	≥48	≥21600.00	16341797661	13361.10
san1000	≥10	≥21600.00	6684993940	15589.30
san200_0.7_1	30	9644.26	15173779791	7755.11
san200_0.7_2	≥18	≥21600.00	28710609315	125.18
san200_0.9_1	≥48	≥21600.00	28958805857	3984.80
san200_0.9_2	≥40	≥21600.00	29097836649	203.70
san200_0.9_3	≥41	≥21600.00	14428331418	16720.30
san400_0.5_1	13	2966.84	1792740152	68.64
san400_0.7_1	≥21	≥21600.00	51670924049	0.05
san400_0.7_2	≥17	≥21600.00	41949116241	13695.40
san400_0.7_3	≥18	≥21600.00	17835622461	19626.10
san400_0.9_1	≥51	≥21600.00	52202587417	0.04
sanr200_0.7	18	13.12	9990289	2.04
sanr200_0.9	≥38	≥21600.00	17545633580	14753.40
sanr400_0.5	13	7.18	4704750	3.21
sanr400_0.7	21	10454.50	7297779227	31.43

Instance	$\lvert C^* \rvert$	time	BB node	time best
brock200_1	21	38.66	2313467	6.05
brock200_2	12	0.17	3518	0.03
brock200_3	15	0.93	31515	0.01
brock200_4	17	3.31	132692	2.21
brock400_1	$\geq$27	$\geq$21600.00	1068560780	16278.60
brock400_2	**$\geq$29**	**$\geq$21600.00**	**704686356**	**4019.71**
brock400_3	$\geq$25	$\geq$21600.00	1132438178	17689.20
brock400_4	$\geq$33	$\geq$21600.00	814076134	13772.50
brock800_1	$\geq$21	$\geq$21600.00	648946316	4906.81
brock800_2	$\geq$20	$\geq$21600.00	838024469	125.29
brock800_3	$\geq$21	$\geq$21600.00	718383093	15806.50
brock800_4	$\geq$21	$\geq$21600.00	722796727	11363.30
c-fat200-1	12	0.01	5	0.01
c-fat200-2	24	0.01	5	0.01
c-fat200-5	58	0.01	5	0.01
c-fat500-1	14	0.06	5	0.06
c-fat500-10	126	0.06	5	0.06
c-fat500-2	26	0.06	5	0.06
c-fat500-5	64	0.08	5	0.08
hamming10-2	$\geq$177	$\geq$21600.00	2804846620	10695.90
hamming10-4	$\geq$28	$\geq$21600.00	1840652559	321.98
hamming6-2	32	0.01	827	0.01
hamming6-4	4	0.01	36	0.01
hamming8-2	$\geq$107	$\geq$21600.00	627098542	8880.29
hamming8-4	16	6.94	325951	0.03
johnson16-2-4	8	4.49	1176074	0.01
johnson32-2-4	$\geq$16	$\geq$21600.00	5251127543	0.05
johnson8-2-4	4	0.01	18	0.01
johnson8-4-4	14	0.01	1913	0.01
keller4	11	1.36	111284	0.01
keller5	$\geq$26	$\geq$21600.00	1723730978	77.08
keller6	$\geq$49	$\geq$21600.00	1278951020	13521.10
MANN_a27	$\geq$114	$\geq$21600.00	15175132797	29.80

Instance	$\lvert C^* \rvert$	time	BB node	time best
MANN_a45	$\geq$220	$\geq$21600.00	21302355103	7277.84
MANN_a81	$\geq$438	$\geq$21600.00	15014846089	66.25
MANN_a9	16	0.27	214424	0.01
p_hat1000-1	10	9.68	76995	1.06
p_hat1000-2	$\geq$41	$\geq$21600.00	1080580618	10416.30
p_hat1000-3	$\geq$46	$\geq$21600.00	1473328818	2035.68
p_hat1500-1	12	91.63	303225	22.57
p_hat1500-2	$\geq$46	$\geq$21600.00	1919302398	8323.22
p_hat1500-3	$\geq$53	$\geq$21600.00	1968529256	9462.90
p_hat300-1	8	0.07	406	0.05
p_hat300-2	25	2.56	161080	1.69
p_hat300-3	36	2102.16	189101562	2070.93
p_hat500-1	9	0.49	4954	0.11
p_hat500-2	36	436.28	30222902	427.51
p_hat500-3	$\geq$44	$\geq$21600.00	1724527301	11578.40
p_hat700-1	11	1.98	22538	1.94
p_hat700-2	44	11630.70	802206146	7551.13
p_hat700-3	$\geq$49	$\geq$21600.00	1511133978	17864.80
san1000	$\geq$10	$\geq$21600.00	34213044	0.51
san200_0.7_1	30	12139.10	3005157489	8189.47
san200_0.7_2	$\geq$18	$\geq$21600.00	3712593410	90.31
san200_0.9_1	$\geq$48	$\geq$21600.00	14174388128	0.22
san200_0.9_2	$\geq$50	$\geq$21600.00	8865107251	13181.50
san200_0.9_3	$\geq$44	$\geq$21600.00	3405968720	4177.24
san400_0.5_1	13	3257.63	87018003	103.91
san400_0.7_1	$\geq$22	$\geq$21600.00	22523343854	0.05
san400_0.7_2	$\geq$30	$\geq$21600.00	2809429031	466.56
san400_0.7_3	$\geq$17	$\geq$21600.00	2332887030	37.93
san400_0.9_1	$\geq$51	$\geq$21600.00	14561425978	4315.51
sanr200_0.7	18	9.53	549887	1.42
sanr200_0.9	$\geq$40	$\geq$21600.00	2124394795	3439.17
sanr400_0.5	13	10.65	263245	4.86
sanr400_0.7	21	6135.33	281397941	59.55

Table B.12: *Alg. 20 + Reordering + Domain Filtering*

Table B.13: *Alg. 20 + Reordering +* $\mathcal{U}_8$

Instance	$\|C^*\|$	time	BB node	time best
brock200_1	21	91.14	34628	17.75
brock200_2	12	0.54	349	0.17
brock200_3	15	3.35	1424	0.01
brock200_4	17	10.83	4672	7.85
brock400_1	≥24	≥21600.00	6771369	3125.74
brock400_2	≥29	≥21600.00	5310557	10337.80
brock400_3	≥24	≥21600.00	6596686	49.62
brock400_4	≥25	≥21600.00	5817543	1488.90
brock800_1	≥21	≥21600.00	6807032	14849.10
brock800_2	≥20	≥21600.00	7288864	1891.66
brock800_3	≥20	≥21600.00	7156255	1501.81
brock800_4	≥20	≥21600.00	7177430	193.44
c-fat200-1	12	0.01	34	0.01
c-fat200-2	24	0.01	70	0.01
c-fat200-5	58	0.06	172	0.06
c-fat500-1	14	0.05	40	0.05
c-fat500-10	126	0.69	376	0.69
c-fat500-2	26	0.06	76	0.06
c-fat500-5	64	0.15	190	0.15
hamming10-2	512	0.83	513	0.83
hamming10-4	≥32	≥21600.00	5491919	0.53
hamming6-2	32	0.01	33	0.01
hamming6-4	4	0.01	153	0.01
hamming8-2	128	0.03	129	0.03
hamming8-4	16	6.33	792	0.01
johnson16-2-4	8	7.94	150523	0.01
johnson32-2-4	≥16	≥21600.00	252894845	0.08
johnson8-2-4	4	0.01	65	0.01
johnson8-4-4	14	0.01	23	0.01
keller4	11	3.65	2181	0.03
keller5	≥26	≥21600.00	3228220	19937.40
keller6	≥43	≥21600.00	2958575	21370.90
MANN_a27	126	3171.52	25184	372.04

Instance	$\|C^*\|$	time	BB node	time best
MANN_a45	≥336	≥21600.00	26634	18642.10
MANN_a81	≥998	≥21600.00	33075	14108.50
MANN_a9	16	0.01	41	0.01
p_hat1000-1	10	28.69	14599	1.61
p_hat1000-2	≥44	≥21600.00	1432135	6687.41
p_hat1000-3	≥51	≥21600.00	1315718	15166.80
p_hat1500-1	12	302.30	106659	67.42
p_hat1500-2	≥52	≥21600.00	848691	2974.68
p_hat1500-3	≥58	≥21600.00	1275562	17717.10
p_hat300-1	8	0.20	225	0.12
p_hat300-2	25	2.63	724	1.99
p_hat300-3	36	590.09	91974	586.01
p_hat500-1	9	1.25	500	0.14
p_hat500-2	36	145.01	18886	143.75
p_hat500-3	≥49	≥21600.00	1304510	16032.90
p_hat700-1	11	4.15	964	3.99
p_hat700-2	44	1100.18	87090	863.46
p_hat700-3	≥55	≥21600.00	997373	16883.60
san1000	15	880.21	29848	880.21
san200_0.7_1	30	0.63	353	0.63
san200_0.7_2	18	0.52	273	0.52
san200_0.9_1	70	27.24	4969	27.24
san200_0.9_2	60	509.80	64745	509.44
san200_0.9_3	44	305.33	26767	79.88
san400_0.5_1	13	2.81	2348	2.81
san400_0.7_1	40	1803.00	271353	1803.00
san400_0.7_2	30	41.68	5625	41.68
san400_0.7_3	22	296.98	62370	296.98
san400_0.9_1	100	2699.95	173619	2667.88
sanr200_0.7	18	30.26	13663	6.15
sanr200_0.9	42	13876.00	2073152	9291.64
sanr400_0.5	13	43.43	18444	20.55
sanr400_0.7	21	16335.00	5543348	67.83

Table B.14: *Alg. 20 + Reordering + Domain Filtering +* $\mathcal{U}_8$

Instance	$\lvert C^*\rvert$	time	BB node	time best
brock200_1	21	72.66	26428	13.45
brock200_2	12	0.56	302	0.16
brock200_3	15	3.34	1406	0.01
brock200_4	17	8.91	3308	6.03
brock400_1	≥27	≥21600.00	5276922	20561.50
brock400_2	≥29	≥21600.00	4038248	5937.40
brock400_3	≥24	≥21600.00	5609626	236.52
brock400_4	33	18115.70	3555795	14776.80
brock800_1	≥21	≥21600.00	5193115	11232.40
brock800_2	≥20	≥21600.00	6220132	391.73
brock800_3	≥20	≥21600.00	5756986	359.41
brock800_4	≥20	≥21600.00	6092696	1144.56
c-fat200-1	12	0.01	5	0.01
c-fat200-2	24	0.01	5	0.01
c-fat200-5	58	0.01	5	0.01
c-fat500-1	14	0.06	5	0.06
c-fat500-10	126	0.07	5	0.07
c-fat500-2	26	0.06	5	0.05
c-fat500-5	64	0.07	5	0.07
hamming10-2	≥206	≥21600.00	759969	19311.10
hamming10-4	≥28	≥21600.00	3298069	72.94
hamming6-2	32	0.01	26	0.01
hamming6-4	4	0.01	20	0.01
hamming8-2	128	5.21	206	5.21
hamming8-4	16	6.12	1199	0.07
johnson16-2-4	8	7.22	27231	0.01
johnson32-2-4	≥16	≥21600.00	42369132	0.08
johnson8-2-4	4	0.01	9	0.01
johnson8-4-4	14	0.04	32	0.01
keller4	11	1.99	934	0.01
keller5	**≥27**	**≥21600.00**	**1419794**	**13055.40**
keller6	≥53	≥21600.00	162325	3877.37
MANN_a27	126	2668.57	22895	286.52

Instance	$\lvert C^*\rvert$	time	BB node	time best
MANN_a45	≥336	≥21600.00	24116	16384.60
MANN_a81	≥998	≥21600.00	9378	6722.11
MANN_a9	16	0.01	27	0.01
p_hat1000-1	10	30.40	11975	1.86
p_hat1000-2	≥44	≥21600.00	1269513	5762.59
p_hat1000-3	≥52	≥21600.00	1132211	11534.30
p_hat1500-1	12	338.35	110362	76.45
p_hat1500-2	≥52	≥21600.00	726728	2216.56
p_hat1500-3	≥59	≥21600.00	874841	17595.50
p_hat300-1	8	0.22	211	0.14
p_hat300-2	25	2.60	461	1.97
p_hat300-3	36	477.52	67826	474.19
p_hat500-1	9	1.45	434	0.16
p_hat500-2	36	132.98	15529	131.87
p_hat500-3	≥49	≥21600.00	1123599	13108.90
p_hat700-1	11	4.80	659	4.63
p_hat700-2	44	947.95	63467	760.73
p_hat700-3	≥56	≥21600.00	843109	17171.80
san1000	**15**	**40.84**	**3959**	**40.84**
san200_0.7_1	30	0.28	110	0.28
san200_0.7_2	18	0.59	170	0.59
san200_0.9_1	70	4.28	472	4.28
san200_0.9_2	60	92.10	8273	91.41
san200_0.9_3	44	317.33	22622	44.96
san400_0.5_1	13	0.88	289	0.88
san400_0.7_1	40	19.87	1846	19.87
san400_0.7_2	**30**	**0.61**	**230**	**0.61**
san400_0.7_3	**22**	**69.50**	**31739**	**69.50**
san400_0.9_1	100	2879.44	140049	2863.05
sanr200_0.7	18	23.09	8716	4.84
sanr200_0.9	42	7218.59	889090	4705.64
sanr400_0.5	13	39.87	13893	16.77
sanr400_0.7	21	12116.70	3945222	142.56

Table B.15: *Alg. 20 + Reordering + Lower Bound Heuristics*

Instance	$\|C^*\|$	time	BB node	time best
brock200_1	21	65.62	49898591	10.42
brock200_2	12	0.09	63601	0.01
brock200_3	15	0.83	538998	0.01
brock200_4	17	3.55	2615051	2.45
brock400_1	≥24	≥21600.00	14490277876	2727.23
brock400_2	≥29	≥21600.00	12406507816	10024.70
brock400_3	≥24	≥21600.00	14167561520	43.71
brock400_4	≥25	≥21600.00	13252428065	1306.64
brock800_1	≥21	≥21600.00	11924335269	6186.32
brock800_2	≥20	≥21600.00	12770853038	661.74
brock800_3	≥20	≥21600.00	12614398127	516.45
brock800_4	≥21	≥21600.00	12759647069	18007.20
c-fat200-1	**12**	**0.01**	**0**	**0.01**
c-fat200-2	**24**	**0.01**	**0**	**0.01**
c-fat200-5	58	1610.14	268435427	0.01
c-fat500-1	**14**	**0.01**	**0**	**0.01**
c-fat500-10	**126**	**0.01**	**0**	**0.01**
c-fat500-2	**26**	**0.01**	**0**	**0.01**
c-fat500-5	**64**	**0.01**	**0**	**0.01**
hamming10-2	**512**	**0.04**	**0**	**0.04**
hamming10-4	≥32	≥21600.00	18303870490	0.04
hamming6-2	**32**	**0.01**	**0**	**0.01**
hamming6-4	4	0.01	244	0.01
hamming8-2	**128**	**0.01**	**0**	**0.01**
hamming8-4	16	13.67	7376219	0.01
johnson16-2-4	8	3.12	5283606	0.01
johnson32-2-4	≥16	≥21600.00	34855051398	0.01
johnson8-2-4	4	0.01	118	0.01
johnson8-4-4	14	0.03	21354	0.01
keller4	11	1.65	1730661	0.01
keller5	≥24	≥21600.00	12915872520	8441.08
keller6	≥42	≥21600.00	12874864511	16587.20
MANN_a27	≥125	≥21600.00	11643988970	0.01

Instance	$\|C^*\|$	time	BB node	time best
MANN_a45	≥342	≥21600.00	161508694	0.04
MANN_a81	≥1096	≥21600.00	38270592	0.44
MANN_a9	16	0.37	914618	0.01
p_hat1000-1	10	3.58	1915600	0.59
p_hat1000-2	≥41	≥21600.00	13065534082	18375.60
p_hat1000-3	≥57	≥21600.00	8160533842	0.05
p_hat1500-1	12	33.01	14911432	7.46
p_hat1500-2	≥54	≥21600.00	9656215811	0.13
p_hat1500-3	≥75	≥21600.00	5710919557	0.09
p_hat300-1	8	0.04	12547	0.03
p_hat300-2	25	2.64	1875236	1.57
p_hat300-3	36	5130.16	3799786469	5060.56
p_hat500-1	9	0.21	97051	0.08
p_hat500-2	36	765.82	514123160	750.23
p_hat500-3	≥43	≥21600.00	14493361208	10507.70
p_hat700-1	11	0.77	369836	0.75
p_hat700-2	≥43	≥21600.00	13757000221	19388.70
p_hat700-3	≥55	≥21600.00	10018477007	0.03
san1000	≥10	≥21600.00	6778283814	15857.60
san200_0.7_1	30	9612.04	15173779790	7732.43
san200_0.7_2	≥18	≥21600.00	28862178896	125.52
san200_0.9_1	≥48	≥21600.00	28704419610	188.12
san200_0.9_2	≥40	≥21600.00	29184881874	149.41
san200_0.9_3	≥41	≥21600.00	14529190629	16681.00
san400_0.5_1	13	2959.04	1792740016	67.86
san400_0.7_1	≥21	≥21600.00	52122147749	0.01
san400_0.7_2	≥17	≥21600.00	42222707651	13552.50
san400_0.7_3	≥18	≥21600.00	18086248245	19524.60
san400_0.9_1	≥51	≥21600.00	53335357925	0.04
sanr200_0.7	18	12.87	9990266	1.99
sanr200_0.9	≥38	≥21600.00	17874776383	14637.60
sanr400_0.5	13	7.10	4704664	3.16
sanr400_0.7	21	10424.60	7297779208	31.15

Table B.16: *Alg. 20 + Reordering + Domain Filtering + Lower Bound Heuristics*

Instance	$\lvert C^*\rvert$	time	BB node	time best
brock200_1	21	38.48	2313406	6.04
brock200_2	12	0.18	3513	0.04
brock200_3	15	0.93	31506	0.01
brock200_4	17	3.31	132683	2.21
brock400_1	≥27	≥21600.00	1047423133	16374.90
brock400_2	≥29	≥21600.00	739862823	4016.16
brock400_3	≥25	≥21600.00	1110106412	17712.80
brock400_4	≥33	≥21600.00	777584470	13970.40
brock800_1	≥21	≥21600.00	594675470	4962.09
brock800_2	≥20	≥21600.00	841028383	125.13
brock800_3	≥21	≥21600.00	642964603	15813.70
brock800_4	≥21	≥21600.00	701700371	11533.30
c-fat200-1	**12**	**0.01**	**0**	**0.01**
c-fat200-2	**24**	**0.01**	**0**	**0.01**
c-fat200-5	**58**	**0.01**	**1**	**0.01**
c-fat500-1	**14**	**0.01**	**0**	**0.01**
c-fat500-10	**126**	**0.01**	**0**	**0.01**
c-fat500-2	**26**	**0.01**	**0**	**0.01**
c-fat500-5	**64**	**0.01**	**0**	**0.01**
hamming10-2	**512**	**0.05**	**0**	**0.04**
hamming10-4	≥32	≥21600.00	1020866809	0.04
hamming6-2	**32**	**0.01**	**0**	**0.01**
hamming6-4	4	0.01	33	0.01
hamming8-2	**128**	**0.01**	**0**	**0.01**
hamming8-4	16	6.89	325009	0.01
johnson16-2-4	8	4.48	1176072	0.01
johnson32-2-4	≥16	≥21600.00	5329632229	0.01
johnson8-2-4	4	0.01	16	0.01
johnson8-4-4	14	0.01	1583	0.01
keller4	11	1.35	111281	0.01
keller5	≥26	≥21600.00	1732864000	77.07
keller6	≥49	≥21600.00	1288361166	13549.00
MANN_a27	≥125	≥21600.00	9041109994	0.01

Instance	$\lvert C^*\rvert$	time	BB node	time best
MANN_a45	≥342	≥21600.00	2543670765	0.04
MANN_a81	≥1096	≥21600.00	131373252	0.40
MANN_a9	16	0.26	214418	0.01
p_hat1000-1	10	9.58	76968	1.03
p_hat1000-2	≥41	≥21600.00	1075709310	9191.15
p_hat1000-3	≥57	≥21600.00	686310486	0.04
p_hat1500-1	12	89.99	303215	22.06
p_hat1500-2	≥54	≥21600.00	573466783	0.11
p_hat1500-3	≥75	≥21600.00	115546349	0.11
p_hat300-1	8	0.07	396	0.05
p_hat300-2	25	2.36	141798	1.50
p_hat300-3	36	2089.43	188956559	2058.52
p_hat500-1	9	0.47	4949	0.10
p_hat500-2	36	432.19	30040329	423.45
p_hat500-3	≥43	≥21600.00	579118281	1552.59
p_hat700-1	11	1.97	22531	1.93
p_hat700-2	44	11511.00	797879323	7454.47
p_hat700-3	≥55	≥21600.00	255147097	0.03
san1000	≥10	≥21600.00	9715727	0.53
san200_0.7_1	30	12273.60	3005157458	8311.43
san200_0.7_2	≥18	≥21600.00	3699741669	90.64
san200_0.9_1	≥48	≥21600.00	14109733959	0.05
san200_0.9_2	≥50	≥21600.00	8644288228	13274.60
san200_0.9_3	≥44	≥21600.00	3364353803	4190.53
san400_0.5_1	13	3268.03	87018003	104.32
san400_0.7_1	≥22	≥21600.00	22309757207	0.05
san400_0.7_2	≥30	≥21600.00	2792819630	467.83
san400_0.7_3	≥17	≥21600.00	2256258217	38.98
san400_0.9_1	≥51	≥21600.00	14466752588	4187.63
sanr200_0.7	18	9.47	549793	1.40
sanr200_0.9	≥40	≥21600.00	2164491186	3444.67
sanr400_0.5	13	10.63	263236	4.88
sanr400_0.7	21	6102.07	281397830	59.25

Table B.17: *Alg. 20 + Reordering + $\mathcal{U}_8$ + Lower Bound Heuristics*

Instance	$\|C^*\|$	time	BB node	time best
brock200_1	21	101.91	34433	19.88
brock200_2	12	0.61	304	0.19
brock200_3	15	3.79	1443	0.01
brock200_4	17	11.99	4660	8.78
brock400_1	≥24	≥21600.00	6437702	3177.50
brock400_2	≥29	≥21600.00	5169019	10605.70
brock400_3	≥24	≥21600.00	6132721	50.07
brock400_4	≥25	≥21600.00	5372791	1517.32
brock800_1	≥21	≥21600.00	6376737	15218.20
brock800_2	≥20	≥21600.00	7063522	2034.61
brock800_3	≥20	≥21600.00	6865329	1513.72
brock800_4	≥20	≥21600.00	6805505	197.59
c-fat200-1	**12**	**0.01**	**0**	**0.01**
c-fat200-2	**24**	**0.01**	**0**	**0.01**
c-fat200-5	**58**	**0.01**	**1**	**0.01**
c-fat500-1	**14**	**0.01**	**0**	**0.01**
c-fat500-10	**126**	**0.01**	**0**	**0.01**
c-fat500-2	**26**	**0.01**	**0**	**0.01**
c-fat500-5	**64**	**0.01**	**0**	**0.01**
hamming10-2	**512**	**0.03**	**0**	**0.03**
hamming10-4	≥32	≥21600.00	5274425	0.04
hamming6-2	**32**	**0.01**	**0**	**0.01**
hamming6-4	4	0.01	151	0.01
hamming8-2	**128**	**0.01**	**0**	**0.01**
hamming8-4	16	7.06	780	0.01
johnson16-2-4	8	8.88	151187	0.01
johnson32-2-4	≥16	≥21600.00	249525359	0.01
johnson8-2-4	4	0.01	63	0.01
johnson8-4-4	**14**	**0.01**	**9**	**0.01**
keller4	11	4.04	2137	0.04
keller5	≥26	≥21600.00	3179801	20082.10
keller6	≥42	≥21600.00	2676541	4068.44
MANN_a27	126	3215.23	23600	355.40

Instance	$\|C^*\|$	time	BB node	time best
MANN_a45	**≥342**	**≥21600.00**	**8657**	**0.04**
MANN_a81	≥1096	≥21600.00	354	0.41
MANN_a9	**16**	**0.01**	**13**	**0.01**
p_hat1000-1	10	32.89	14555	1.82
p_hat1000-2	≥44	≥21600.00	1220347	6755.36
p_hat1000-3	≥57	≥21600.00	615050	0.06
p_hat1500-1	12	349.32	106611	77.79
p_hat1500-2	≥54	≥21600.00	596413	0.11
p_hat1500-3	≥75	≥21600.00	239075	0.12
p_hat300-1	8	0.23	195	0.14
p_hat300-2	25	2.61	397	1.87
p_hat300-3	36	670.13	91129	665.56
p_hat500-1	9	1.42	464	0.16
p_hat500-2	36	162.02	17853	160.61
p_hat500-3	≥49	≥21600.00	1178875	16212.30
p_hat700-1	11	4.76	920	4.58
p_hat700-2	44	1231.92	82805	961.12
p_hat700-3	≥56	≥21600.00	817371	17204.10
san1000	15	949.91	16632	949.89
san200_0.7_1	30	0.62	116	0.62
san200_0.7_2	18	0.48	103	0.48
san200_0.9_1	70	6.78	534	6.78
san200_0.9_2	60	516.05	63344	515.62
san200_0.9_3	44	307.38	25862	80.80
san400_0.5_1	13	2.86	409	2.85
san400_0.7_1	40	1848.61	269278	1848.61
san400_0.7_2	30	42.29	5155	42.27
san400_0.7_3	22	301.67	62033	301.67
san400_0.9_1	100	2768.34	171221	2735.33
sanr200_0.7	18	30.55	13650	6.33
sanr200_0.9	42	14301.40	2070622	9581.30
sanr400_0.5	13	43.49	18403	20.62
sanr400_0.7	21	17177.00	5543268	68.43

Instance	$\|C^*\|$	time	BB node	time best
brock200_1	21	73.67	26370	13.63
brock200_2	12	0.57	292	0.17
brock200_3	15	3.41	1408	0.01
brock200_4	17	9.01	3284	6.09
brock400_1	≥27	≥21600.00	5453784	20564.90
brock400_2	≥29	≥21600.00	4051058	5967.43
brock400_3	≥24	≥21600.00	5624812	238.21
brock400_4	**33**	**18159.10**	**3555730**	**14828.00**
brock800_1	≥21	≥21600.00	5343415	11248.50
brock800_2	≥20	≥21600.00	6253697	392.92
brock800_3	≥20	≥21600.00	5786058	360.20
brock800_4	≥20	≥21600.00	6119490	1147.72
c-fat200-1	**12**	**0.01**	**0**	**0.01**
c-fat200-2	**24**	**0.01**	**0**	**0.01**
c-fat200-5	**58**	**0.01**	**1**	**0.01**
c-fat500-1	**14**	**0.01**	**0**	**0.01**
c-fat500-10	**126**	**0.01**	**0**	**0.01**
c-fat500-2	**26**	**0.01**	**0**	**0.01**
c-fat500-5	**64**	**0.01**	**0**	**0.01**
hamming10-2	**512**	**0.05**	**0**	**0.05**
hamming10-4	≥32	≥21600.00	2167384	0.05
hamming6-2	**32**	**0.01**	**0**	**0.01**
hamming6-4	**4**	**0.01**	**17**	**0.01**
hamming8-2	**128**	**0.01**	**0**	**0.01**
hamming8-4	16	6.15	1147	0.01
johnson16-2-4	8	7.32	27155	0.01
johnson32-2-4	**≥16**	**≥21600.00**	**42409426**	**0.01**
johnson8-2-4	**4**	**0.01**	**7**	**0.01**
johnson8-4-4	14	0.03	12	0.01
keller4	11	2.01	926	0.01
keller5	≥27	≥21600.00	1425874	13052.20
keller6	**≥53**	**≥21600.00**	**161322**	**3890.13**
MANN_a27	126	2642.58	22057	266.29

Instance	$\|C^*\|$	time	BB node	time best
MANN_a45	≥342	≥21600.00	10980	0.04
MANN_a81	≥1096	≥21600.00	478	0.39
MANN_a9	16	0.01	17	0.01
p_hat1000-1	10	30.50	11947	1.87
p_hat1000-2	≥44	≥21600.00	1189604	5798.74
p_hat1000-3	≥57	≥21600.00	627630	0.05
p_hat1500-1	12	343.22	110341	77.94
p_hat1500-2	≥54	≥21600.00	576100	0.10
p_hat1500-3	≥75	≥21600.00	245262	0.14
p_hat300-1	8	0.22	194	0.14
p_hat300-2	25	2.32	345	1.70
p_hat300-3	36	484.77	67445	481.38
p_hat500-1	9	1.44	422	0.17
p_hat500-2	36	133.47	14933	132.37
p_hat500-3	≥49	≥21600.00	1074573	13236.00
p_hat700-1	11	4.85	640	4.68
p_hat700-2	44	944.18	60713	753.65
p_hat700-3	≥56	≥21600.00	674314	12134.00
san1000	15	43.24	3919	43.24
san200_0.7_1	**30**	**0.30**	**61**	**0.28**
san200_0.7_2	18	0.62	124	0.62
san200_0.9_1	70	4.35	289	4.35
san200_0.9_2	60	97.01	8071	96.34
san200_0.9_3	44	334.02	22297	47.19
san400_0.5_1	**13**	**0.83**	**288**	**0.83**
san400_0.7_1	40	21.04	1751	21.01
san400_0.7_2	30	0.65	160	0.65
san400_0.7_3	22	74.18	31701	74.18
san400_0.9_1	100	2904.94	137044	2891.18
sanr200_0.7	18	23.38	8656	4.92
sanr200_0.9	42	7366.73	888517	4794.17
sanr400_0.5	13	40.25	13921	16.95
sanr400_0.7	21	12282.50	3945125	145.01

Table B.18: *Alg. 20 + Reordering + Domain Filtering +* $\mathcal{U}_8$ *+ Lower Bound Heuristics*

Table B.19: *Alg. 21*

Instance	$\|C^*\|$	time	BB node	time best
brock200_1	21	40.99	40763864	39.03
brock200_2	12	0.08	47946	0.08
brock200_3	15	0.56	568162	0.56
brock200_4	**17**	**1.26**	**1058449**	**0.74**
brock400_1	$\geq$25	$\geq$21600.00	16744343445	1205.77
brock400_2	$\geq$25	$\geq$21600.00	17624720710	17604.70
brock400_3	$\geq$24	$\geq$21600.00	17960143939	57.82
brock400_4	$\geq$25	$\geq$21600.00	16569459718	3120.70
brock800_1	$\geq$21	$\geq$21600.00	15763475452	19926.40
brock800_2	**$\geq$21**	**$\geq$21600.00**	**15364807482**	**11196.20**
brock800_3	**$\geq$22**	**$\geq$21600.00**	**12984394338**	**1519.60**
brock800_4	$\geq$20	$\geq$21600.00	16035660281	1682.35
c-fat200-1	12	0.01	266	0.01
c-fat200-2	24	0.01	476	0.01
c-fat200-5	58	1822.34	268437253	0.01
c-fat500-1	14	0.05	591	0.05
c-fat500-10	126	0.12	8375	0.07
c-fat500-2	26	0.05	825	0.05
c-fat500-5	64	0.07	2516	0.06
hamming10-2	512	6.19	131840	2.23
hamming10-4	$\geq$32	$\geq$21600.00	32259594986	0.47
hamming6-2	32	0.01	560	0.01
hamming6-4	4	0.01	230	0.01
hamming8-2	128	0.07	8384	0.03
hamming8-4	16	5.80	3742273	0.01
johnson16-2-4	8	2.09	4177665	0.01
johnson32-2-4	$\geq$16	$\geq$21600.00	46937027805	0.06
johnson8-2-4	4	0.01	99	0.01
johnson8-4-4	14	0.01	8416	0.01
keller4	**11**	**0.63**	**1003121**	**0.39**
keller5	$\geq$20	$\geq$21600.00	34083725546	5854.36
keller6	$\geq$31	$\geq$21600.00	40974572043	13.46
MANN_a27	$\geq$117	$\geq$21600.00	8925632960	0.04

Instance	$\|C^*\|$	time	BB node	time best
MANN_a45	$\geq$330	$\geq$21600.00	3033506091	0.90
MANN_a81	$\geq$1080	$\geq$21600.00	907536555	33.57
MANN_a9	16	0.08	274543	0.08
p_hat1000-1	10	3.17	1686442	0.45
p_hat1000-2	$\geq$46	$\geq$21600.00	15833817626	565.30
p_hat1000-3	$\geq$60	$\geq$21600.00	19385985248	3082.38
p_hat1500-1	12	27.54	15753173	26.98
p_hat1500-2	$\geq$61	$\geq$21600.00	16053867267	1.57
p_hat1500-3	$\geq$85	$\geq$21600.00	13435662980	630.48
p_hat300-1	8	0.04	8428	0.01
p_hat300-2	25	1.23	944950	0.03
p_hat300-3	36	1071.48	774525947	9.62
p_hat500-1	9	0.20	83943	0.07
p_hat500-2	36	159.23	100476737	0.39
p_hat500-3	$\geq$50	$\geq$21600.00	15316807664	6671.02
p_hat700-1	11	0.59	187935	0.19
p_hat700-2	44	7857.52	5022334663	356.73
p_hat700-3	$\geq$62	$\geq$21600.00	14841334382	9158.72
san1000	$\geq$8	$\geq$21600.00	40251583444	0.46
san200_0.7_1	$\geq$15	$\geq$21600.00	72167962008	0.01
san200_0.7_2	$\geq$12	$\geq$21600.00	73596061385	0.01
san200_0.9_1	$\geq$45	$\geq$21600.00	26762063204	0.01
san200_0.9_2	$\geq$35	$\geq$21600.00	36870465389	0.01
san200_0.9_3	$\geq$24	$\geq$21600.00	54510972529	0.01
san400_0.5_1	13	3257.59	4714133041	3257.58
san400_0.7_1	$\geq$20	$\geq$21600.00	63144249943	0.04
san400_0.7_2	$\geq$15	$\geq$21600.00	73952176568	0.04
san400_0.7_3	$\geq$12	$\geq$21600.00	76892809612	0.04
san400_0.9_1	$\geq$50	$\geq$21600.00	24266146323	0.03
sanr200_0.7	18	7.37	6506241	0.33
sanr200_0.9	$\geq$41	$\geq$21600.00	17875107504	6838.80
sanr400_0.5	**13**	**3.80**	**2902155**	**2.67**
sanr400_0.7	21	6432.37	5226950345	110.21

Instance	$\lvert C^*\rvert$	time	BB node	time best
brock200_1	21	52.65	3809437	49.99
brock200_2	12	0.13	3088	0.13
brock200_3	15	0.92	47432	0.92
brock200_4	17	1.99	82857	1.15
brock400_1	≥25	≥21600.00	1047235395	1444.47
brock400_2	≥25	≥21600.00	1185423407	21303.80
brock400_3	≥24	≥21600.00	1191320924	68.33
brock400_4	≥25	≥21600.00	1019550053	3801.54
brock800_1	≥20	≥21600.00	694985151	1830.39
brock800_2	≥21	≥21600.00	703074375	16067.30
brock800_3	≥22	≥21600.00	499594513	2190.91
brock800_4	≥20	≥21600.00	709682147	2368.43
c-fat200-1	12	0.01	211	0.01
c-fat200-2	24	0.01	223	0.01
c-fat200-5	58	0.01	257	0.01
c-fat500-1	14	0.05	513	0.05
c-fat500-10	126	0.11	625	0.07
c-fat500-2	26	0.05	525	0.05
c-fat500-5	64	0.08	563	0.06
hamming10-2	512	5.16	1535	1.24
hamming10-4	≥32	≥21600.00	5396081984	0.43
hamming6-2	32	0.01	95	0.01
hamming6-4	4	0.01	67	0.01
hamming8-2	128	0.07	383	0.01
hamming8-4	16	7.85	252454	0.01
johnson16-2-4	8	3.63	904465	0.01
johnson32-2-4	≥16	≥21600.00	7962372028	0.06
johnson8-2-4	4	0.01	31	0.01
johnson8-4-4	14	0.01	1084	0.01
keller4	11	0.92	133911	0.55
keller5	≥20	≥21600.00	5207232560	5602.99
keller6	≥31	≥21600.00	23153877676	13.26
MANN_a27	≥117	≥21600.00	23181265259	0.03
MANN_a45	≥330	≥21600.00	22952138384	0.67
MANN_a81	≥1080	≥21600.00	23566687196	22.86
MANN_a9	16	0.13	138787	0.13
p_hat1000-1	10	8.45	75156	0.47
p_hat1000-2	≥46	≥21600.00	2235855371	522.98
p_hat1000-3	≥61	≥21600.00	8438845622	16736.60
p_hat1500-1	12	84.23	703941	82.78
p_hat1500-2	≥62	≥21600.00	4575631546	17992.60
p_hat1500-3	≥85	≥21600.00	11872713614	171.21
p_hat300-1	8	0.05	363	0.01
p_hat300-2	25	1.90	107966	0.04
p_hat300-3	36	1365.33	95821409	10.16
p_hat500-1	9	0.40	4821	0.07
p_hat500-2	36	225.26	11681596	0.43
p_hat500-3	≥50	≥21600.00	2396804695	6468.78
p_hat700-1	11	1.29	10591	0.21
p_hat700-2	44	9758.65	628055012	367.07
p_hat700-3	≥62	≥21600.00	3073251854	7635.24
san1000	≥8	≥21600.00	3826623750	0.47
san200_0.7_1	≥15	≥21600.00	23884042229	0.01
san200_0.7_2	≥12	≥21600.00	21011103362	0.01
san200_0.9_1	≥45	≥21600.00	21781906220	0.01
san200_0.9_2	≥35	≥21600.00	22185945440	0.01
san200_0.9_3	≥24	≥21600.00	22523065742	0.01
san400_0.5_1	13	4876.80	450539488	4876.79
san400_0.7_1	≥20	≥21600.00	22963713648	0.05
san400_0.7_2	≥15	≥21600.00	23623596407	0.05
san400_0.7_3	≥12	≥21600.00	21108189660	0.04
san400_0.9_1	≥50	≥21600.00	22766607890	0.04
sanr200_0.7	18	10.35	530953	0.45
sanr200_0.9	≥41	≥21600.00	2160419473	6823.89
sanr400_0.5	13	7.89	178928	5.38
sanr400_0.7	21	8831.31	376886905	141.77

Table B.20: *Alg. 21 + Domain Filtering*

Table B.21: *Alg. 21* + $\mathcal{U}_8$

Instance	$\lvert C^*\rvert$	time	BB node	time best
brock200_1	21	91.26	42767	87.42
brock200_2	12	0.24	377	0.24
brock200_3	15	3.49	2839	3.49
brock200_4	17	4.63	2457	3.02
brock400_1	≥25	≥21600.00	5998687	1649.73
brock400_2	≥25	≥21600.00	6710250	20981.70
brock400_3	≥24	≥21600.00	6745388	68.26
brock400_4	≥25	≥21600.00	5902546	3721.87
brock800_1	≥20	≥21600.00	7531481	3743.90
brock800_2	≥20	≥21600.00	7516642	885.72
brock800_3	≥22	≥21600.00	5747476	4013.31
brock800_4	≥20	≥21600.00	7490461	4766.56
c-fat200-1	12	0.01	266	0.01
c-fat200-2	24	0.03	476	0.03
c-fat200-5	58	0.53	1853	0.51
c-fat500-1	14	0.06	591	0.05
c-fat500-10	126	9.65	8375	9.61
c-fat500-2	26	0.08	825	0.07
c-fat500-5	64	0.80	2516	0.78
hamming10-2	512	2425.72	131840	2421.76
hamming10-4	≥32	≥21600.00	5939826	0.49
hamming6-2	32	0.06	560	0.06
hamming6-4	4	0.01	230	0.01
hamming8-2	128	10.29	8384	10.24
hamming8-4	**16**	**2.75**	**862**	**0.01**
johnson16-2-4	8	11.10	122781	0.01
johnson32-2-4	≥16	≥21600.00	15120429	0.06
johnson8-2-4	4	0.01	99	0.01
johnson8-4-4	14	0.01	164	0.01
keller4	11	2.01	2947	0.94
keller5	≥20	≥21600.00	13132294	1846.73
keller6	≥39	≥21600.00	1306103	1744.38
MANN_a27	**126**	**887.82**	**26396**	**167.96**

Instance	$\lvert C^*\rvert$	time	BB node	time best
MANN_a45	≥342	≥21600.00	107901	2032.58
MANN_a81	≥848	≥21600.00	360102	21224.50
MANN_a9	16	0.01	178	0.01
p_hat1000-1	10	19.10	13404	0.49
p_hat1000-2	≥46	≥21600.00	1379492	6.54
p_hat1000-3	≥64	≥21600.00	1200137	1973.85
p_hat1500-1	12	235.02	107530	234.21
p_hat1500-2	≥65	≥21600.00	844836	8819.92
p_hat1500-3	≥92	≥21600.00	443039	7084.69
p_hat300-1	8	0.04	328	0.01
p_hat300-2	**25**	**0.83**	**630**	**0.05**
p_hat300-3	36	165.33	18873	0.61
p_hat500-1	9	0.29	560	0.07
p_hat500-2	**36**	**34.57**	**4163**	**0.19**
p_hat500-3	≥50	≥21600.00	1183948	38.64
p_hat700-1	11	0.73	758	0.20
p_hat700-2	**44**	**447.60**	**30107**	**11.51**
p_hat700-3	≥62	≥21600.00	941276	2.53
san1000	15	101.81	8185	101.60
san200_0.7_1	30	0.43	675	0.43
san200_0.7_2	**18**	**0.26**	**358**	**0.25**
san200_0.9_1	70	1.88	2615	1.88
san200_0.9_2	60	2.06	2078	2.05
san200_0.9_3	44	449.27	69006	449.27
san400_0.5_1	13	1.23	525	1.22
san400_0.7_1	**40**	**13.14**	**3017**	**13.13**
san400_0.7_2	30	97.36	11358	97.36
san400_0.7_3	22	606.96	145060	606.96
san400_0.9_1	100	534.17	35170	534.16
sanr200_0.7	18	21.38	9449	1.03
sanr200_0.9	42	7010.62	1007007	5937.11
sanr400_0.5	13	25.41	12476	19.10
sanr400_0.7	21	13619.40	4762337	254.64

Instance	$\lvert C^*\rvert$	time	BB node	time best
brock200_1	21	92.73	40856	88.77
brock200_2	12	0.27	312	0.27
brock200_3	15	3.65	2694	3.65
brock200_4	17	4.92	2295	3.20
brock400_1	≥25	≥21600.00	5705834	1607.69
brock400_2	≥25	≥21600.00	6428350	20814.80
brock400_3	≥24	≥21600.00	6504665	67.62
brock400_4	≥25	≥21600.00	5702967	3704.13
brock800_1	≥20	≥21600.00	7062587	3779.79
brock800_2	≥20	≥21600.00	7019896	894.75
brock800_3	≥22	≥21600.00	5381786	4083.26
brock800_4	≥20	≥21600.00	6979390	4839.30
c-fat200-1	12	0.01	211	0.01
c-fat200-2	24	0.01	223	0.01
c-fat200-5	58	0.01	257	0.01
c-fat500-1	14	0.04	513	0.04
c-fat500-10	126	0.13	625	0.08
c-fat500-2	26	0.06	525	0.05
c-fat500-5	64	0.06	563	0.05
hamming10-2	512	5.22	1535	1.27
hamming10-4	≥32	≥21600.00	5795183	0.42
hamming6-2	32	0.01	95	0.01
hamming6-4	4	0.01	67	0.01
hamming8-2	128	0.07	383	0.01
hamming8-4	16	3.20	733	0.01
johnson16-2-4	8	12.23	58009	0.01
johnson32-2-4	≥16	≥21600.00	14886797	0.05
johnson8-2-4	4	0.01	31	0.01
johnson8-4-4	14	0.01	92	0.01
keller4	11	1.99	3043	0.94
keller5	≥20	≥21600.00	13350366	1619.28
keller6	≥39	≥21600.00	1170365	1493.98
MANN_a27	126	1833.36	25867	324.82
MANN_a45	≥342	≥21600.00	17512	1753.16
MANN_a81	≥1080	≥21600.00	5143	22.95
MANN_a9	16	0.01	129	0.01
p_hat1000-1	10	22.46	12837	0.50
p_hat1000-2	**≥46**	**≥21600.00**	**1386354**	**5.11**
p_hat1000-3	**≥65**	**≥21600.00**	**1183638**	**20679.90**
p_hat1500-1	12	274.43	106328	272.96
p_hat1500-2	**≥65**	**≥21600.00**	**835237**	**6833.05**
p_hat1500-3	**≥92**	**≥21600.00**	**412249**	**5213.79**
p_hat300-1	8	0.05	309	0.01
p_hat300-2	25	0.92	361	0.01
p_hat300-3	**36**	**166.16**	**17439**	**0.51**
p_hat500-1	9	0.44	529	0.08
p_hat500-2	36	36.35	3579	0.11
p_hat500-3	**≥50**	**≥21600.00**	**1134209**	**32.67**
p_hat700-1	11	1.26	721	0.23
p_hat700-2	44	456.30	27572	9.88
p_hat700-3	**≥62**	**≥21600.00**	**939722**	**1.43**
san1000	15	142.59	9769	142.30
san200_0.7_1	30	0.41	440	0.41
san200_0.7_2	18	0.29	386	0.28
san200_0.9_1	**70**	**0.90**	**726**	**0.90**
san200_0.9_2	**60**	**1.64**	**827**	**1.62**
san200_0.9_3	44	447.97	66651	447.97
san400_0.5_1	13	1.61	714	1.60
san400_0.7_1	40	14.24	2570	14.23
san400_0.7_2	30	114.90	12762	114.90
san400_0.7_3	22	766.37	178388	766.37
san400_0.9_1	**100**	**493.40**	**27224**	**493.37**
sanr200_0.7	18	21.82	8887	1.03
sanr200_0.9	**42**	**6853.37**	**859718**	**5798.95**
sanr400_0.5	13	27.65	12097	20.62
sanr400_0.7	21	13877.50	4531210	255.15

Table B.22: *Alg. 21 + Domain Filtering +* $\mathcal{U}_8$

B.4 Combined Statistics

When collecting all results of the previously presented tables, we have a database of more than 1300 different experiments. The following tables analyze the collected data. We group those of the 1300 experiments that have used a certain technique and compare them to those that have not used it (see also Sec. 10.3.5). The classification categories are described in Sec. B.2.

Summary: Domain Filtering versus no Domain Filtering

	domain filtering	no domain filtering
Instances:	660	660
within timelimit:	401	395
best $\|C^*\|$:	639	606
fastest termination for best $\|C^*\|$:	447	532
best $\|C^*\|$ with fewest bb nodes:	560	140
fastest termination for best $\|C^*\|$ with fewest bb nodes:	418	87
fastest best $\|C^*\|$ found:	440	446
overall best:	426	274
total time :	1657.14 h	1682.17 h
total size :	37394	37461

Summary: Coloring Bounds versus no Coloring Bounds

	using $\mathcal{U}_8$	not using $\mathcal{U}_8$
Instances:	660	660
within timelimit:	461	335
best $\|C^*\|$:	641	490
fastest termination for best $\|C^*\|$:	343	537
best $\|C^*\|$ with fewest bb nodes:	490	245
fastest termination for best $\|C^*\|$ with fewest bb nodes:	343	74
fastest best $\|C^*\|$ found:	292	481
overall best:	424	304
total time :	1298.26 h	2041.06 h
total size :	39623	35232

Summary: Domain Filtering plus Coloring Bounds versus Either of Both

	$\mathcal{U}_8$ + DF	$\mathcal{U}_8$, no DF
Instances:	330	330
within timelimit:	231	230
best $\|C^*\|$:	328	314
fastest termination for best $\|C^*\|$:	238	250
best $\|C^*\|$ with fewest bb nodes:	274	78
fastest termination for best $\|C^*\|$ with fewest bb nodes:	216	59
fastest best $\|C^*\|$ found:	237	215
overall best:	226	126
total time :	646.49 h	651.77 h
total size :	19800	19823

	$\mathcal{U}_8$ + DF	not using $\mathcal{U}_8$, but DF
Instances:	330	330
within timelimit:	231	170
best $\|C^*\|$:	322	247
fastest termination for best $\|C^*\|$:	177	271
best $\|C^*\|$ with fewest bb nodes:	247	129
fastest termination for best $\|C^*\|$ with fewest bb nodes:	177	45
fastest best $\|C^*\|$ found:	151	246
overall best:	217	158
total time :	646.49 h	1010.65 h
total size :	19800	17594

Summary: Heuristics versus no Heuristics

	using heuristics	not using heuristics
Instances:	528	528
within timelimit:	323	313
best $\|C^*\|$:	524	468
fastest termination for best $\|C^*\|$:	408	444
best $\|C^*\|$ with fewest bb nodes:	400	133
fastest termination for best $\|C^*\|$ with fewest bb nodes:	342	75
fastest best $\|C^*\|$ found:	398	354
overall best:	389	142
total time :	1311.35 h	1376.61 h
total size :	31970	27283

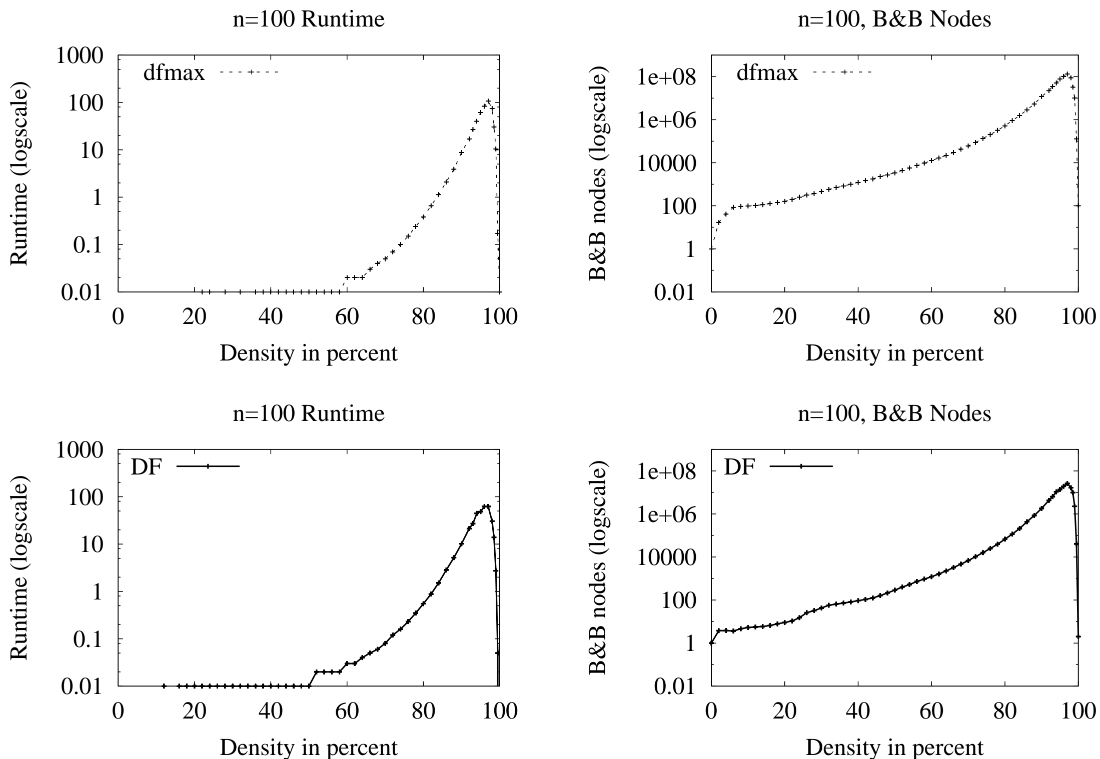
Summary: Reordering versus Static Order

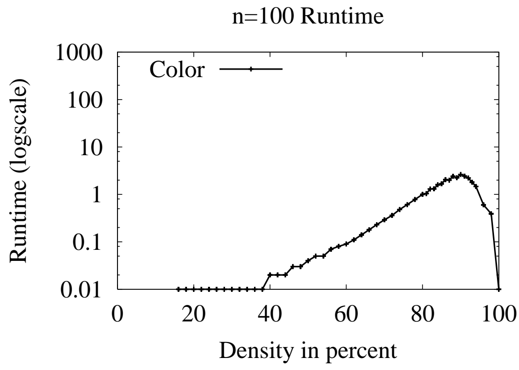
	re-ordering	static ordering
Instances:	528	528
within timelimit:	318	318
best $\lvert C^* \rvert$:	485	491
fastest termination for best $\lvert C^* \rvert$:	380	397
best $\lvert C^* \rvert$ with fewest bb nodes:	316	277
fastest termination for best $\lvert C^* \rvert$ with fewest bb nodes:	291	247
fastest best $\lvert C^* \rvert$ found:	336	357
overall best:	276	313
total time :	1348.22 h	1339.74 h
total size :	29214	30039

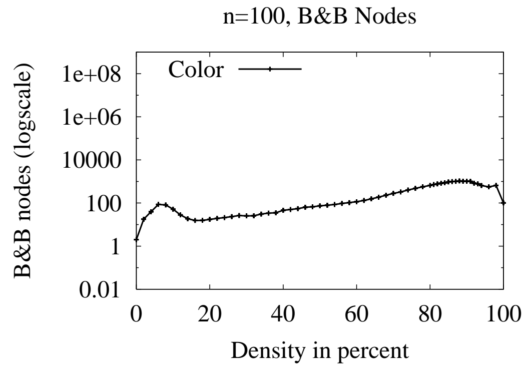
B.5 Random Graphs

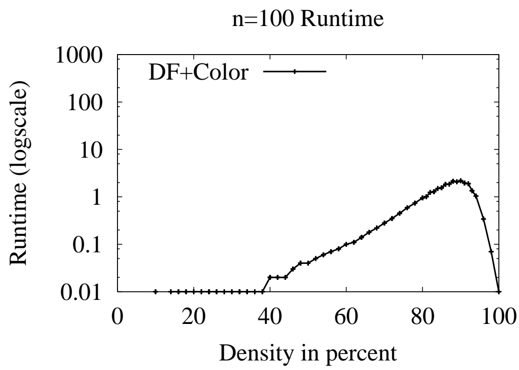
We presented diagrams for convergence of various techniques on random graphs in Sec. 10.4. In the following figures we split these diagrams, thus allowing a more detailed view of the behavior of each individual technique.

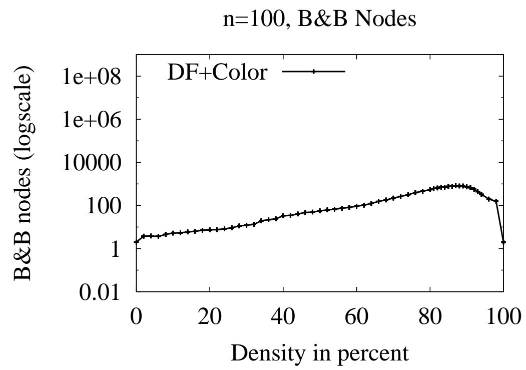
Graphs with 100 Nodes

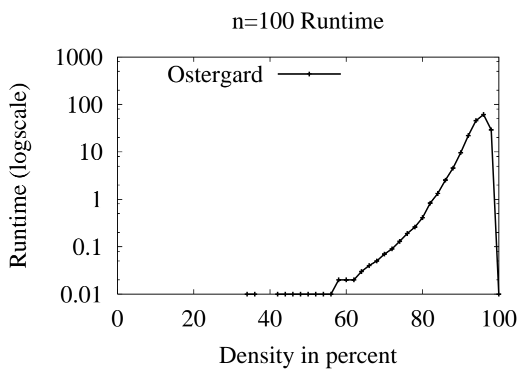


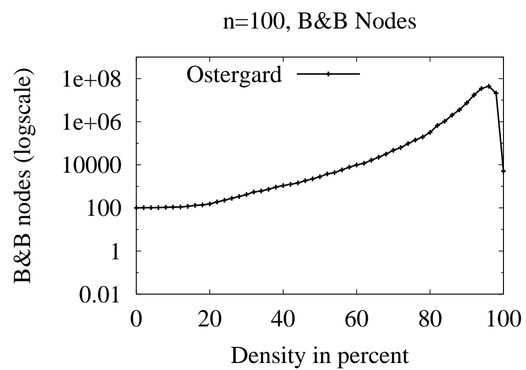

n=100 Runtime
Color
Runtime (logscale)
1000
100
10
1
0.1
0.01
0
20
40
60
80
100
Density in percent

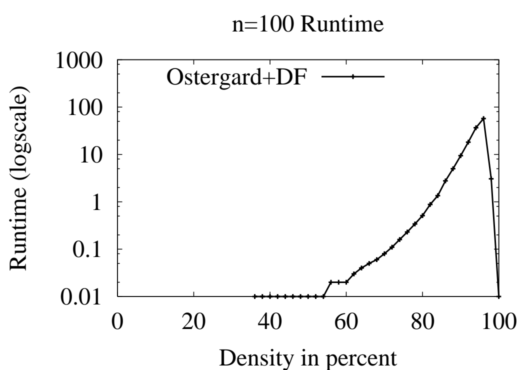

n=100, B&B Nodes
Color
B&B nodes (logscale)
1e+08
1e+06
10000
100
1
0.01
0
20
40
60
80
100
Density in percent

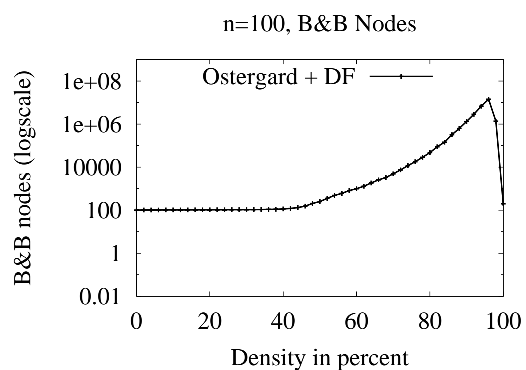

n=100 Runtime
DF+Color
Runtime (logscale)
1000
100
10
1
0.1
0.01
0
20
40
60
80
100
Density in percent

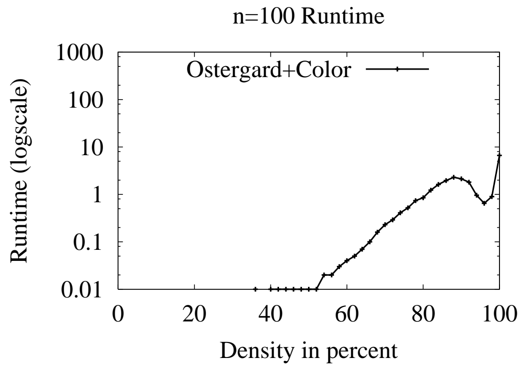

n=100, B&B Nodes
DF+Color
B&B nodes (logscale)
1e+08
1e+06
10000
100
1
0.01
0
20
40
60
80
100
Density in percent

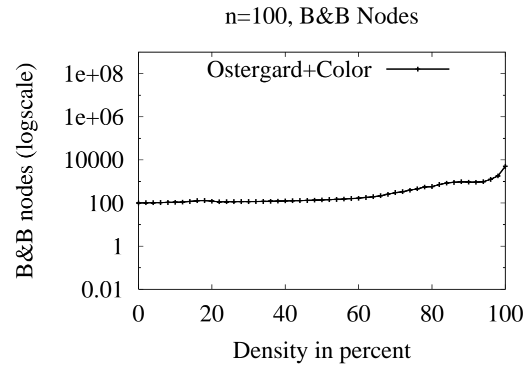

n=100 Runtime
Ostergard
Runtime (logscale)
1000
100
10
1
0.1
0.01
0
20
40
60
80
100
Density in percent

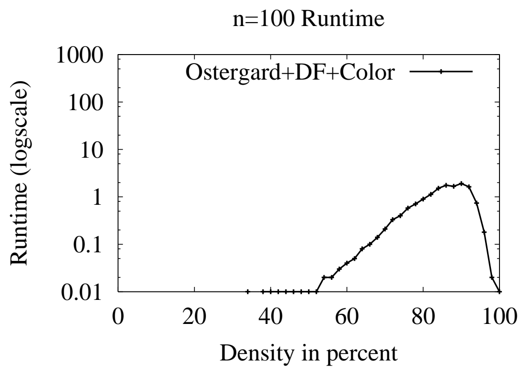

n=100, B&B Nodes
Ostergard
B&B nodes (logscale)
1e+08
1e+06
10000
100
1
0.01
0
20
40
60
80
100
Density in percent

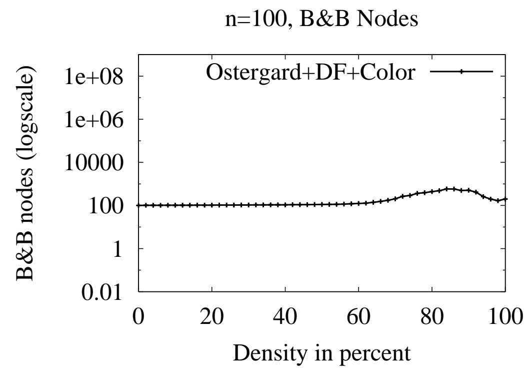

n=100 Runtime
Ostergard+DF
Runtime (logscale)
1000
100
10
1
0.1
0.01
0
20
40
60
80
100
Density in percent

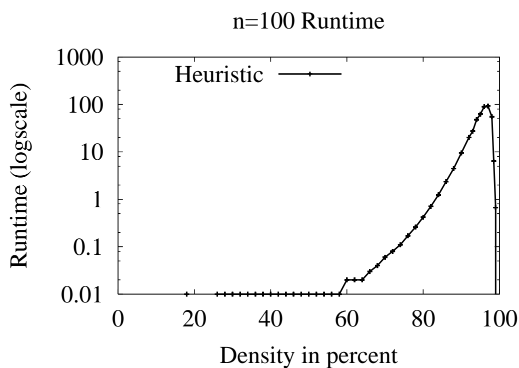

n=100, B&B Nodes
Ostergard + DF
B&B nodes (logscale)
1e+08
1e+06
10000
100
1
0.01
0
20
40
60
80
100
Density in percent

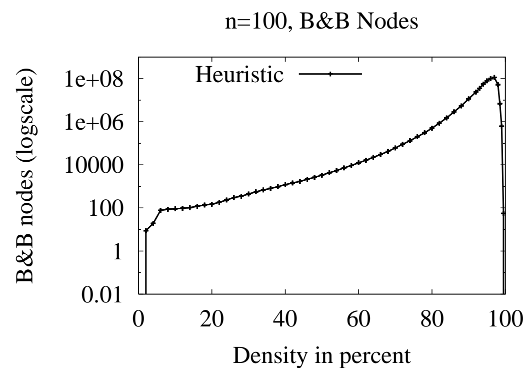

n=100 Runtime
Ostergard+Color
Runtime (logscale)
1000
100
10
1
0.1
0.01
0
20
40
60
80
100
Density in percent

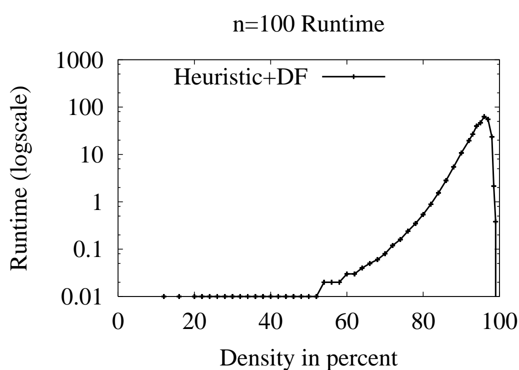

n=100, B&B Nodes
Ostergard+Color
B&B nodes (logscale)
1e+08
1e+06
10000
100
1
0.01
0
20
40
60
80
100
Density in percent

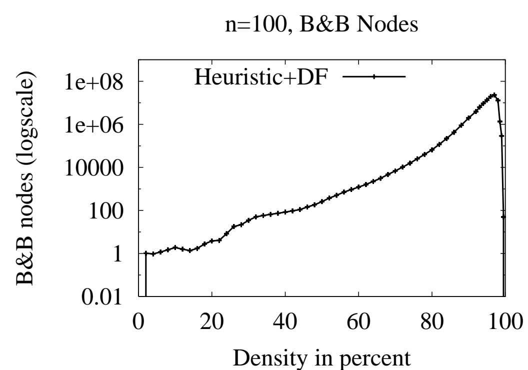

n=100 Runtime
Ostergard+DF+Color
Runtime (logscale)
1000
100
10
1
0.1
0.01
0
20
40
60
80
100
Density in percent

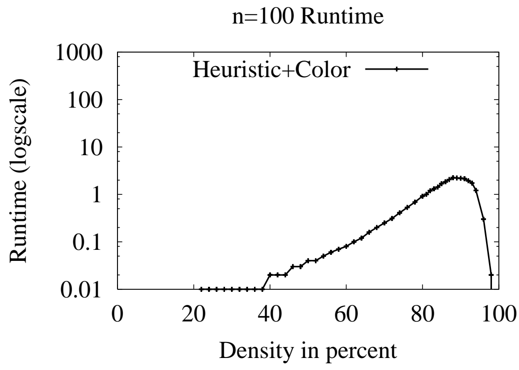

n=100, B&B Nodes
Ostergard+DF+Color
B&B nodes (logscale)
1e+08
1e+06
10000
100
1
0.01
0
20
40
60
80
100
Density in percent

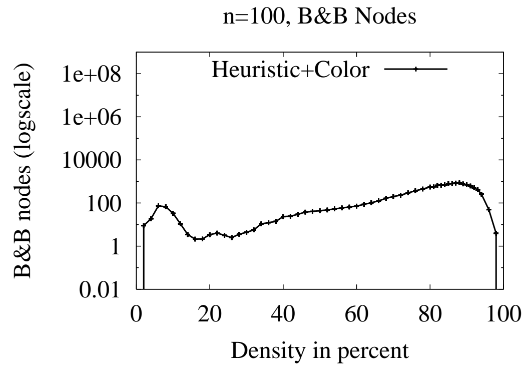

n=100 Runtime
Heuristic
Runtime (logscale)
1000
100
10
1
0.1
0.01
0
20
40
60
80
100
Density in percent

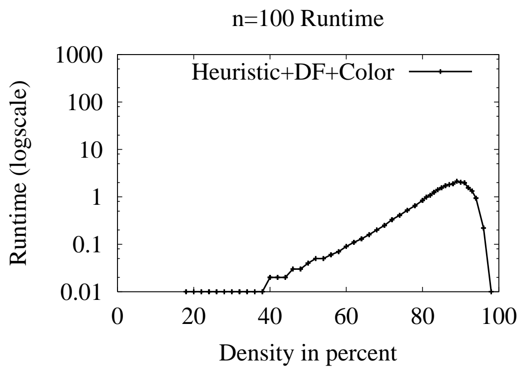

n=100, B&B Nodes
Heuristic
B&B nodes (logscale)
1e+08
1e+06
10000
100
1
0.01
0
20
40
60
80
100
Density in percent

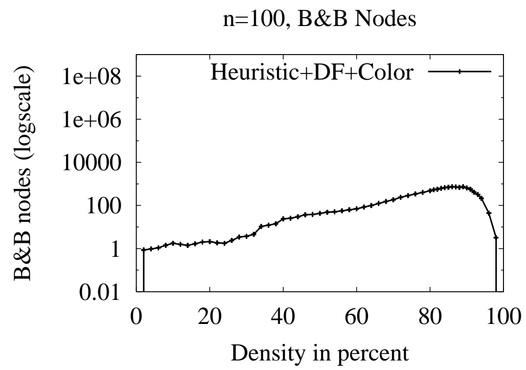

n=100 Runtime
Heuristic+DF
Runtime (logscale)
1000
100
10
1
0.1
0.01
0
20
40
60
80
100
Density in percent


n=100, B&B Nodes
Heuristic+DF
B&B nodes (logscale)
1e+08
1e+06
10000
100
1
0.01
0
20
40
60
80
100
Density in percent

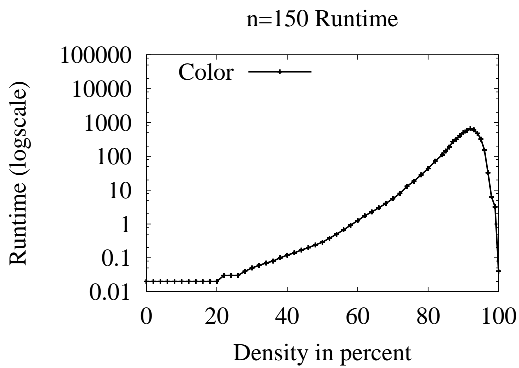


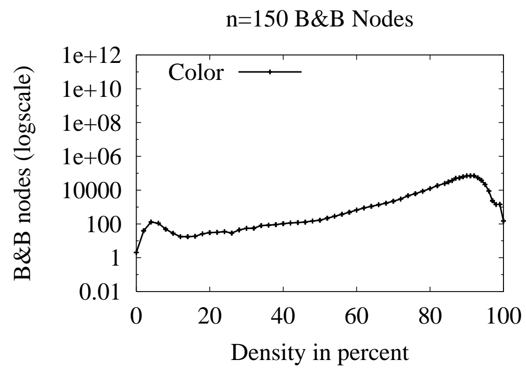


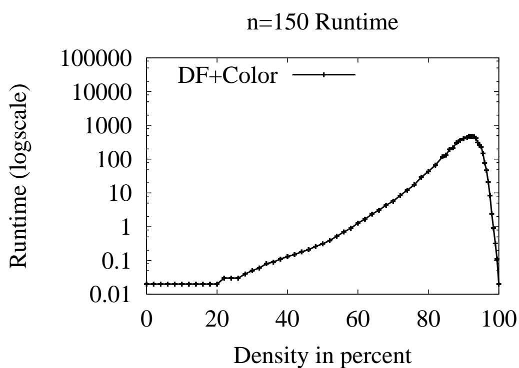


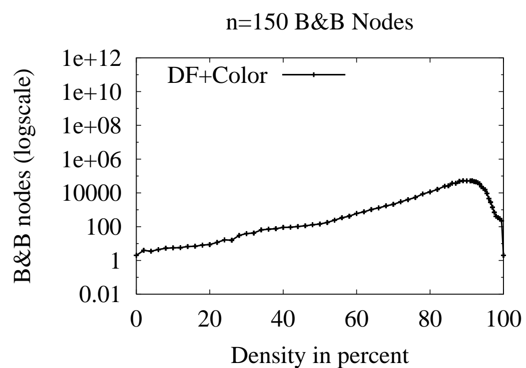


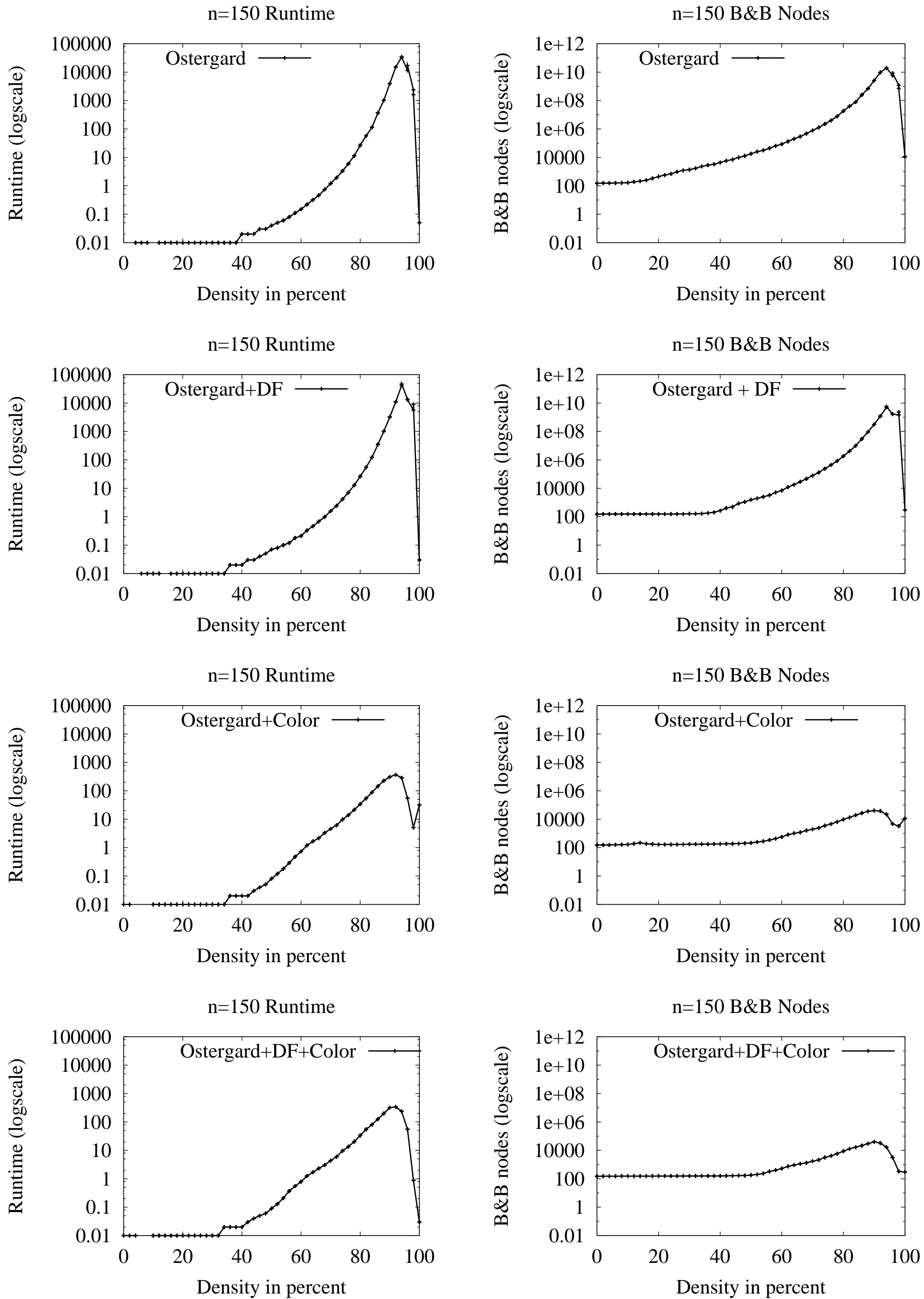
Graphs with 150 Nodes

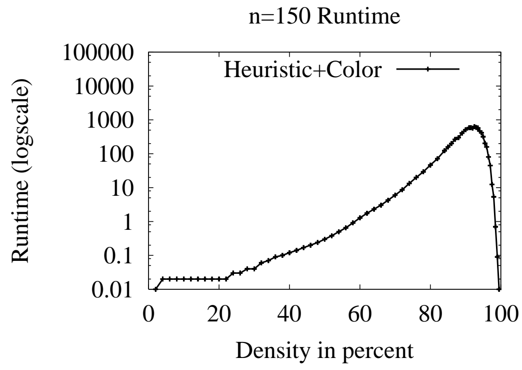


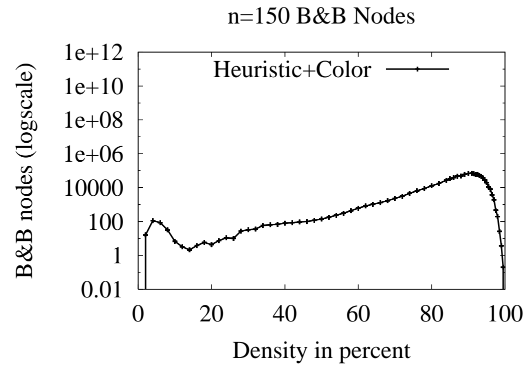


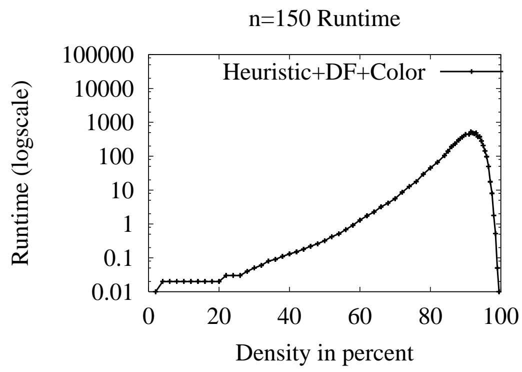


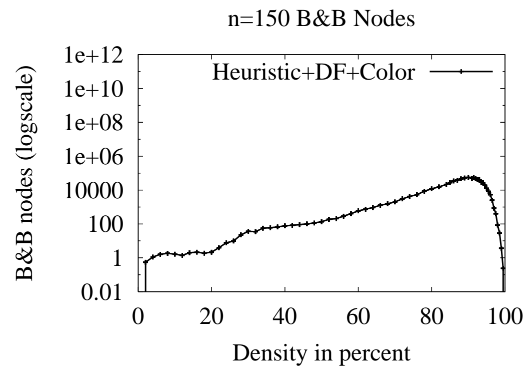



n=150 Runtime
Ostergard
Runtime (logscale)
Density in percent
n=150 B&B Nodes
B&B nodes (logscale)
n=150 Runtime
Ostergard+DF
n=150 B&B Nodes
Ostergard + DF
n=150 Runtime
Ostergard+Color
n=150 B&B Nodes
Ostergard+Color
n=150 Runtime
Ostergard+DF+Color
n=150 B&B Nodes
Ostergard+DF+Color


n=150 Runtime
Heuristic+Color
Runtime (logscale)
100000
10000
1000
100
10
1
0.1
0.01
0
20
40
60
80
100
Density in percent


n=150 B&B Nodes
Heuristic+Color
B&B nodes (logscale)
1e+12
1e+10
1e+08
1e+06
10000
100
1
0.01
0
20
40
60
80
100
Density in percent


n=150 Runtime
Heuristic+DF+Color
Runtime (logscale)
100000
10000
1000
100
10
1
0.1
0.01
0
20
40
60
80
100
Density in percent


n=150 B&B Nodes
Heuristic+DF+Color
B&B nodes (logscale)
1e+12
1e+10
1e+08
1e+06
10000
100
1
0.01
0
20
40
60
80
100
Density in percent

List of Figures

3.1 The width of each element is proportional to its weight. The elements are ordered with respect to the efficiencies p_i/w_i. The leftmost element has the biggest efficiency, and the rightmost the smallest one. s marks the critical item in U_1. 34

3.2 U_3 requires $s \in \{0, 1\}$. The figures show $U_1(\mathrm{KP}[x_s = 0])$, and $U_1(\mathrm{KP}[x_s = 1])$. 35

3.3 The figure illustrates the proceeding of the reduction algorithm presented for $\mathrm{KP}[x_i = 0]$. The weight ordering in which the items are tested ensures that the critical item moves monotonically to the right. 37

4.1 An optimal and legal roster is equivalent to a constrained shortest path in a weighted DAG. 54

4.2 The entire approach: The inner loop generates columns using dual information, the outer loop solves the master problem. 62

4.3 Number of choice points versus master iteration (left), and running time versus master iteration (right) for SPC, NRC, and total enumeration. The tests were run with a data instance of type 10-0-20 that was solved to optimality. (Lines connecting measuring points are given for easier readability and do not mark intermediate values.) 63

4.4 Number of choice points versus master iteration using SPC, NRC with a data set of type 7-0-30. 64

4.5 The picture shows time versus the number of calls of the propagation routine using the incremental and the non-incremental implementation of the shortest path constraint. Both versions were stopped after $10\,000$ seconds total CPU time. The experiment was run with a data instance of type 10-0-70. 64

4.6 Time versus quality on a data instance of type 67-165-280. The picture shows a comparison of NRC (upper curve) and SPC (lower curve). 65

5.1 Conflicting columns in the CGA when considering a set partitioning problem as the master problem 70

5.2 Schematic view on the first way of integrating column generation and heuristic tree search via a set covering approach on the one, and a repair method on the other side. Column from overcovered rows are deassigned, and the remaining information is used to guide the search in the HTS (gray triangle). If the solution cannot be repaired, backtracking allows to deassign more (dashed lines). 72

5.3 Schematic view on the second way of integrating column generation and heuristic tree search: HTS provides an initial solutions, and thus, meaningful dual values. During optimization, CGA provides new dual values and requests solutions from HTS that respect an objective which is based on reduced costs. 75
5.4 Data set with 65 crew members and 959 pairings. Column generation fails to find a solution within 120 000 sec. 77
5.5 Data set with 50 crew members and 766 pairings. Column generation fails to find a solution within 120 000 sec. 78
5.6 Data set with 7 crew members and 129 pairings. 79
5.7 Data set with 30 crew members and 279 pairings. 80

6.1 An overview of constraints relevant to the home health care problem. 84
6.2 Penalty concept for Time Windows. Arriving before ***hard_begin*** produces waiting time, arriving after ***hard_end*** is infeasible. Penalties proportional to earliness or lateness are used for arrivals within the hard, but before or after the soft time window. 86
6.3 LP (6.21): Finding optimal starting times for a given sequence 92
6.4 Sequencing cache fail rate 102
6.5 Comparison of the heuristics 104
6.6 Comparison of combined heuristics 106

7.1 The scenario for the Automatic Recording Problem. 108

8.1 The polytope of (a) $\{x \in [0,1]^3 \mid x_1 + x_2 \leq 1 \ \wedge \ x_1 + x_3 \leq 1 \ \wedge \ x_2 + x_3 \leq 1\}$ and the corresponding valid inequality (b) $\{x \in [0,1]^3 \mid x_1 + x_2 + x_3 \leq 1\}$. 120
8.2 The polytope of (a) $\{x \in [0,1]^3 \mid 2x_1 + 3x_2 + 4x_3 \leq 7\}$ and the tighter representation (b) $\{x \in [0,1]^3 \mid 2x_1 + 3x_2 + 4x_3 \leq 7 \ \wedge \ x_1 + x_2 + x_3 \leq 2\}$. 123
8.3 A well-known problem: Extreme points may occur in the intersection of polytopes. (a) shows the initial LP inequalities (green/magenta) of two substructure and the convex hull of feasible integer points (blue). Using tight cuts for the two substructures we find the convex hull of either substructure (b). When intersecting these regions (c), new fractional extreme points occur (arrows). . 124
8.4 Robustness of the approaches for all 50 instances in 5CU, 24 h, 50 ch. Runtime is given on a logarithmic scale. Figures for other classes look similar. (Lines connecting measuring points are given for easier readability and do not mark intermediate values.) 135

9.1 (a) a clique C and its candidate set P. (b) applying lemma 5 fixes one more node. (c) as a result, two nodes now have degree zero, and can be eliminated according to lemma 4. 143
9.2 (a) $\mathcal{U}_2$ versus $\mathcal{U}_5$, (b) $\mathcal{U}_4$ versus $\mathcal{U}_7$ 148
9.3 Three graphs showing that $\mathcal{U}_{LP(9.1)}$ can be (a) worse than $\mathcal{U}_3$, (b) worse than $\mathcal{U}_5$, or (c) better than $\mathcal{U}_8$, respectively. 148
9.4 Graphs used for illustrating the connection between domain filtering and $\mathcal{U}_8$. . 150

10.1 Random graphs with 100 nodes. Runtime and the number of branch-and-bound nodes, respectively, given on the y-axis in logarithmic scale. On the x-axis, graph density varies from 0% – 100%. Lines connecting measuring points are given for easier readability and do not mark intermediate values. (see also Appendix B.5.) 165
10.2 Random graphs with 150 nodes. The y-axis gives runtime and number of branch-and-bound nodes, respectively. The y-axis is scaled logarithmic On the x-axis, graph density varies from 0% – 100%. Each sample point represents an average value of 50 instances. (see also Appendix B.5.) 166
10.3 Random graphs with 150 nodes. The y-axis gives runtime and number of branch-and-bound nodes, respectively. The y-axis is scaled linearly. On the x-axis, graph density varies from 0% – 100%. Each sample point represents an average value of 50 instances. 167

List of Tables

1.1 A comparison of CP and ILP adapted from Trick [200]. 3

3.1 Characteristics of the four algorithms used in the experiments. 41
3.2 The pure CP approach for both problem classes. ***cp*** is the average number of choice points, ***time*** the average time in seconds for 100 instances of the given size. 42
3.3 Uncorrelated data instances. We give the average numbers for 100 test sets per size. ***time*** is the time in seconds, ***cp*** the number of choice points. 42
3.4 Weakly correlated data instances. We give the average numbers for 100 test sets per size. ***time*** is the time in seconds, ***cp*** the number of choice points. . . 43
3.5 Uncorrelated and weakly correlated data instances. We give the average time per choice point in milliseconds for 100 test sets per size. 44
3.6 Uncorrelated data. Comparison of running times for the new amortized linear time propagation algorithms and implementations of DHR, and MTR. We give the average time in seconds as well as the number of choice points for 100 test sets per size. 44
3.7 Uncorrelated data. Comparison of running times per choice point for the new amortized linear time propagation algorithm based on bound U_2 and the implementation of MTR. We give the average time per choice point in milliseconds for 100 test sets per size. 45
3.8 Comparison of running times of linU_2 and MTR on uncorrelated and weakly correlated data. ***cp*** is the number of choice points, ***time*** the running time in seconds. 45

4.1 Domain reductions after an assignment 53

6.1 Sequencing algorithm efficiency 101
6.2 Finding initial solutions via CP 102
6.3 Results for constraint programming (CP), Simulated Annealing (SA), and Tabu Search (TS) started from an empty solution, and for Tabu Search started on the first solution found by constraint programming (CP+TS). A dash indicates that no solution was found. Each approach ran for 600sec. 104
6.4 Comparing combined approaches. Results for CP+TS, alternating CP and TS (CP+TS loop), and applying TS to 10 initial CP solutions (10* CP+TS). Running times between 600sec and 840sec. 105

7.1 Test sets with 5 classes, 12 hours, 20 channels and different objectives. Times in seconds and choice points are averages for 50 randomly generated instances. The average number of programs per instance is between 607.6 and 612.6. 114
7.2 Test sets with 5 classes, objective CU for various time horizons (in hours) and channel numbers (ch). Italic numbers give the average (avg) time and nodes, resp., of 50 instances. Numbers below are: minimum (min), maximum (max), and standard deviation (std) for these 50 instances. The average number of programs per instance is 315.2 for (12h/20ch), 793.5 (12h/50ch), 607.6 (24h/20ch), 1512.1 (24h/50ch), and 1782.6 (72h/20ch), resp. 115
7.3 Effectiveness of the different approaches for different benchmark classes. We have an average of 1956.7 programs for the 120h/20ch test, 1782.6 for 72h/20ch, and 1423.3 for 24h/50ch. 116
7.4 Subset sum data sets for 12 hours, 20 channels, up to 72 hours and 50 channels. The average number of programs is given as parameter p in the upper table. 116
7.5 Different benchmark sets for $\approx \mathbf{1000} - \mathbf{1200}$ programs for 3 classes and objective CU. 117

8.1 Test sets with 5 classes, 12 hours, 20 channels and different objectives. Times in seconds and choice points are averages for 50 randomly generated instances. The average number of movies per instance is between 607.6 and 612.6. 128
8.2 Test sets with 5 classes, objective CU for various time horizons (in hours) and channel numbers (ch). Italic numbers give the average (avg) time and nodes, resp., of 50 instances. Numbers below are: minimum (min), maximum (max), and standard deviation (std) for these 50 instances. The average number of movies per instance is 315.2 for (12 h/20 ch), 793.5 (12 h/50 ch), 607.6 (24 h/20 ch), 1512.1 (24 h/50 ch), and 1782.6 (72h/20 ch), resp. 129
8.3 Effectiveness of the different approaches for different benchmark classes. We have an average of 1956.7 programs for the 120h/20 ch test, 1782.6 for 72h/20 ch, and 1423.3 for 24 h/50 ch. 129
8.4 Subset sum data sets for 12 hours, 20 channels, up to 72 hours and 50 channels. 130
8.5 Different benchmark sets for $\approx \mathbf{1000} - \mathbf{1200}$ programs for 3 classes and objective CU. 130

10.1 Algorithm 20 on DIMACS Instances (Part I). We compare the number of choice points and running time in seconds of our implementation of `dfmax`, our algorithm using coloring bounds only (χ), our algorithm using domain filtering only (DF), and a fourth using both ($\chi + DF$). Boldface numbers indicate best runs. (see also summary on page 192). 158
10.2 Algorithm 20 on DIMACS Instances (Part II). We compare the number of choice points and running time in seconds of our implementation of `dfmax`, our algorithm using coloring bounds only (χ), our algorithm using domain filtering only (DF), and a fourth using both ($\chi + DF$). Boldface numbers indicate best runs. (see also summary on page 192). 159

10.3 Comparison of Algorithm 20 plus domain filtering, coloring bounds, and primal heuristics. Boldface numbers indicate best runs in comparison to the same setting without primal heuristics (fourth column in tables 10.1, 10.2). 161

B.1 Algorithm 21 on DIMACS Instances (Part I). We compare the number of choice points and running time in seconds of our implementation of Östergård's idea, our algorithm using coloring bounds only (χ), our algorithm using domain filtering only (DF), and a fourth using both ($\chi + DF$). Boldface numbers indicate best runs. (see also summary on page 192). 193
B.2 Algorithm 21 on DIMACS Instances (Part II). We compare the number of choice points and running time in seconds of our implementation of Östergård's idea, our algorithm using coloring bounds only (χ), our algorithm using domain filtering only (DF), and a fourth using both ($\chi + DF$). Boldface numbers indicate best runs. (see also summary on page 192). 194
B.3 Alg. 20 196
B.4 Alg. 20 + Domain Filtering 197
B.5 Alg. 20 + $\mathcal{U}_8$ 198
B.6 Alg. 20 + Domain Filtering + $\mathcal{U}_8$ 199
B.7 Alg. 20 + Lower Bound Heuristics 200
B.8 Alg. 20 + Domain Filtering + Lower Bound Heuristics 201
B.9 Alg. 20 + $\mathcal{U}_8$ + Lower Bound Heuristics 202
B.10 Alg. 20 + Domain Filtering + $\mathcal{U}_8$+ Lower Bound Heuristics 203
B.11 Alg. 20 + Reordering 204
B.12 Alg. 20 + Reordering + Domain Filtering 205
B.13 Alg. 20 + Reordering + $\mathcal{U}_8$ 206
B.14 Alg. 20 + Reordering + Domain Filtering + $\mathcal{U}_8$ 207
B.15 Alg. 20 + Reordering + Lower Bound Heuristics 208
B.16 Alg. 20 + Reordering + Domain Filtering + Lower Bound Heuristics 209
B.17 Alg. 20 + Reordering + $\mathcal{U}_8$ + Lower Bound Heuristics 210
B.18 Alg. 20 + Reordering + Domain Filtering + $\mathcal{U}_8$ + Lower Bound Heuristics . . 211
B.19 Alg. 21 212
B.20 Alg. 21 + Domain Filtering 213
B.21 Alg. 21 + $\mathcal{U}_8$ 214
B.22 Alg. 21 + Domain Filtering + $\mathcal{U}_8$ 215

List of Algorithms

1 Revise an arc (i, j) . . . 13
2 Arc consistency for a network of constraints: AC-3 (Mackworth [137]) . . . 13
3 LP-based Branch-and-Bound for a minimization IP P . . . 16
4 Outline of the Column Generation . . . 18
5 (Non-Incremental) Shortest Path Constraint . . . 58
6 Propagation Algorithm for Rules (R1) and (R2) . . . 59
7 A Non-Deterministic Algorithm Describing the Search Tree . . . 59
8 Top level algorithm for the first method . . . 72
9 Heuristics for the first method . . . 73
10 Top level algorithm for the second method . . . 74
11 Modified value selection heuristic for the second method . . . 75
12 Find best sequence for set U . . . 93
13 Goal 1: Branch on job with smallest domain, and assign best nurse first . . . 94
14 Simulated Annealing . . . 96
15 Tabu Search . . . 97
16 A pool Ω stores initial and improved solutions found during optimization . . . 99
17 Goal 2: Branch on job with smallest domain and assign the nurse that was also used in solution L . . . 99
18 Goal 3: Branch on job with best soft constraint values first, and assign the nurse that was also used in solution L . . . 99
19 Goal 4: Branch on job/nurse pairs that occur in good solutions found earlier . 100
20 The Carraghan/Pardalos Algorithm for Finding a Maximum Clique C^* . . . 139
21 The Östergård Algorithm for Finding a Maximum Clique C^* . . . 140
22 Domain Filtering for Maximum Clique Problem . . . 141
23 DSATUR greedy method . . . 154
24 Bigg's Greedy Method for Graph Coloring . . . 154

Own Publications

Parts of this thesis have been presented at conferences, and have been (or will be) published in journals, respectively. Previous versions of some of these results have additionally been presented at workshops. Some workshops only provide informal proceedings. This includes the CP-AI-OR workshop series[1], Route 2000[2], ISMP 2000[3], the IJCAI'99[4] workshop on non-binary constraints and the ICGCO[5]. Whereas the CP-AI-OR conference series is peer-reviewed, the other workshops are not.

The following lists publications co-authored by the author of this thesis.

Journals

[184] J. SCHULZE AND T. FAHLE. A parallel algorithm for the vehicle routing problem with time window constraints. *Annals of Operations Research*, 86:585–607, 1999.

[73] 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.

[77] T. FAHLE AND M. SELLMANN. Cost based filtering for the constrained knapsack problem. *Annals of Operations Research*, 115:73–94, 2002.

[189] M. SELLMANN, K. ZERVOUDAKIS, P. STAMATOPOULOS, AND T. FAHLE. Crew assignment via constraint programming: Integrating column generation and heuristic tree search. *Annals of Operations Research*, 115:207–226, 2002.

[187] M. SELLMANN AND T. FAHLE. Constraint programming based lagrangian relaxation for the automatic recording problem. *Annals of Operations Research*, 118:17–33, 2003 (to appear).

Conferences

[125] U. JUNKER, S. E. KARISCH, N. KOHL, B. VAABEN, T. FAHLE, AND M. SELLMANN. A framework for constraint programming based column generation. In *Proceedings of CP'99*, volume 1713 of *LNCS*, pages 261–274. Springer, 1999.

[186] M. SELLMANN AND T. FAHLE. Coupling variable fixing algorithms for the automatic recording problem. In *Proceedings of ESA'01, Århus/Denmark*, volume 2161 of *LNCS*, pages 134–145. Springer, 2001.

[75] T. FAHLE, S. SCHAMBERGER, AND M. SELLMANN. Symmetry breaking. In *Proceedings of the CP'01*, volume 2239 of *LNCS*, pages 93–107. Springer, 2001.

[71] T. FAHLE. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In *Proceedings of ESA'02, Rome/Italy*, volume 2461 of *LNCS*, pages 485–498. Springer, 2002.

[1] 1st–4th International Workshop on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming for Combinatorial Optimization Problems, Ferrara/Italy, 1999, Paderborn/Germany, 2000, Wye/UK, 2001, Le Croisic/France, 2002.

[2] International Conference on Vehicle Routing, Skodsborg, Denmark, 2000.

[3] 17th International Symposium on Mathematical Programming, Atlanta/USA, 2000.

[4] 16th International Joint Conference on Artificial Intelligence, Stockholm, Sweden, 1999.

[5] International Conference on Global and Combinatorial Optimization, ICGCO, Chania, Greece, 1998.

Refereed Workshops

[76] T. FAHLE AND M. SELLMANN. Constraint programming based column generation with knapsack subproblems. In *Proceedings of the CP-AI-OR'00, Paderborn/Germany*, pages 33–44, 2000.

[188] M. SELLMANN, K. ZERVOUDAKIS, P. STAMATOPOULOS, AND T. FAHLE. Integrating direct CP search and CP-based column generation for the airline crew assignment problem. In *Proceedings of the CP-AI-OR'00, Paderborn/Germany*, page 163–170, 2000.

[185] M. SELLMANN AND T. FAHLE. Constraint programming based lagrangian relaxation for a multimedia application. In *Proceedings CP-AI-OR'01, Ashford/UK*, pages 1–14, 2001.

[70] T. FAHLE. Cost based filtering vs. upper bounds for maximum clique. In *Proceedings of CP-AI-OR'02, Le Croisic/France*, pages 93–108, 2002.

Others

J. SCHULZE, T. FAHLE. A Parallel Algorithm for Constrained Vehicle Routing Problems *Symposium on Combinatorial Optimization (CO'96)*, Imperial College, London, 1996.

T. FAHLE, J. SCHULZE. Parallelization Strategies for the Vehicle Routing Problem with Time Windows *International Conference on Global and Combinatorial Optimization, ICGCO*, Chania, Greece, 1998.

U. JUNKER, S. E. KARISCH, N. KOHL, B. VAABEN, T. FAHLE, AND M. SELLMANN. A Framework for Constraint Programming Based Column Generation *Sixteenth International Joint Conference on Artificial Intelligence - IJCAI, Workshop on Non Binary Constraints*, Stockholm, Sweden, 1999.

T. FAHLE. Scheduling of Ambulant Nursing Staff *International Workshop on Vehicle Routing*, Skodsborg, Denmark, August 2000.

T. FAHLE, U. JUNKER, S. E. KARISCH, N. KOHL, M. SELLMANN, AND B. VAABEN. Constraint Propagation for Complex Column Generation Subproblems *ISMP 2000 - 17th International Symposium on Mathematical Programming*, Atlanta/USA, August 2000.

[74] T. FAHLE, U. JUNKER, AND S. E. KARISCH (editors). *Integrating Constraint Programming, Artificial Intelligence, and Operations Research*, volume 115 of *Annals of Operations Research*. Kluwer Academic Publishers, 2002.

Submitted

[30] S. BERTELS AND T. FAHLE. A hybrid setup for a hybrid scenario: Combining heuristics for the home health care problem. *Annals of Operations Research*, submitted.

[72] T. FAHLE. Domain filtering for maximum clique. *Annals of Operations Research*, submitted.

Bibliography

[1] E. H. L. AARTS, F. M. J. DE BONT, E. H. A. HABERS, AND P. J. M. VAN LAARHOVEN. Parallel implementation of the statistical cooling algorithm. *INTEGRATION, the VLSI Journal*, 4(3):209–238, 1986.

[2] S. ABDENNADHER AND H. SCHLENKER. Nurse scheduling using constraint logic programming. In *Procs. of the 11th Conf. on Innovative Applications of Artificial Intelligence*, pages 838–843, Menlo Park, California, 1999. AAAI Press.

[3] R. ANBIL, E. GELMAN, B. PATTY, AND R. TANGA. Recent advantages in crew-pairing optimization at American Airlines. *Interfaces*, 21(1):62–74, 1991.

[4] E. ANDERSEN AND K. ANDERSEN. Presolving in linear programming. *Mathematical Programming*, 71:221–245, 1995.

[5] E. ANDERSSON, E. HOUSOS, N. KOHL, AND D. WEDELIN. Crew pairing optimization. In Yu [213], pages 228–258.

[6] Y. P. ANEJA, V. AGGARWAL, AND K. P. K. NAIR. Shortest chain subject to side constraints. *Networks*, 13:295–302, 1983.

[7] D. APPLEGATE AND D. JOHNSON. dfmax. URL `ftp://dimacs.rutgers.edu/pub/challenge/graph/solvers/`.

[8] K. R. APT. The rough guide to constraint programming. In *Proceedings of the CP'99*, volume 1713 of *LNCS*, pages 1–23. Springer, 1999.

[9] A. ATAMTÜRK, G. L. NEMHAUSER, AND M. W. P. SAVELSBERGH. Conflict graphs in solving integer programming problems. *European Journal on Operational Research*, 121(1):40–55, 2000.

[10] AXCENT AG. *mediaTV, technical description*. URL `http://www.axcent.de`.

[11] L. BABEL. Finding maximum cliques in arbitrary and in special graphs. *Computing*, 46:321–341, 1991.

[12] L. BABEL AND G. TINHOFER. A branch and bound algorithm for the maximal clique problem. *Methods of Operations Research*, 34:207–217, 1990.

[13] E. BALAS. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.

[14] E. Balas, S. Ceria, G. Couruéjols, and G. Pataki. Polyhedral methods for the maximum clique problem. In Johnson and Trick [122], pages 11–28.

[15] E. Balas and J. Xue. Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring. *Algorithmica*, 15:397–412, 1996.

[16] E. Balas and C. S. Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 14(4):1054–1068, 1986.

[17] E. Balas and E. Zemel. An algorithm for large-scale zero-one knapsack problems. *Operations Research*, 28:119–148, 1980.

[18] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint Based Scheduling – Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research and Management Science. Kluwer Academic Publisher, 2001.

[19] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[20] C. Barnhart and R. G. Shenoi. An approximate model and solution approach for the long-haul crew pairing problem. *Transportation Science*, 32(3):221–231, 1998.

[21] C. Barnhart, P. Vance, E. L. Johnson, and G. L. Nemhauser. Airline crew scheduling: A new formulation and decomposition algorithm. *Operations Research*, 43:188–200, 1997.

[22] N. Barnier and P. Brisset. Graph coloring for air traffic flow management. In *Proceedings of CP-AI-OR'02, Le Croisic/France*, pages 133–147, 2002.

[23] R. Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of WSD'99*, volume Part IV, pages 555–564, Prague, 1999.

[24] R. Barták. Theory and practice of constraint programming. In *Proceedings of CPDC'2001*, pages 7–14, Gliwice, Poland, 2001.

[25] J. E. Beasley. A lagrangian heuristic for set-covering problems. *Naval Research Logistics*, 37:151–164, 1990.

[26] J. E. Beasley. Lagrangean relaxation. In C. R. Reeves (editor), *Modern heuristic techniques for combinatorial problems*, pages 243–303. Wiley & Sons Ltd., 1993.

[27] S. V. Begur, D. M. Miller, and J. R. Weaver. An integrated spatial dss for scheduling and routing home-health-care nurses. *Interfaces*, 27(4):35–48, 1997.

[28] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. In C. Beierle and L. Plumer (editors), *Logic Programming: Formal Methods and Practical Applications*, pages 245–272. Elsevier, 1995.

[29] S. BERTELS. Integrierte Personal- und Tourenplanung am Beispiel ambulanter Krankenpflege. Diploma thesis, University of Paderborn, 2002.

[30] S. BERTELS AND T. FAHLE. A hybrid setup for a hybrid scenario: Combining heuristics for the home health care problem. *Annals of Operations Research*, submitted.

[31] C. BESSIÈRE. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65: 179–190, 1994.

[32] E. BEUTLER (editor). *Johann Wolfgang Goethe: Gedenkausgabe der Werke, Briefe und Gespräche*, volume 11: Italienische Reise. Artemis Verlags-AG Zürich, 2 edition, 1962.

[33] N. BIGGS. Some heuristics for graph coloring. In NELSON AND WILSON (editors), *Graph Colourings*, pages 87–96. Longman New York, 1990.

[34] R. E. BIXBY. A new generation of mixed integer programming codes. Invited Talk at CP-AI-OR'02, Le Croisic/France, March 23–24 2002. URL `http://cpaior.emn.fr:8000/`.

[35] R. E. BIXBY. Solving real-word linear programs: A decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[36] A. BOCKMAYR AND T. KASPER. Branch and infer:a unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10(3): 287–300, 1998.

[37] I. M. BOMZE, M. BUDINICH, P. M. PARDALOS, AND M. PELILLO. The maximum clique problem. In D.-Z. DU AND P. PARDALOS (editors), *Handbook of Combinatorial Optimization*, volume 4, pages 1–74. Kluwer Academic Publishers, 1999.

[38] R. BOPPANA AND M. M. HALLDÓRSSON. Approximating maximum independent sets by excluding subgraphs. *Bit*, 32:180–196, 1992.

[39] A. L. BOUTHILLIER, T. CRAINIC, AND R. K. KELLER. Parallel co-operative evolutionary algorithms for vehicle routing problems with time windows. Talk at PAREO 2000, Paderborn/Germany, 2000.

[40] D. BRÉLAZ. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[41] P. BRUCKER AND S. KNUST. A linear programming and constraint propagation-based lower bound for the RCPS. *European Journal on Operational Research*, 127:355–362, 2000.

[42] E. BURKE, P. DE CAUSMAECKER, AND G. VANDEN BERGHE. A hybrid tabu search algorithm for the nurse rostering problem. In *Simulated Evolution And Learning, Selected papers of SEAL'98*, volume 1585 of *LNAI*, pages 187–194. Springer, 1999.

[43] A. Caprara, M. Fischetti, and P. Toth. A heuristic algorithm for the set covering problem. In *Proceedings of IPCO'96*, volume 1084 of *LNCS*, pages 72–84. Springer, 1996.

[44] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. *Operations Research*, 47(5):730–743, 1999.

[45] A. Caprara, F. Focacci, E. Lamma, P. Mello, M. Milano, P. Toth, and D. Vigo. Integrating constraint logic programming and operations research techniques for the crew rostering problem. *Software – Practice and Experience*, 28(1):49–76, 1998.

[46] A. Caprara, D. Pisinger, and P. Toth. Exact solution of the quadratic knapsack problem. *INFORMS Journal on Computing*, 11(2):125–137, 1999.

[47] A. Caprara, P. Toth, D. Vigo, and M. Fischetti. Modeling and solving the crew rostering problem. *Operations Research*, 46(6):820–830, 1998.

[48] R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9:375–382, 1990.

[49] Y. Caseau and F. Laburthe. Heuristics for large constrained vehicle routing problems. *Journal of Heuristics*, 5:281–303, 1999.

[50] L. Cavique, C. Rego, and I. Themido. Subgraph ejection chains and tabu search for the crew scheduling problem. *Journal of the Operational Research Society*, 50: 608–616, 1999.

[51] B. M. W. Cheng, J. H. M. Lee, and J. C. K. Wu. A nurse rostering system using constraint programming and redundant modeling. *IEEE Transactions in Information Technology in Biomedicine*, 1(1):44–54, 1997.

[52] E. Cheng and J. L. Rich. A home health care routing and scheduling problem. Technical Report CAAM TR98-04, Rice University, 1998. (an earlier version was presented at *ISMP'97*).

[53] C. Chu and J. Antonio. Approximation algorithm to solve real-life multicriteria cutting stock problems. *Operations Research*, 47(4):495–508, 1999.

[54] H. D. Chu, E. Gelman, and E. L. Johnson. Solving large scale crew scheduling problems. *European Journal on Operational Research*, 97:260–268, 1997.

[55] V. Chvátal. *Linear Programming*. W. H. Freemann and Company, 1983.

[56] T. H. Cormen, C. E. Leierson, and R. L. Riverste. *Introduction to Algorithms*. McGraw-Hill, 1990.

[57] G. B. Dantzig. Discrete variable extremum problems. *Operations Research*, 5:266–277, 1957.

[58] G. B. DANTZIG. *Linear Programming And Extensions*. Princeton University Press, 1963.

[59] G. B. DANTZIG. Linear programming. *Operations Research*, 50(1):42–47, 2002. reprint of a 1991 paper.

[60] G. B. DANTZIG AND P. WOLFE. The decomposition algorithm for linear programs. *Econometrica*, 29(4):767–778, 1961.

[61] P. R. DAY AND D. M. RYAN. Flight attendant rostering for short-haul airline operations. *Operations Research*, 45(5):649–661, 1997.

[62] R. S. DEMBO AND P. L. HAMMER. A reduction algorithm for knapsack problems. *Methods of Operations Research*, 36:49–60, 1980.

[63] G. DESAULNIERS, J. DESROSIERS, Y. DUMAS, S. MARC, B. RIOUX, M. M. SOLOMON, AND F. SOUMIS. Crew pairing at Air France. *European Journal on Operational Research*, 97:245–259, 1997.

[64] M. DESROCHERS AND F. SOUMIS. A generalized permanent labeling algorithm for the shortest path problem with time windows. *INFOR*, 26(3):191–212, 1988.

[65] J. DESROSIERS, Y. DUMAS, M. M. SOLOMON, AND F. SOUMIS. Time constrained routing and scheduling. In *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*, pages 35–139. North-Holland, 1995.

[66] DEUTSCHE LUFTHANSA AG. Konzernbericht Januar – Juni 2002. URL `http://www.lufthansa-financials.com`.

[67] Dimacs. Dimacs clique benchmark instances, 1993. URL `ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/`.

[68] A. C. DOYLE. *The Sign of Four*, chapter 6. Penguin Classics, 2001.

[69] M. ELF, C. GUTWENGER, M. JÜNGER, AND G. RINALDI. Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS. In Jünger and Naddef [124], pages 157–223.

[70] T. FAHLE. Cost based filtering vs. upper bounds for maximum clique. In *Proceedings of CP-AI-OR'02, Le Croisic/France*, pages 93–108, 2002.

[71] T. FAHLE. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In *Proceedings of ESA'02, Rome/Italy*, volume 2461 of *LNCS*, pages 485–498. Springer, 2002.

[72] T. FAHLE. Domain filtering for maximum clique. *Annals of Operations Research*, submitted.

[73] T. FAHLE, U. JUNKER, S. KARISCH, N. KOHL, M. SELLMANN, AND B. VAABEN. Constraint programming based column generation for crew assignment. *Journal of Heuristics*, 8(1):59–81, 2002.

[74] T. FAHLE, U. JUNKER, AND S. E. KARISCH (editors). *Integrating Constraint Programming, Artificial Intelligence, and Operations Research*, volume 115 of *Annals of Operations Research*. Kluwer Academic Publishers, 2002.

[75] T. FAHLE, S. SCHAMBERGER, AND M. SELLMANN. Symmetry breaking. In *Proceedings of the CP'01*, volume 2239 of *LNCS*, pages 93–107. Springer, 2001.

[76] T. FAHLE AND M. SELLMANN. Constraint programming based column generation with knapsack subproblems. In *Proceedings of the CP-AI-OR'00, Paderborn/Germany*, pages 33–44, 2000. full version in [77].

[77] T. FAHLE AND M. SELLMANN. Cost based filtering for the constrained knapsack problem. *Annals of Operations Research*, 115:73–94, 2002.

[78] D. FAYARD AND G. PLATEAU. An algorithm for the solution of the 0-1 knapsack problem. *Computing*, 28:269–287, 1983.

[79] M. L. FISCHER. The lagrangean relaxation method for solving integer programming problems. *Management Science*, 27(1):1–18, 1981.

[80] M. L. FISHER. Vehicle routing. In M. O. BALL, T. L. MAGNANTI, C. L. MONMA, AND G. L. NEMHAUSER (editors), *Network Routing*, volume 8 of *Handbooks on Operations Research and Management Science*, pages 1–33. North-Holland, Amsterdam, 1995.

[81] F. FOCACCI, A. LODI, AND M. MILANO. Cost-based domain filtering. In *Proceedings of the CP'99*, volume 1713 of *LNCS*, pages 189–203. Springer, 1999.

[82] F. FOCACCI, A. LODI, AND M. MILANO. Cutting planes in constraint programming: An hybrid approach. In *Proceedings of the CP-AI-OR'00, Paderborn/Germany*, pages 45–51, 2000.

[83] D. R. FULKERSON. Blocking and anti-blocking pairs of polyhedral. *Mathematical Programming*, 1:168–194, 1971.

[84] M. GAMACHE, F. SOUMIS, D. VILLENEUVE, D. J., AND E. GÉLINAS. The preferential bidding system at Air Canada. *Transportation Science*, 32(3):246–255, 1998.

[85] M. R. GAREY AND D. S. JOHNSON. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[86] M. GENDREAU, G. LAPORTE, AND J.-Y. POTVIN. Vehicle routing: modern heuristics. In E. H. L. AARTS AND J. K. LENSTRA (editors), *Local Search in Combinatorial Optimization*, pages 311–336. John Wiley & Sons, 1997.

[87] A. M. GEOFFRION. Lagrangean relaxation for integer programming. *Mathematical Programming Study*, 2:82–114, 1974.

[88] P. C. GILMORE AND R. E. GOMORY. A linear programming approach to the cutting stock problem. *Operations Research*, 9:849–859, 1961.

[89] F. GLOVER. Surrogate constraints. *Operations Research*, 13:741–749, 1968.

[90] F. GLOVER. Tabu Search—Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[91] F. GLOVER. Tabu Search—Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[92] J.-L. GOFFIN. On the convergence rate subgradient optimization methods. *Mathematical Programming*, 13:329–347, 1977.

[93] GOLDBERG. *Genetic Algorithms in search, optimization and machine learning*. Addison Wesley, 1984.

[94] M. C. GOLUMBIC. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press New York, 1991.

[95] H. G. GRÄF. *Goethe über seine Dichtungen: Versuch einer Sammlung aller Äußerungen des Dichters über seine poetischen Werke*, page 976. Rütten & Loening, Nachdruck im Verlag d. Wiss. Buchges. Darmstadt, 1965.

[96] G. W. GRAVES, R. D. MCBRIDE, I. GERSHKOFF, D. ANDERSON, AND D. MAHIDHARA. Flight crew scheduling. *Management Science*, 39(6):736–745, 1993.

[97] H. J. GREENBERG AND W. P. PIERSKALLA. Surrogate mathematical programming. *Operations Research*, 18:924–939, 1968.

[98] M. GRÖTSCHEL, L. LOVÁSZ, AND A. SCHRIJVER. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[99] M. GRÖTSCHEL, L. LOVÁSZ, AND A. SCHRIJVER. Relaxations of vertex packing. *Journal of Combinatorial Theory B*, 40:330–343, 1986.

[100] Z. GU, G. L.NEMHAUSER, AND M. W. P. SAVELSBERGH. Lifted cover inequalities for 0-1 integer programs: Computation. *INFORMS Journal on Computing*, 10(4):427–437, 1998.

[101] Z. GU, G. L. NEMHAUSER, AND M. W. P. SAVELSBERGH. Lifted cover inequalities for 0-1 integer programs: Complexity. *INFORMS Journal on Computing*, 11(1):117–123, 1999.

[102] M. GUIGNARD. Efficient cuts in lagrangean 'relax-and-cut' schemes. *European Journal on Operational Research*, 105:216–223, 1998.

[103] U. I. GUPTA, D. T. LEE, AND Y. Y.-T. LEUNG. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12:459–467, 1982.

[104] P. HAMMER, E. L. JOHNSON, AND U. N. PELED. Facets of regular 0-1 polytopes. *Mathematical Programming*, 8:179–206, 1975.

[105] W. D. HARVEY AND M. L. GINSBERG. Limited discrepancy search. In *Proceedings of IJCAI'95*, pages 607–613. Morgan Kaufmann, 1995.

[106] J. HÅSTAD. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, 182:105–142, 1999.

[107] S. HEIPCKE. Comparing constraint programming and mathematical programming approaches to discrete optimisation – the change problem. *Journal of the Operational Research Society*, 50:581–595, 1999.

[108] M. HELD AND R. M. KARP. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[109] M. HELD AND R. M. KARP. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1:6–25, 1971.

[110] M. HELD, P. WOLFE, AND H. P. CROWDER. Validation of subgradient optimization. *Mathematical Programming*, 6:62–88, 1974.

[111] K. L. HOFFMAN AND M. PADBERG. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39(6):657–682, 1993.

[112] J. HOOKER. Unifying optimization and constraint satisfaction, invited talk at IJCAI '99, 1999. URL `http://ba.gsia.cmu.edu/jnh/ijcai.ppt`.

[113] P. D. HUDSON. Improving the branch and bound algorithm for the knapsack problem. Research report, Queen's University, Belfast, 1977.

[114] *Ilog Cplex V6.5 Reference manual and User manual*. ILOG, 1999.

[115] *Ilog Planner V3.0 Reference manual and User manual*. ILOG, 1999.

[116] *Ilog Solver V4.4 Reference manual and User manual*. ILOG, 1999.

[117] *Ilog Solver V5.0 Reference manual and User manual*. ILOG, 2000.

[118] *Ilog Cplex V7.1 Reference manual and User manual*. ILOG, 2001.

[119] *Ilog Cplex V7.5 Reference manual and User manual*. ILOG, 2002.

[120] *Ilog Solver V5.2 Reference manual and User manual*. ILOG, 2002.

[121] G. P. INGARGIOLA AND J. F. KORSH. A reduction algorithm for zero-one single knapsack problems. *Management Science*, 20:460–463, 1973.

[122] D. S. JOHNSON AND M. A. TRICK (editors). *Cliques, Colorings and Satisfiability – 2nd DIMACS Implementation Challenge*. American Mathematical Society, 1996.

[123] E. L. JOHNSON, G. L. NEMHAUSER, AND M. W. P. SAVELSBERGH. Progress in linear programming based branch-and-bound algorithms: An exposition. *INFORMS Journal on Computing*, 12:2–23, 2000.

[124] M. JÜNGER AND D. NADDEF (editors). *Computational Combinatorial Optimization – Optimal or Provably Near-Optimal Solutions*, volume 2241 of *LNCS*. Springer, 2001.

[125] U. Junker, S. E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. A framework for constraint programming based column generation. In *Proceedings of CP'99*, volume 1713 of *LNCS*, pages 261–274. Springer, 1999.

[126] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[127] R. M. Karp. Reducibility among combinatorial problems. In Miller and Thatcher (editors), *Complexity of Computer Computations*, pages 85–103. Plenum Press, NewYork, 1972.

[128] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[129] V. Klee and G. J. Minty. How good is the simplex method? In O. Shisha (editor), *Inequalities III*, pages 159–175. Academic Press, New York, 1972.

[130] G. Kliewer, K. Klohs, and S. Tschöke. Parallel simulated annealing library (parsa): User manual. Technical report, University of Paderborn, 1999.

[131] N. Kohl and S. E. Karisch. Airline crew assignment: modeling and optimization. Technical report, Carmen System, 1999.

[132] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[133] A. D. Land and A. G. Doig. The decomposition algorithm for linear programs. *Econometrica*, 28(3):497–520, 1960.

[134] C. Lemaréchal. Lagrangian relaxation. In Jünger and Naddef [124], pages 112–156.

[135] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2): 173–186, 1999.

[136] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[137] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8 (1):99–118, 1977.

[138] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[139] I. Maros and G. Mitra. Simplex algorithms. In J. E. Beasley (editor), *Advances in Linear and Integer Programming*, pages 1–46. Oxford Scientific Publications, 1996.

[140] K. Marriot and P. J. Stuckey. *Programming with Constraints*. MIT Press, Cambridge, Massachusetts, 1998.

[141] S. MARTELLO, D. PISINGER, AND P. TOTH. Dynamic programming and tight bounds for the 0-1 knapsack problem. *Management Science*, 45:414–424, 1999.

[142] S. MARTELLO AND P. TOTH. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. *European Journal on Operational Research*, 1: 169–175, 1977.

[143] S. MARTELLO AND P. TOTH. Solution of the zero-one multiple knapsack problems. *European Journal on Operational Research*, 4:276–283, 1981.

[144] S. MARTELLO AND P. TOTH. A new algorithm for the 0-1 knapsack problem. *Management Science*, 34:633–644, 1988.

[145] S. MARTELLO AND P. TOTH. *Knapsack Problems – Algorithms and Computer Implementations*. Wiley Interscience, 1990.

[146] S. MARTELLO AND P. TOTH. Upper bounds and algorithms for hard 0-1 knapsack problems. *Operations Research*, 45(5):768–778, 1997.

[147] A. MARTIN. General mixed integer programming: Computational issues for branch-and-cut algorithms. In Jünger and Naddef [124], pages 1–25.

[148] A. J. MASON AND M. C. SMITH. A nested column generator for solving rostering problems with integer programming. In L. CACCETTA, K. L. TEO, P. F. SIEW, Y. H. LEUNG, L. S. JENNINGS, AND V. REHBOCK (editors), *Int. Conf. on Optimisation : Techniques and Applications*, pages 827–834, 1998.

[149] K. MEHLHORN. Constraint programming and graph algorithms. In *ICALP 2000*, volume 1853 of *LNCS*, pages 571–575. Springer, 2000.

[150] A. MEHROTRA AND M. TRICK. A column generation approach for exact graph coloring. *INFORMS Journal on Computing*, 8:344–354, 1996.

[151] M. MILANO. Integration of mathematical programming and constraint programming for combinatorial optimization problems. Tutorial at CP'00, Singapore, 2000.

[152] R. MOHR AND T. C. HENDERSON. Arc and path consistency revised. *Artificial Intelligence*, 28:225–233, 1986.

[153] U. MONTANARI. Networks of constraints: fundamental properties and applications. *Information Science*, 7(2):95–132, 1974.

[154] G. L. NEMHAUSER AND L. E. TROTTER JR. Properties of vertex packing and independence system polyhedral. *Mathematical Programming*, 6:48–61, 1974.

[155] G. L. NEMHAUSER AND L. E. TROTTER JR. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.

[156] G. L. NEMHAUSER AND L. A. WOLSEY. *Integer and Combinatorial Optimization*. Wiley Interscience, 1988.

[157] I. H. OSMAN. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41:421–451, 1993.

[158] P. R. J. ÖSTERGÅRD. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120:197–207, 2002.

[159] G. OTTOSSON AND E. S. THORSTEINSSON. Linear relaxation and reduced-cost based propagation of continuous variable subscripts. *Annals of Operations Research*, 115: 207–226, 2002.

[160] M. PADBERG. On the facial structure of set packing polyhedral. *Mathematical Programming*, 5:199–215, 1973.

[161] M. PADBERG. A note on zero-one programming. *Operations Research*, 23:833–837, 1975.

[162] M. PADBERG. Classical cuts for mixed-integer programming and branch-and-cut. *Mathematical Methods of Operations Research*, 53(2):173–203, 2001.

[163] C. H. PAPADIMITRIOU AND K. STIEGLITZ. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[164] PARPAP. Partizipative Personaleinsatzplanung für den ambulanten Pflegedienst. Vorhabensbeschreibung, BMBF-Projekt 01HR9955, 1999.

[165] PARROT. Executive summary. Project report, ESPRIT-Project 24 960, 1997.

[166] G. PESANT AND M. GENDREAU. A view of local search in constraint programming. In *Proceedings of the CP'96*, volume 1118 of *LNCS*, pages 353–356. Springer, 1996.

[167] G. PESANT AND M. GENDREAU. A constraint programming framework for local search methods. *Journal of Heuristics*, 5(3):255–279, 1999.

[168] D. PISINGER. An expanding-core algorithm for the exact 0-1 knapsack problem. *European Journal on Operational Research*, 87:175–187, 1995.

[169] D. PISINGER. An exact algorithm for large multiple knapsack problem. *European Journal on Operational Research*, 114:528–541, 1999.

[170] O. PORTO, M. DE MORAES, AND L. LUCENA. A relax and cut algorithm for the quadratic knapsack problem. In *Proceedings of ISMP'00, Atlanta/USA*, 2000.

[171] S. PRESTWICH. Combining the scalability of local search with the pruning techniques of systematic search. *Annals of Operations Research*, 115:51–72, 2002.

[172] J.-C. RÉGIN. A filtering algorithm for constraints of difference in CSPs,. In *Proceedings of AAAI-94*, pages 362–367. MIT Press, 1994.

[173] J.-C. RÉGIN. Generalized arc consistency for global cardinality constraint. In *Proceedings of AAAI-96*, pages 209–215, 1996.

[174] J. M. ROBSON. Algorithms for maximum independent sets. *Journal of Algorithms*, 7: 425–440, 1986.

[175] Y. ROCHAT AND E. D. TAILLARD. Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics*, 1:147–167, 1995.

[176] R. RODOSEK, M. WALLACE, AND M. T. HAIJAN. A new approach to integrating mixed integer programming and constraint logic programming. *Annals of Operations Research*, 86:63–87, 1999.

[177] L.-M. ROUSSEAU, F. FOCACCI, M. GENDREAU, AND G. PESANT. Solving VRPTWs with constraint programming based column generation, 2002. (submitted) see also [178].

[178] L.-M. ROUSSEAU, M. GENDREAU, AND G. PESANT. Solving small VRPTWs with constraint programming based column generation. In *Proceedings of CP-AI-OR'02, Le Croisic/France*, pages 333–344, 2002.

[179] L.-M. ROUSSEAU, M. GENDREAU, AND G. PESANT. Using constraint logic-based operators to solve vehicle routing problems with time windows. *Journal of Heuristics*, 8(1):43–58, 2002.

[180] L.-M. ROUSSEAU, G. PESANT, AND M. GENDREAU. A general approach to the physician rostering problem. *Annals of Operations Research*, 115:193–205, 2002.

[181] R. A. RUSHMEIER, K. L. HOFFMAN, AND M. PADBERG. Recent advances in exact optimization of airline scheduling problems. Technical report, George Mason University, 1995.

[182] D. M. RYAN. The solution of massive generalized set partitioning problems in aircrew rostering. *Journal of the Operational Research Society*, 43(5):459–467, 1992.

[183] M. W. P. SAVELSBERGH. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[184] J. SCHULZE AND T. FAHLE. A parallel algorithm for the vehicle routing problem with time window constraints. *Annals of Operations Research*, 86:585–607, 1999.

[185] M. SELLMANN AND T. FAHLE. Constraint programming based lagrangian relaxation for a multimedia application. In *Proceedings CP-AI-OR'01, Ashford/UK*, pages 1–14, 2001. full version in [187].

[186] M. SELLMANN AND T. FAHLE. Coupling variable fixing algorithms for the automatic recording problem. In *Proceedings of ESA'01, Århus/Denmark*, volume 2161 of *LNCS*, pages 134–145. Springer, 2001.

[187] M. SELLMANN AND T. FAHLE. Constraint programming based lagrangian relaxation for the automatic recording problem. *Annals of Operations Research*, 118:17–33, 2003 (to appear).

[188] M. SELLMANN, K. ZERVOUDAKIS, P. STAMATOPOULOS, AND T. FAHLE. Integrating direct CP search and CP-based column generation for the airline crew assignment problem. In *Proceedings of the CP-AI-OR'00, Paderborn/Germany*, pages 163–170, 2000.

[189] M. SELLMANN, K. ZERVOUDAKIS, P. STAMATOPOULOS, AND T. FAHLE. Crew assignment via constraint programming: Integrating column generation and heuristic tree search. *Annals of Operations Research*, 115:207–226, 2002.

[190] P. SHAW. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of CP'98*, volume 1520 of *LNCS*, pages 417–431. Springer, 1998.

[191] P. SHAW, B. DE BACKER, AND V. FURNON. Improved local search for cp toolkits. *Annals of Operations Research*, 115:31–50, 2002.

[192] B. SMITH. A tutorial on constraint programming. Technical Report 95.14, University of Leeds, 1995.

[193] M. M. SOLOMON. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, March–April 1987.

[194] P. STAMATOPOULOS, G. BOUKEAS, V. ZERVOUDAKIS, K. STOUMPOS, AND C. HALATSIS. Parallel CP-based direct crew rostering. PARROT deliverable d-tec2.1, University of Athens, 1999.

[195] É. D. TAILLARD, P. BADEAU, M. GENDREAU, F. GUERTIN, AND J.-Y. POTVIN. A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation Science*, 31:170–186, 1997.

[196] É. D. TAILLARD, L. M. GAMBARDELLA, M. GENDREAU, AND J.-Y. POTVIN. Adaptive memory programming: A unified view of metaheuristics. *European Journal on Operational Research*, 135(1):1–16, 2001.

[197] R. E. TARJAN AND A. E. TROJANOWSKI. Finding a maximum independent set. *SIAM Journal on Computing*, 6:537–546, 1977.

[198] P. TOTH AND D. VIGO (editors). *The Vehicle Routing Problem*. SIAM Monographs on Discrete Mathematics and Applications, 2002.

[199] M. TRICK. A dynamic programming approach for consistency and propagation for knapsack constraints. In *Proceedings of CP-AI-OR'01, Ashford/UK*, pages 113–124, 2001.

[200] M. TRICK. Constraint programming – a tutorial. Tutorial at the CP-AI-OR'02 School on Optimization, Le Croisic/France, March 23–24 2002. URL `http://mat.gsia.cmu.edu/trick/cp.ppt`.

[201] E. P. K. TSANG. *Foundations of Constraint Programming*. Academic Press, 1993.

[202] P. Van Hentenryck, Y. Deville, and M. C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[203] P. J. M. van Laarhoven and E. Aarts. *Simulated Annealing: Theory and Application*. D. Reidel Publishing Company, 1987.

[204] G. Verfaillie, M. Lemaître, and T. Schiex. Russian doll search for solving constraint satisfaction problems. In *Proceedings of AAAI-96*, pages 181–187, 1996.

[205] P. R. C. Villela and C. T. Bornstein. An improved bound for the 0-1 knapsack problem. Technical report, COPPE-Federal University of Rio de Janeiro, 1983.

[206] T. Walsh. Depth-bounded discrepancy search. In *Proceedings of the IJCAI'97*, pages 1388–1393, 1997.

[207] D. Waltz. Understanding line drawings of scenes with shadows. In P. H. Winston (editor), *The Psychology of Computer Vision*, pages 19–91. McGraw Hill, 1975.

[208] D. Wedelin. An algorithm for large scale 0-1 integer programming with application to airline crew scheduling. *Annals of Operations Research*, 57:283–301, 1995.

[209] H. P. Williams. *Model building in mathematical programming*. Wiley, 4th edition, 2001.

[210] L. A. Wolsey. Faces for a linear inequality in 0-1 variables. *Mathematical Programming*, 8:165–178, 1975.

[211] L. A. Wolsey. *Integer Programming*. Wiley Interscience, 1998.

[212] D. R. Wood. An algorithm for finding a maximum clique in a graph. *Operations Research Letters*, 21:211–217, 1997.

[213] G. Yu (editor). *Operations Research in the Airline Industry*, volume 9 of *International Series in Operations Research and Management Science*. Kluwer Academic Publishers, 1998.

So eine Arbeit wird eigentlich nie fertig,
man muß sie für fertig erklären,
wenn man nach Zeit und Umständen
das möglichste getan hat.[6]

Johann Wolfgang Goethe (1749 – 1832),
Italienische Reise, 16. März 1787 [32].

[6]Such work is never finished; one must declare it so when, according to time and circumstances, one has done one's best. (non-authorized translation)