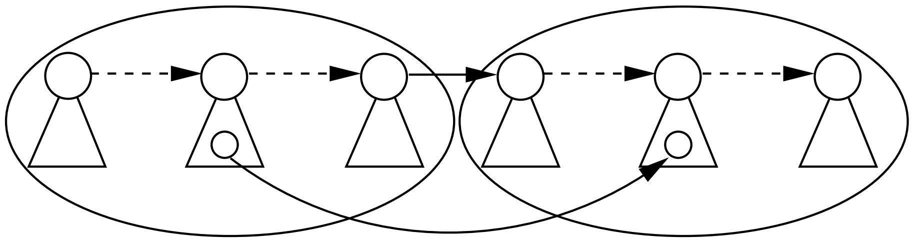
Universität Paderborn
Fachbereich Mathematik/Informatik

Reduction Techniques

in Constraint Programming and Combinatorial Optimization



Dissertation

von

Meinolf Sellmann

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Paderborn, im August 2002.

Für Olga.

Dank

Diese Dissertation entstand in den vergangenen vier Jahren während meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Arbeitsgruppe Monien im Fachbereich Mathematik/Informatik an der Universität Paderborn. Sie wäre ohne die Unterstützung und Hilfe einer Vielzahl von Wissenschaftlern, Kollegen, Freunden und Verwandten in dieser Form nicht möglich gewesen. Bei ihnen möchte ich mich an dieser Stelle bedanken.

Zuerst gilt mein besonderer und herzlicher Dank Prof. Dr. Burkhard Monien, der diese Arbeit mit seiner großen wissenschaftlichen Kompetenz und viel Wohlwollen sowohl inhaltlich als auch menschlich betreut hat. Die vielfältigen Themen und Projekte, die an seinem Lehrstuhl aktiv verfolgt werden, bieten einen außerordentlichen Einblick in das aktuelle Forschungsgeschehen, für den ich ihm sehr dankbar bin. Darüber hinaus hat mich über die vergangenen Jahre die Gewissheit seiner Unterstützung getragen, die ebenso Leistung wie Bescheidenheit fordert und fördert.

Für ihre Liebe, ihren Rat in schwierigen Situationen, ihre Geduld und ihren Glauben an mich bedanke ich mich ganz besonders bei meiner Freundin Olga. Ohne ihren Rückhalt und ihre Hilfe wäre diese Arbeit nicht gelungen. Dasselbe gilt auch für meine Familie, meine Eltern und Brüder, die mich stets nach Kräften unterstützen und meine Entwicklung fördern.

Mein weiterer Dank für die gute wissenschaftliche Zusammenarbeit gilt meinen Kollegen in der Arbeitsgruppe, im EU-Projekt PARROT, im DFG-Sonderforschungsbereich Massive Parallelität, im EU-Projekt UP-TV und im DFG-Schwerpunktprogramm Algorithmik großer und komplexer Netzwerke. Viele der in dieser Arbeit dargestellten Ergebnisse sind in diesen Forschungsgruppen durch einen intensiven Ideenaustausch entstanden. Ausdrücklich nennen möchte ich Torsten Fahle, Dr. Warwick Harvey, Georg Kliewer, Norbert Sensen und Kyriakos Zervoudakis.

Schließlich bedanke ich mich noch sehr herzlich bei Dr. Michael Laska für seine Unterstützung als Projektmanager der AG und bei Ulrich Ahlers, Sigrid Gundelach, Marion Rohloff und Thomas Thissen für die unermüdliche Hilfe bei organisatorischen und technischen Problemen.

Vielen Dank!

Paderborn, im August 2002. *Meinolf Sellmann*

Contents

Chapter 1

Introduction

1.1 A World of Optimization

On all levels, the world we live in is full of optimization problems and processes. Many macroscopic physical and chemical phenomena can be explained by the theory that nature tries to reach a state of minimum enthalpy. Also, every competition is inherently associated with a measure of success that can be optimized. Biological life itself is based on competition, and the law of evolution favors efficiency and adaptiveness. The same principle determines our economic system that is based on the egoistic striving of every economic subject to maximize its wealth. Therefore, to forecast natural processes and to be economically competitive, optimization problems have to be solved.

The progress that is made in algorithmic computer science can help to take up this challenge. Nowadays, most computers that are sold are used to edit texts, to manage large amounts of data, and to provide access to the Internet. On the other hand, provided with state-of-the-art optimization software, even a simple personal computer has the potential to become a valuable tool for the simulation of natural processes, efficient production and decision support in a progressive, up-to-date company. However, the technology push is dragging and the solutions that algorithmic computer science offers are only laboriously transferred into practice. Especially small and medium sized companies often cannot afford the risk of investing a considerable amount of money into the development of company-specific software that may soon turn out to be over-specialized and too rigid to ensure a return on investment in the perpetually changing environments of a globalized economy.

Of course, commercial optimization software that efficiently supports a stable, non-malleable process by solving a general optimization problem — like Job-Shop Problems, for example — has a chance to successfully find its way into the industry. Due to highly dynamic production conditions, the number of such applications is rather limited, though, and there is certainly a need for flexible and also company specific optimization software. For this purpose, there are general solvers and software libraries available. Outstanding examples are ILOG SOLVER [121], ECLIPSE [58], ABACUS [127], LEDA [153], ILOG CPLEX [118], and MINTO [159]. However, with respect to the wide range of applications that they potentially address, even these comparably successful tools could reach a much larger market than the one that is currently serviced. A major obstacle for a broader use of standard optimization software is a lack of expertise outside the scientific world. In the current situation, we observe that the more powerful a solver is, the more knowledge is necessary to use it successfully. Therefore, to ease the handling of optimization software, the modeling of real-world problems has to become more intuitive, and its influence on the efficiency of the solution process must be reduced.

1.2 Modeling and Efficiency

Consider the situation in mathematical programming. The user is forced to crush a possibly well-structured problem into a set of very basic linear and integer constraints. If this modeling process is carried out carefully, standard mixed integer program solvers like CPLEX can tackle many problems with an astounding efficiency. However, to set up an efficiently solvable integer program is an art, it requires much experience and is not at all an easy task [215]. In constraint programming, the situation with respect to problem modeling is much more comfortable. Nowadays constraint programming solvers offer sets of predefined, so-called *symbolic constraints* that reflect the user's intuition much better than linear programs. However, even though the modeling is easier, due to the loose connection of constraints, the optimization abilities of constraint programming solvers are much more limited than those based on mathematical programming. To a large extent, the lack of efficiency is caused by the fact that unfavorable regions of the search space are being explored unnecessarily, which could be avoided by using a tight global bound on the objective.

In order to overcome the obstacle of complicated problem modeling in mathematical programming, work is in progress that tries to provide the user with higher lever building blocks that can be used to describe a problem. The SCIL-library [186] for branch-and-cut-and-price approaches is an excellent example of this attempt. Using a description language that provides constructs in the style of symbolic constraints, the user can set up a problem model. Not only does this make the modeling process easier, but on top of that, the solver is no longer provided with only a set of dis-aggregated linear constraints, but is made aware of the basic substructures of a problem. Therefore, it is able to exploit specific knowledge about these substructures, for example by adding global cuts to a problem that are valid for one of the polytopes that are intersected. This may improve the quality of problem relaxations.

On the other hand, to improve the optimization abilities of constraint programming solvers, there is a remarkable effort visible that tries to incorporate the merits of mathematical programming into constraint programming machineries. The ILOG CONCERT technology, for example, combines ILOG CPLEX and ILOG SOLVER. On a broader scale, a considerable number of researchers work on the integration of methods from operations research into constraint programming. Particularly, since 1999, the International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR) has become a major terminal for the exchange of ideas between the two research areas. Even though combinatorial optimization and constraint programming cover a widespread area of research topics and go far beyond simple enumeration techniques, tree search has been identified and used as a key juncture between the two fields.

In combinatorial optimization, following the branch & bound paradigm, tree search is used to solve hard optimization problems exactly, a task that consists of computing a feasible solution and proving its optimality. Upper and lower bounding routines form the beating heart of every branch & bound algorithm: primal heuristics or approximation algorithms are used to find near-optimal solutions quickly, while relaxations overestimate the best performance of any solution that can be found in a given sub-tree. Of course, if that estimate is worse than the performance of the best known, so-called *incumbent solution*, the corresponding sub-tree does not need to be explored any further and can be pruned.

On the other hand, in constraint programming, tree search is used to overcome the incompleteness of a pure inference calculus that achieves a state of local consistency only. In every search node, the finite domains of the variables of the CP-model are reduced with respect to the model's constraints; a process called constraint propagation. Sequentially, constraints are propagated until a state of the domains is reached that achieves the desired degree of consistency. Only then, a case distinction takes place and a branching step is carried out. This way, constraints interact only via the domains of their variables, which makes the approach extremely flexible with respect to the addition or removal of constraints.

1.3 Contribution

In this thesis, we develop reduction techniques for combinatorial optimization and constraint satisfaction problems that can be embedded in a tree search approach. In combinatorial optimization, bound estimates and variable fixing algorithms are commonly used for that purpose, whereas in constraint programming filtering algorithms undertake the task of shrinking the search space by eliminating values from variable domains. The algorithms that we develop are meant to be used as symbolic constraints in constraint programming solvers and also in optimization software that provides high level description constructs like the SCIL-library. Therefore, we consider the work done in this thesis as a contribution toward the development of efficient and easy to use optimization software.

Note that, when solving discrete optimization problems to optimality, there are really two tasks to be considered. First, an optimal solution must be constructed, and second, its optimality must be proven. Optimal or at least near-optimal solutions can often be found quickly by heuristics or by approximation algorithms, both specially tailored for the given problem. In contrast to the construction of a high quality solution, the algorithmic optimality proof requires the investigation of the entire search space, which in general is much harder than to partly explore the most promising regions only. By eliminating parts of the search space that do not contain improving solutions, problem reduction can help with respect to both aspects of discrete optimization.

1.3.1 Outline and Major Results

The thesis is organized in two parts. Part I consists of the Chapters 2–4 and is method oriented. This means that the task of achieving a certain degree of consistency for some special filtering problems is studied theoretically. The efficiency of the algorithms developed is measured in terms of worst case complexity and the degree of consistency that they achieve.

The first type of reduction algorithm that we develop involves a special kind of symbolic constraint: In Chapter 2, our goal is to provide a set of so-called *optimization constraints*. By linking the objective function with the constraint structure of a problem, such constraints can be used for pruning and problem reduction with respect to cost considerations, a process called *cost-based filtering*. That way, optimization constraints naturally combine the optimization abilities of operations research algorithms and the efficient modeling and filtering concepts of constraint programming.

Particularly, we study the problem of achieving different degrees of consistency for optimization constraints. Since achieving a state of hyper-arc-consistency may turn out to be an NP-hard problem itself, we introduce a new type of consistency for optimization constraints, so-called *relaxed consistency*. Based on the two concepts, we develop efficient cost-based filtering algorithms for shortest path constraints (on directed acyclic graphs, undirected graphs with non-negative edge weights, and directed graphs without negative cycles), weighted stable set constraints in interval graphs, weighted all-different constraints, and knapsack constraints. These constraints are supposed to be used as basic building blocks when modeling real-life discrete optimization problems. By exploiting the knowledge of the given constraint structures, the corresponding reduction algorithms make use of previously developed bounds and the efficient ways known to compute them.

As we shall see, the loose connection of optimization constraints via variable domains results in less effective and thus also less efficient problem reduction. Therefore, in Chapter 3, we present a theory that motivates the linking of optimization constraints via the standard operations research decomposition techniques column generation and Lagrangian relaxation.

Then, a second type of reduction algorithm is developed that bases its decisions on the constraint structure of a problem rather than on cost considerations. Obviously, a search node does not need to be expanded if it represents a previously considered configuration. However, this situation occurs frequently when tackling problems that contain symmetry. In Chapter 4, we present a general symmetry breaking method called SBDD that is based on dominance detection between choice points. An experimental evaluation shows that the method is better suited to tackle highly symmetric problems than previously developed symmetry breaking techniques.

Part II of the thesis covers the Chapters 5–9 and is application oriented. Several combinatorial optimization and constraint satisfaction problems are investigated. The approaches that we develop are based on the algorithms and methods from Part I. This allows an empirical evaluation of the previously developed reduction algorithms on top of the theoretical work done in the first part.

In particular, we consider the Airline Crew Assignment Problem in Chapter 5. The approach presented is based on the concept of CP-based column generation in combination with shortest path constraints. By exploiting CP and OR specific advantages, we are able to speed up the computation of real-world airline crew schedules considerably. The ideas that we present have been integrated in an industrial airline crew assignment software system and have yielded drastic savings in running time.

In Chapter 6, we study the Automatic Recording Problem, that evolves in the context of modern multimedia applications. After giving an approximation scheme for the NP-hard problem, an exact algorithmic approach is presented that links knapsack constraints and weighted stable set constraints on interval graphs following the idea of CP-based Lagrangian relaxation. Numerical results show that our implementation is efficient enough to tackle real-size problem instances in an amount of time that is well affordable in practice.

The Capacitated Network Design Problem is tackled in Chapter 7. Lower bounds can be computed by decomposing the problem. We review previously developed reduction techniques and use CP-based Lagrangian relaxation to link them together. Moreover, a new technique is presented that adds locally valid cuts based on a Lagrangian relaxation of the problem. In our experiments, we show that a heuristic version of our potentially exact solver is able to provide solutions of higher quality in less time than the best known heuristic techniques known so far.

A new approach for the Social Golfer Problem is developed in Chapter 8. Using SBDD for symmetry breaking and the new idea of heuristic constraint propagation, we are able to solve problems that were previously out of reach for solvers based on constraint programming.

Finally, in Chapter 9, we develop a solver for the Graph Bisection Problem. The core of the algorithm is a lower bounding procedure that approximates maximum multicommodity flows with multiple sinks. Comparisons with a previously developed bound based on semi-definite programming show the gains in quality and computation time on sparse, structured graphs. Especially, our implementation is the first to compute the bisection widths of DeBruijn 9, Shuffle-Exchange 9, and Shuffle-Exchange 10.

1.3.2 Background

To a large extent, the thesis is self-contained. However, we assume that the reader is familiar with the basic concepts of algorithm and complexity theory (dynamic programming, NP-completeness, approximation schemes, etc.), operations research (linear programming, Lagrangian relaxation, column generation, etc.), and constraint programming (logic programming, hyper-arc-consistency, constraint propagation, etc.). For introductions, we refer the reader to:

- **Algorithm and Complexity Theory**
  - Cormen, Leiserson, and Rivest: *Introduction to Algorithms* [43].
  - Garey and Johnson: *Computers and Intractability* [88].
  - Hochbaum: *Approximation Algorithms for NP-hard Problems* [110].
- **Operations Research.**
  - Nemhauser and Wolsey: *Integer and Combinatorial Optimization* [158].
  - Ahuja, Magnati, and Orlin: *Network Flows* [1].
  - Jünger and Naddef: *Computational Combinatorial Optimization* [126].
- **Constraint Programming.**
  - Marriott and Stuckey: *Programming with Constraints: An Introduction* [145].
  - Kumar: *Algorithms for Constraints-Satisfaction Problems: A Survey* [138].
  - Apt: *The Rough Guide to Constraint Propagation* [6].

1.3.3 Publications

Many parts of the work presented have been published on several workshops and conferences. In case of multiple authors, the results have been achieved in a joint effort of the collaborating researchers. An alphabetical ordering of the list of authors indicates that all researchers contributed equally (this applies to Section 2.5 and Chapters 4, 9), whereas a deviation from the alphabetical ordering denotes that the first mentioned authors contributed significantly more than the other authors (Chapters 3, and 5–8).

Unreviewed Workshops

- U. Junker, S.E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. A Framework for Constraint Programming Based Column Generation. *16th International Joint Conference on Artificial Intelligence (IJCAI)*, Workshop on Non-Binary Constraints, 1999.

- T. Fahle, U. Junker, S.E. Karisch, N. Kohl, M. Sellmann, and B. Vaaben. Constraint Propagation for Complex Column Generation Subproblems. *17th International Symposium on Mathematical Programming (ISMP)*, 2000.

- G. Kliewer, M. Sellmann, and A. Koberstein. Solving the capacitated network design problem in parallel. *3rd meeting of the PAREO Euro working group on Parallel Processing in Operations Research (PAREO)*, 2002.

Reviewed Workshops

- T. Fahle and M. Sellmann. Constraint Programming Based Column Generation with Knapsack Subproblems. *2nd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, Paderborn Center for Parallel Computing, Technical Report tr-001-2000:33–44, 2000.

- M. Sellmann, K. Zervoudakis, P. Stamatopoulos, and T. Fahle. Integrating Direct CP Search and CP-based Column Generation for the Airline Crew Assignment Problem. *2nd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, Paderborn Center for Parallel Computing, Technical Report tr-001-2000:163–170, 2000.

- M. Sellmann and T. Fahle. CP-Based Lagrangian Relaxation for a Multimedia Application. *3rd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, pp. 1–14, 2001.

- M. Sellmann and W. Harvey. Heuristic Constraint Propagation. *4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, pp. 191–204, 2002.

International Conferences

- U. Junker, S.E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. A Framework for Constraint Programming Based Column Generation. *5th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 1713:261–274, 1999.

- M. Sellmann and T. Fahle. Coupling Variable Fixing Algorithms for the Automatic Recording Problem. *9th Annual European Symposium on Algorithms (ESA)*, LNCS 2161:134–145, 2001.

- T. Fahle, S. Schamberger, and M. Sellmann. Symmetry Breaking. *7th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2239:93–107, 2001.

- M. Sellmann, G. Kliewer, and A. Koberstein. Lagrangian Cardinality Cuts and Variable Fixing for Capacitated Network Design. *10th Annual European Symposium on Algorithms (ESA)*, LNCS 2461:845–858, 2002.
- M. Sellmann. An Arc-Consistency Algorithm for the Weighted All Different Constraint. *8th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2470:744–749, 2002.
- M. Sellmann and W. Harvey. Heuristic Constraint Propagation. *8th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2470:738–743, 2002.

Journals

- 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.
- 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*, 115:207–225, 2002.
- T. Fahle and M. Sellmann. Cost-Based Filtering for the Constrained Knapsack Problem. *Annals of Operations Research*, 115:73–93, 2002.
- M. Sellmann and T. Fahle. Constraint Programming Based Lagrangian Relaxation for the Automatic Recording Problem. *Annals of Operations Research*, to appear.

PART I

—Methods —

In the first part of this thesis, we introduce general purpose methods for pruning and filtering with respect to cost considerations and symmetry.

In particular, we develop a tool box of cost-based filtering algorithms in Chapter 2. We introduce the notion of relaxed consistency and study the complexity of achieving different levels of consistency for shortest path constraints, weighted stable set constraints on interval graphs, weighted all-different constraints, and knapsack constraints.

Then, in Chapter 3, we investigate how the interplay of optimization constraints can be improved with the help of the standard problem decomposition techniques column generation and Lagrangian relaxation.

Finally, in Chapter 4, we develop a straightforward symmetry breaking method that is based on the detection of dominance relations between choice points. The method is applied to three different problems and is shown to be particularly suited for highly symmetric problems.

Chapter 2

Optimization Constraints

In this chapter, we develop a “tool box” of domain filtering algorithms that can be used for solving combinatorial optimization problems. While concentrating on some specific problems, it is important to keep in mind that the domain filtering algorithms we develop are to be used as *building blocks* for tackling more complex optimization problems. As a matter of fact, for the problems considered here, domain filtering usually only makes sense in the presence of additional constraints. Different methods of how cost-based filtering algorithms can be exploited in the context of more general optimization problems will be presented in Chapter 3.

2.1 Definitions and General Observations

Within a tree search, during the course of optimization, we compute a sequence of feasible solutions. We refer to the best known feasible solution as the *incumbent solution*. Obviously, once we have found a solution of a certain quality, we are searching for better solutions only. Thus, we impose a restriction on the objective. That restriction, in combination with other side-constraints of the original problem, forms an *optimization constraint* [65, 79, 80, 128, 162, 189], which is the core concept that we will be using throughout this chapter. It was developed by a community that has been working on the integration of constraint programming (CP) and operations research (OR) in recent years. In the OR world, though never explicitly stated as constraints, optimization constraints are frequently used for bound computations and variable fixing. From a CP perspective, they can be viewed as *global constraints* that link the objective with some other constraints of the problem:

Given $n \in \mathbb{N}$, let $X_1, \dots, X_n$ denote some variables with finite domains $D_1 := D(X_1), \dots, D_n := D(X_n)$. Furthermore, given a constraint $\zeta : D_1 \times \cdots \times D_n \to \{0,1\}$, and an objective function $Z : D_1 \times \cdots \times D_n \to \mathbb{Q}$, let $x_i \in D_i \ \forall\, 1 \leq i \leq n$.

Definition 2.1 *Let $B \in \mathbb{Q}$ denote an upper (lower) bound on the objective Z to be minimized (maximized).*

- $\vartheta_{\zeta,Z}[B] : D_1 \times \cdots \times D_n \to \{0,1\}$ *with* $\vartheta_{\zeta,Z}[B](x_1,\ldots,x_n) = 1$ *iff* $\zeta(x_1,\ldots,x_n) = 1$ *and* $Z(x_1,\ldots,x_n) < B$ *is called* minimization constraint.
- $\vartheta_{\zeta,Z}[B] : D_1 \times \cdots \times D_n \to \{0,1\}$ *with* $\vartheta_{\zeta,Z}[B](x_1,\ldots,x_n) = 1$ *iff* $\zeta(x_1,\ldots,x_n) = 1$ *and* $Z(x_1,\ldots,x_n) > B$ *is called* maximization constraint.
- *A minimization or maximization constraint is also called an* optimization constraint.

The purpose of optimization constraints is twofold: first, they can be used for pruning by computing an upper/lower bound on the objective, which is the common idea in branch & bound algorithms. Second, they may also be used to remove those values from variable domains that cannot be part of any improving solution, which may be viewed as a generalization of the variable fixing technique: for binary problems, variable fixing and domain filtering are essentially the same.

2.1.1 On the Complexity of Cost-based Domain Filtering Problems

In order to achieve a state of (hyper-)arc-consistency [6, 138][1] of an optimization constraint, we have to find and remove all assignments that cannot be extended to an improving solution that is feasible with respect to ζ. That is, if ζ is the only constraint of a combinatorial optimization problem (we call that optimization problem and the optimization constraint *corresponding to* or *associated with* each other), an arc-consistency algorithm allows us to compute improving solutions in a backtrack-free search. Consequently, if the original problem is NP-hard, so is the problem of achieving arc-consistency of the corresponding optimization constraint. The Knapsack Problem is an example for such a situation.

If the optimization problem associated with an optimization constraint is polynomial, then the arc-consistency problem may also be polynomial. The Weighted Bipartite Matching Problem is an example, because there exists a polynomial time algorithm for the problem and the removal of an edge or two nodes (when the edge between the nodes is chosen to be part of the matching) does not change the structure of the problem.

The situation may change, however, if the problem structure is not preserved when a variable is forced to take a specific value. Consider a Shortest Path Problem in an arbitrary network, where we use a binary variable for each edge. The problem of finding a shortest path is of course solvable in polynomial time. However, if we are to compute the set of edges that must or

[1] To be precise: here and in the remainder of this thesis, we consider the problem of achieving a state of hyper-arc-consistency. However, for historical reasons and also to improve the readability, we simply write arc-consistency when we refer to a state of local consistency with respect to one constraint only.

cannot be part of any simple path that does not exceed a certain length, we are facing an NP-hard problem: it is easy to see that the Two Vertex Disjoint Paths Problem [83] can be reduced to this problem.

2.1.2 Degrees of Consistency

The discussion shows that we cannot always hope for a cost-based domain filtering algorithm that achieves arc-consistency. Therefore, we may consider to develop less effective but polynomial time bounded filtering algorithms that may only achieve a weaker degree of consistency. Note that, in a different context, the idea of weaker forms of consistency gives yield to the notion of *bound consistency* that can also be achieved more easily than general arc-consistency, and that has proven valuable for many applications.

Regarding cost-based filtering, an idea that has been developed in OR to perform variable fixing on integer linear problems is the *reduced-cost filtering* method: when solving the continuous relaxation bound on a linear combinatorial optimization problem with the help of a general LP solver (such as the simplex algorithm or interior point methods), we get dual information and reduced-cost data for free. That data can be used to compute a lower bound on the loss of performance that we have to accept when adding a new constraint of the form $X = x$ (usually this is done by performing one dual simplex re-optimization step). Of course, if the loss is too large, we can deduce that x must be removed from the domain of X.

We strengthen and generalize the basic idea by coupling optimization constraints and relaxations:

Definition 2.2 *Given an optimization constraint $\vartheta_{\zeta,Z}[B] : D_1 \times \cdots \times D_n \to \{0,1\}$, let $\Delta := D_1 \times \cdots \times D_n$. Furthermore, denote the set of all subsets of Δ by 2^Δ.*

- *Let $\vartheta_{\zeta,Z}[B]$ be a minimization constraint, and let $L : 2^\Delta \to \mathbb{Q}$ such that for all $M_i \subseteq D_i$, $1 \le i \le n$,*

$$L(M_1 \times \cdots \times M_n) \le \min\{Z(x_1,\dots,x_n) \mid \zeta(x_1,\dots,x_n) = 1,\ x_i \in M_i,\ 1 \le i \le n\},$$

  where $\min\emptyset = \infty$. *We call L a* relaxation of $\vartheta_{\zeta,Z}$ *and say that $\vartheta_{\zeta,Z}[B]$ is* relaxed L-consistent, *iff for any given $1 \le i \le n$ and $x_i \in D_i$, $L(D_1 \times \cdots \times \{x_i\} \times \cdots \times D_n) < B$.*

- *Analogously, let $\vartheta_{\zeta,Z}[B]$ be a maximization constraint, and let $U : 2^\Delta \to \mathbb{Q}$ such that for all $M_i \subseteq D_i$, $1 \le i \le n$,*

$$U(M_1 \times \cdots \times M_n) \ge \max\{Z(x_1,\dots,x_n) \mid \zeta(x_1,\dots,x_n) = 1,\ x_i \in M_i,\ 1 \le i \le n\},$$

  where $\max\emptyset = -\infty$. *We call U a* relaxation of $\vartheta_{\zeta,Z}$ *and say that $\vartheta_{\zeta,Z}[B]$ is* relaxed U-consistent, *iff for any given $1 \le i \le n$ and $x_i \in D_i$, $U(D_1 \times \cdots \times \{x_i\} \times \cdots \times D_n) > B$.*

As one would expect, the definition states that relaxed L-consistency (relaxed U-consistency follows analogously) can be achieved the easier the weaker the relaxation L is. For $L \equiv -\infty$, there is no work to do to achieve relaxed L-consistency, whereas arc-consistency is enforced when $L(M_1 \times \cdots \times M_n) = \min\{Z(x_1, \ldots, x_n) \mid \zeta(x_1, \ldots, x_n) = 1,\ x_i \in M_i,\ 1 \leq i \leq n\}$. That is, the choice of L determines the degree of domain filtering.

In practice, L is usually chosen as a fairly tight bound that can still be computed quickly. Generally, within a tree search there is a trade-off between the time spent per search node and the total number of search nodes. Thus, the favorable choice of the accuracy of the relaxation is always subject to the optimization problem at hand. Note that the definition of relaxed consistency allows to compare domain filtering algorithms with respect to the running time and the degree of consistency they achieve.

In the following, we develop cost-based domain filtering algorithms for shortest path constraints, weighted stable set constraints on interval graphs, weighted all-different constraints, and knapsack constraints.

We have seen already, that achieving arc-consistency for knapsack or shortest path constraints are NP-hard tasks. Therefore, the challenge is to develop efficient filtering algorithms for the two constraints that achieve relaxed consistency for favorably strong relaxations.

On the other hand, for weighted stable set constraints on interval graphs and weighted all-different constraints, arc-consistency can be achieved in polynomial time. Thus, we aim at developing filtering methods for these constraints that achieve arc-consistency and run faster than in time $O(nT)$, where n denotes the number of (binary) variables, and T is the time needed to solve the corresponding optimization problem. That time bound can obviously be achieved by probing all variable values in a brute force manner. As we will see, we can achieve arc-consistency for both constraints in time $O(T)$, i.e. the same time that is needed to compute the bound on the objective.

2.2 Shortest Path Constraints

Many real-world problems, e.g. in personnel scheduling and transportation planning, can be modeled naturally as Constrained Shortest Path Problems (CSPs), i.e., as Shortest Path Problems with additional constraints. A well-studied problem in this class is the Resource Constrained Shortest Path Problem. Reduction techniques are vital ingredients of solvers for the CSP, that is frequently NP-hard, depending on the nature of the additional constraints. Viewed as heuristics, until today these techniques have not been studied theoretically with respect to their efficiency, i.e., with respect to the relation of filtering power and running time. Using core concepts of constraint programming and the notion of relaxed consistency, we provide a sound theoretical study of cost-based filtering for shortest path constraints on acyclic, on undirected and on directed graphs that must not contain negative cycles.

Real-world problems can frequently be modeled as Shortest Path Problems with additional constraints. The best known Constrained Shortest Path Problem (CSP) is probably the *Resource Constrained Shortest Path Problem* [4, 17, 57, 103, 125] that consists in the combination of a Shortest Path Problem and capacity constraints on a set of resources. Even on DAGs, for non-negative objective functions and for only one resource that problem is known to be NP-hard [88].

Standard applications for the Resource Constrained Shortest Path Problem are route planning in traffic networks and quality of service routing [161, 216]. The Crew Scheduling Problem is another example of a real-world problem where CSPs are used in many successful approaches: In a column generation process, CSPs have to be solved to generate columns, which correspond to individual lines of work in this context [219]. In Chapter 5, we present an example of such a crew scheduling approach based on column generation with shortest path sub-problems.

Generally, CSPs appear very often as sub-problems in column generation approaches. Examples range from route guidance [123] and duty scheduling in public transit [25] up to the scheduling of switching engines [142]. In Section 3.1, a general framework for constraint programming based column generation is developed that formalizes the use of optimization constraints in this context.

To solve Constrained Shortest Path Problems, state of the art solvers compute lower and upper bounds on the problem and then close the duality gap. The latter task is carried out by an enumeration procedure such as a tree search [17], dynamic programming [154] or a k-shortest path algorithm [103]. Particularly in a tree search, but also in the other approaches the tightening of (sub-)problems is vital for an effective gap closing procedure. Therefore, it is essential for the overall performance and the practical success of the entire approach.

The first tightening strategy that was proposed goes back to a work done by Aneja, Aggarwal, and Nair [4] for problem reduction of the Resource Constrained Shortest Path Problem. The basic idea consists in identifying nodes and arcs that cannot be visited by any path that obeys the given resource restrictions. The same method can also be used to identify nodes and arcs that cannot be visited by any improving path, which gives a first cost-based filtering algorithm for the problem. Dumitrescu and Boland [57] proposed a repeated problem reduction procedure that has shown to be very successful for hard constrained problems. Beasley and Christofides [17] have shown how a tighter global, Lagrangian relaxation based bound can be used for the elimination of nodes and arcs.

Apparently, none of these heuristics has been classified with respect to its filtering abilities. Moreover, the reduction techniques used all focus on the removal of nodes and arcs, but those arcs and nodes that must be visited by all path of a certain quality remain undetected. However, with respect to the additional constraints of the CSP this information can be very valuable as it may prove useful for an additional reduction of the problem.

Constraint programming theory provides means for the level of consistency that a constraint filtering algorithm achieves. Using the extended notion of relaxed consistency for optimization constraints, we are able to measure a non-exact filtering algorithm not only with respect to its running time, but also to its filtering power that is determined by the quality of the relaxation used. With respect to shortest path constraints, we study the complexity of achieving a state of (hyper-)arc-consistency. Since the problem is NP-hard in the general case, we introduce shortest path relaxations and develop and compare different filtering algorithms for different graph classes.

In Section 2.2.2, we develop an efficient linear time filtering algorithm for shortest path constraints on DAGs. In Section 2.2.3, we investigate the problem on undirected graphs, where it is shown to be NP-hard. We introduce a shortest path relaxation L_1 and formulate a linear time algorithm that achieves a state of relaxed L_1-consistency. Finally, in Section 2.2.4, we develop cost-based filtering algorithms for shortest path constraints on general directed networks with non-negative costs or graphs that at least do not contain negative weight cycles.

2.2.1 Definition

Definition 2.3 *Denote a weighted (directed or undirected) graph by $G = (V, E, c)$, and let $h \in \mathbb{N}$.*

- *A sequence of nodes $P = (i_1, \ldots, i_h) \in V^h$ with $(i_f, i_{f+1}) \in E$ for all $1 \leq f < h$ is called a* path from i_1 to i_h in G.
- *A path P is called* simple *iff P visits every node at most once. For all $i, j \in V$, denote the set of all simple paths from i to j by $\pi(i, j)$.*
- *For all paths P, nodes $i \in V$ and edges $(i, j) \in E$, we write $i \in P$ or $(i, j) \in P$ iff P visits node i or the edge (i, j), respectively. For a set of nodes or edges S, we write $S \subseteq P$, iff $s \in P$ for all $s \in S$. Correspondingly, we write $P \subseteq S$ iff $s \in S$ for all $s \in P$.*

- *The cost of a path $P = (i_1, \dots, i_h)$ is defined as $cost(P) := \sum_{1 \le j < h} c_{i_j i_{j+1}}$. Accordingly, for any set $S \subseteq E$, we define $cost(S) := \sum_{(i,j) \in S} c_{ij}$.*

Definition 2.4 *Let $G = (V, E, c)$ denote a (directed or undirected) graph with $n = |V|$ and $m = |E|$, a designated source $v_1 \in V$ and sink $v_n \in V$, and arc costs $c_{ij} \in \mathbb{Z}$. Furthermore, assume we are given binary variables $X_1, \dots, X_m$, an integer variable Z, and an objective bound $B \in \mathbb{Z}$.*

- *A* shortest path constraint *has binary variables $X_1, \dots, X_m$ and an integer variable Z, and an instantiation $X_i = x_i$, for all $1 \le i \le m$, and $Z = z$ is consistent iff the following holds:*

  1. *The set $\{e_i \mid X_i = 1\} \subseteq E$ determines a simple path in the graph G from the source v_1 to the sink v_n.*

  2. *z is the cost of the path represented by the value of X, and $z < B$.*

- *Every simple path from source to sink with costs less than B is called* admissible.

To ease the notation, for the remainder of this section we assume that a shortest path constraint is associated with a set variable $Y \subseteq E$ that represents the set of edges e_i for which $X_i = 1$. The (current) domains of the variables X will be represented by two sets: the set of *possible members* $pos(Y)$, and the set of *required members* $req(Y)$ of Y. In the sub-tree of the search rooted at the current choice point we require $req(Y) \subseteq Y \subseteq pos(Y)$. That is, $req(Y)$ represents the set of variables for which it has been set $X_i = 1$, and the set $pos(Y)$ represents the set of all variables for which it has not been decided to set $X_i = 0$ already. Then, in the current choice point, we have to search for admissible paths P such that $req(Y) \subseteq P \subseteq pos(Y)$. Note that we use the set variable Y only to ease the presentation. It has no impact on the implementation that is assumed to use only the variables X. Especially, the didactic use of a set variable has no impact on the definition of arc-consistency. To achieve arc-consistency of a shortest path constraint, we must ensure:

- For all $e \in pos(Y)$, there exists an admissible path P with $req(Y) \cup \{e\} \subseteq P \subseteq pos(Y)$, and

- for all $e \in pos(Y) \setminus req(Y)$, there exists an admissible path with $req(Y) \subseteq P \subseteq pos(Y) \setminus \{e\}$.

Obviously, on the existence of an admissible path can be decided by applying a shortest path algorithm. However, to decide whether there exists a simple path that visits a set of edges is an NP-hard task which can be shown by a simple reduction to the Two Vertex Disjoint Path Problem [83]. Consequently, the arc-consistency problem for the general shortest path constraint is also NP-hard.

2.2.2 Shortest Path Problems on DAGs

The reduction on the Two Vertex Disjoint Path Problem does not prove NP-hardness for acyclic graphs. As a matter of fact, on DAGs the problem is solvable in polynomial time. In the following, we develop a cost-based domain filtering algorithm for shortest path constraints that achieves arc-consistency in time $O(n+m)$.

As stated above, to perform domain reduction for a shortest path constraint, we are facing two tasks: first, in order to shrink the set $pos(Y)$ as much as possible, we need to identify all arcs in E that cannot be visited by any admissible path $req(Y) \subseteq P \subseteq pos(Y)$. Second, to increase $req(Y)$ as much as possible, we must compute all arcs that must be visited by all admissible path $req(Y) \subseteq P \subseteq pos(Y)$.

First, to ensure that all paths computed are subsets of $pos(Y)$, we remove all arcs from G that are not in $pos(Y)$. Then we want to make sure that all admissible paths are super-sets of $req(Y)$. To do so, we set $M := m \max\{c_{ij} \mid c_{ij} \geq 0\}$, we decrease the arc weight c_{ij} of edge $(i, j) \in req(Y)$ by M and adapt the upper bound B by subtracting $M|req(Y)|$. In the following, we assume that $G = (V, E, c)$ has been updated accordingly. Note that the removal of nodes and arcs can be performed in time $O(n+m)$ when an adjacency list representation (such as the forward or backward star representations [1]) of G is used.

2.2.2.1 Removing Arcs from the Possible Set

Without loss of generality, we may assume that the nodes in V are ordered topologically. If a node $i \in V$ precedes a node $j \in V$ in the topological ordering, we write $i \leq_{top} j$. We write $i <_{top} j$ iff $i \leq_{top} j$ and $i \neq j$. Note that, for all arcs $(i, j) \in E$, it holds that $i <_{top} j$. Furthermore, we may assume that v_1 and v_n are the first and last nodes in this ordering, respectively. To find out for a given arc $(i, j) \in E$ whether there still exists an admissible path P with $(i, j) \in P$, we use a method that was originally developed for the Resource Constrained Shortest Path Problem [4]:

First, we compute the shortest path distances $c(v_1, i)$ from the source v_1 to all nodes $i \in V$. Once a topological ordering of the nodes is known (which takes time $O(n+m)$), this can be done in time $O(n+m)$ even in the presence of negative arc weights (see [43]). Next, we compute the shortest path distances $c(i, v_n)$ from all nodes $i \in V$ to the sink v_n, which can also be done in linear time by reversing all arcs and using the same procedure as before with v_n as the starting node. Finally, for every arc $(i, j) \in pos(Y)$ we check whether the shortest simple path from v_1 to v_n via (i, j) has costs lower than B, i.e., we remove (i, j) from $pos(Y)$ iff

$$c(v_1, i) + c_{ij} + c(j, v_n) \geq B. \tag{2.1}$$

Using the same idea we can also identify nodes that can be removed from the graph, because they can never be part of any path from source to sink that is short enough to beat the current upper bound. Figure 2.1 illustrates the situation.

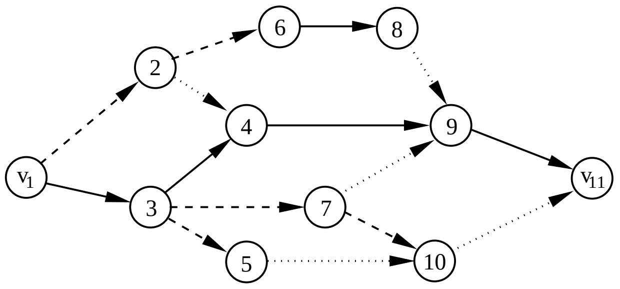


Fig. 2.1: The figure shows arcs on shortest paths from v_1 and to v_{11} in a DAG. Dashed lines mark shortest-path arcs from v_1, dotted lines those to v_{11}. Solid lines represent arcs that are in both sets. Consider for example node 7: the shortest path from v_1 to 7 is $(v_1, 3, 7)$, and the shortest path from node 7 to v_{11} is $(7, 9, v_{11})$. Therefore, a shortest path from v_1 to v_{11} via node 7 is $(v_1, 3, 7, 9, v_{11})$.

2.2.2.2 Adding Arcs to the Required Set

After having shrunk the possible set, we still need to identify all arcs that must be visited by all admissible paths to achieve a state of arc-consistency for a shortest path constraint. If we perform the algorithm for the removal of members from the possible set first, we may assume that the graph G only contains arcs and nodes that are part of at least one admissible path. Then the following lemma characterizes the arcs that are required.

Lemma 2.1 *Denote the graph that is obtained by un-directing all arcs in G by G_u. If, for all arcs $e \in E$, there exists an admissible path P in G such that $e \in P$, then the following statements are equivalent:*

1. *Every admissible path P in G contains the arc $(i, j) \in E$.*
2. *For all arcs $(k,l) \in E$ with $(k,l) \neq (i, j)$, it holds that $k,l \leq_{top} i$ or $j \leq_{top} k,l$.*
3. *The edge $\{i, j\}$ is a bridge in G_u.*

Proof:

- **1. $\Rightarrow$ 2.** Assume that there exist nodes $k,l \in V$ such that $(k,l) \in E$, $(k,l) \neq (i, j)$. Then there exists an admissible path P such that $(k,l) \in P$ and $(i, j) \in P$. Thus, $k,l \leq_{top} i$ or $j \leq_{top} k,l$.
- **2. $\Rightarrow$ 3.** Statement (2) implies that the arc (i, j) is the only arc that leaves node i. Also, there exists no arc that by-passes the node i in G. Thus the removal of $\{i, j\}$ disconnects v_1 and v_n in G_u, i.e., $\{i, j\}$ is a bridge in G_u.
- **3. $\Rightarrow$ 1.** If $\{i, j\}$ is a bridge in G_u, then every path from v_1 to v_n in G must contain the arc (i, j). Therefore, for every admissible path P, it also holds that $(i, j) \in P$. ∎

Lemma 2.1 allows us to compute the arcs to be required by searching for bridges in the undirected version of G, and the bridges of an undirected graph can be computed in time $O(n+m)$ [43].

The following theorem summarizes the results in Sections 2.2.2.1 and 2.2.2.2.

Theorem 2.1 *On DAGs, arc-consistency for the shortest path constraint can be achieved in linear time.*

2.2.2.3 Incremental Shortest Path Constraint Propagation

During the search, the domain of Y changes frequently, which means that many and often similar SSSPs have to be solved. Therefore, we develop an incremental version of the algorithm. Instead of restarting the computation from scratch, it makes use of previously computed shortest path information. Moreover, it uses the information on the differences in the domains of the current and the last call: the required set may have grown, and the possible set may have shrunk.

For an efficient implementation of the algorithm, we use both the forward and the backward star representation of G. We choose this redundant data structure, first because we need to compute shortest paths in G and the reverse of G, and second because we are able to perform the incremental shortest path update more efficiently.

In order to compute the arcs and nodes that have to be removed from the graph, a support idea in the style of AC-6 [21] can be used to reduce the computational effort required. If a node i leaves the possible set, we mark all its adjacent nodes j as *affected* by the removal or distance change of node i. If the node i was even the direct shortest path predecessor of node j in the preceding call to the propagation routine, we refer to i as the *support node* of j.

We iterate over all nodes in their topological order. If the node j is affected, we check whether its support still exists or is replaceable by another node without a change in the shortest path distance $c(v_1, j)$. Since this requires iterating over all in-going arcs, a backward star representation is used. Only if the support is lost and cannot be replaced, we need to propagate the distance update and mark the successors of node j as affected. To do this efficiently, the forward star representation of G is used. That way, we perform a continuing update on all affected nodes in only one pass.

If a new arc (i, j) becomes required, we do not need to re-compute the shortest path distances of all nodes in the graph, because nodes that precede i in the topological ordering are not affected by that change. Therefore, it is sufficient to restart the SSSP-algorithm for DAGs at node i. The distance $c(v_1, i)$ can be reused from the previous call to the constraint propagation algorithm. Moreover, we can stop examining all outgoing arcs when we run over the first node k for which (k, l) was already formerly required: the shortest path tree structure “behind” a required arc remains intact, and the difference in the distance $c(v_1, k)$ before and after the change simply applies to all following nodes as well.

In the worst case, the incremental variant of the propagation algorithm may still require time $O(n+m)$. However, in practice the ideas sketched in the above can reduce the computational effort considerably as we shall see in Chapter 5.

2.2.3 Shortest Path Problems on Undirected Graphs

Next we consider shortest path constraints on undirected graphs with non-negative edge weights. Unlike in the previous section, achieving arc-consistency for a shortest path constraint on undirected graphs is an NP-hard task, as the following observation shows.

Lemma 2.2 *Given an undirected graph $G = (V, E)$, $n := |V|$, $m := |E|$, two designated nodes $v_1, v_n \in V$ and a set of edges $S \subseteq E$, it is NP-hard to decide whether there exists a simple path $P \in \pi(v_1, v_n)$ with $S \subseteq P$.*

Proof: We reduce the problem to the Hamiltonian Path Problem: Given an undirected graph $G = (V, E)$, do there exist two nodes $s, t \in V$ and a simple path $P \in \pi(s, t)$ with $V \subseteq P$? We transform G into an instance (G', v_1, v_n, S) such that there exists a simple path $P' \in \pi(v_1, v_n)$ with $S \subseteq P'$ iff there exists a Hamiltonian path in the original graph G.

First, we add two new nodes v_1, v_n, and all edges in $V \times \{v_1, v_n\}$. Then every node $v \in V$ is replaced by the structure given in Figure 2.2: The ellipse sketches a former node $v \in V$. For all edges $e_1, \ldots, e_d \in V \cup \{v_1, v_n\} \times V \cup \{v_1, v_n\}$ incident to v, we add new nodes $1_v, \ldots, d_v$ that connect the new structure with their corresponding edges. Moreover, we add two new nodes a_v and b_v and edges $\{1_v, \ldots, d_v\} \times \{a_v, b_v\}$. Finally, we set $S := \{\{a_v, b_v\} \mid v \in V\}$, the set of edges that must be visited.

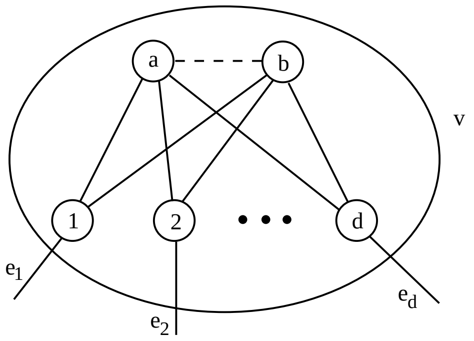


Fig. 2.2: The structure replacing a node in G.

Then there exists a simple path $P' \in \pi(v_1, v_n)$ with $S \subseteq P'$ iff there exists a Hamiltonian path in the original graph G: Given a path $P' \in \pi(v_1, v_n)$ with $S \subseteq P'$. P' must visit all structures sketched in Figure 2.2 at least once, because it must visit all edges $\{a_v, b_v\}$. On the other hand, after P' has visited the edge $\{a_v, b_v\}$ it can never return to the current structure because all paths that pass through it must visit either node a_v or b_v again. Therefore, P' visits all structures corresponding to the original nodes in V exactly once, and thus defines a Hamiltonian path in G.

On the other hand, assume there exists a Hamiltonian path $P \in \pi(s,t)$ in G for some nodes $s,t \in V$. Then we construct a path $P' \in \pi(v_1,v_n)$ with $S \subseteq P'$ in the following manner: We start at v_1 and go to node s first. Now, for each $v \in V$ that the Hamiltonian path visits we enter via some edge e_i, go to node a_v from there, we visit the edge $\{a_v,b_v\}$, and find our way out via the node incident to e_j that is visited by P next. Since $V \subseteq P$, we visit all edges in S like that. Finally, we end at node t and proceed to v_n from there. ■

As a simple consequence of Lemma 2.2, we get the following

Corollary 2.1 *To check the arc-consistency of a shortest path constraint on an undirected graph is NP-hard.*

Due to this result, in the following, we develop a cost-based filtering algorithm that achieves relaxed consistency rather than arc-consistency. In order to introduce the relaxation we want to use, we first start with the following

Defi nition 2.5 *Denote a weighted (directed or undirected) graph by $G = (V,E,c)$.*

- *A path P is called a* k-simple path in *G iff, for all $j \in V$, the path P visits j at most k times. Note that a 1-simple path is a simple path in G.*
- *With $P(i,j) \in \pi(i,j)$ we refer to a shortest path from i to j (with respect to c). Then, to ease the notation, we set $c(i,j) := cost(P(i,j))$.*
- *Given a shortest path constraint, a k-simple path P from v_1 to v_n is called a k-*admissible path *iff $cost(P) < B$.*

Note that in a graph with non-negative edge weights, a shortest admissible path is also a shortest 2-admissible path. Now, instead of checking for admissible paths only, we consider the following shortest path relaxation: Let $D(Y)$ denote the domain of Y represented as the pair of sets $(req(Y), pos(Y))$. We set $H := \{P \mid P \in \pi(v_1,v_n) \text{ with } P \subseteq pos(Y)\}$ and $F_f := \{P \mid P \text{ is a 2-simple path from } v_1 \text{ to } v_n \text{ with } f \in P\}$ for all $f \in E$. Then we define

$$L_1(D(Y)) := \max\{\min\{cost(P) \mid P \in H\}, \max_{f \in req(Y)}\{\min\{cost(P) \mid P \in F_f\}\}.$$

Lemma 2.3 *L_1 is a shortest path relaxation, i.e., it holds that*

$$L_1(D(Y)) \leq \min\{cost(P) \mid P \in \pi(v_1,v_n), req(Y) \subseteq P \subseteq pos(Y)\}.$$

Proof: Let $P \in \pi(v_1,v_n)$ denote the shortest path in G with $req(Y) \subseteq P \subseteq pos(Y)$. Obviously, it holds that $P \in H$ and $P \in F_f$ for all $f \in req(Y)$. Therefore, $L_1(D(Y)) \leq cost(P)$. ■

The big advantage of the relaxation above is that it allows to be checked for consistency very easily, as we shall see below. Note however, that L_1 does not require that the 2-admissible paths must visit all nodes in $req(Y)$ simultaneously. Of course, this weakens the relaxation. We can reduce the negative effects by improving the probability that a 2-admissible path visits the edges in $req(Y)$: we set $c_{ij} := 0$ for all $\{i,j\} \in req(Y)$ and subtract $cost(req(Y))$ from B.

According to the definition, a shortest path constraint is relaxed L_1-consistent, iff

1. for all $f \in pos(Y)$, there exists a 2-admissible path $P \in F_f$, and

2. for all $f \in pos(Y) \setminus req(Y)$, there exists an admissible path $P \in H$ with $f \notin P$.

In the following two sections, we show how relaxed L_1-consistency can be achieved efficiently.

2.2.3.1 Removing Edges from the Possible Set

In order to check whether there exists a 2-admissible path in G that visits an edge $\{i,j\} \in E$, we can use the same idea as in the previous section on shortest paths in DAGs. Obviously, the shortest 2-simple path from v_1 to v_n that visits $\{i,j\}$ is either $(P(v_1,i),P(j,v_n))$ with costs $c(v_1,i)+c_{ij}+c(j,v_n)$ or $(P(v_1,j),P(i,v_n))$ with costs $c(v_1,j)+c_{ij}+c(i,v_n)$. Therefore, to check whether an edge has to be removed from $pos(Y)$ with respect to the relaxation L_1, it is sufficient to know the shortest path distances from the source and to the sink of all nodes. Both values can be computed for all nodes by only two shortest path computations in G in time $O(m+n\log n)$ by using Dijkstra's algorithm in combination with Fibonacci heaps [86]. In a RAM model, shortest paths on undirected graphs can be computed in time $O(m+n)$ when using the algorithm of Thorup (see [207] and the recent extension of Pettie and Ramachandran in [166]). Thus, the set of edges that has to be removed from $pos(Y)$ to achieve relaxed L_1-consistency can be computed in time $O(m+n\log n)$, and in time $O(m+n)$ on a RAM.

2.2.3.2 Adding Edges to the Required Set

After having removed all edges from G that cannot be part of any 2-admissible path, the edges that must be visited by all such paths can be characterized by the following

Theorem 2.2 *Assume that all edges in G are part of at least one 2-admissible path. Then an edge $\{r,s\} \in E$ must be visited by all admissible paths, iff*

- $\{r,s\} \in P(v_1,v_n)$*, and*
- $\{r,s\}$ *is a bridge in G.*

Before we can prove the above theorem, we need to show the following two lemmas first:

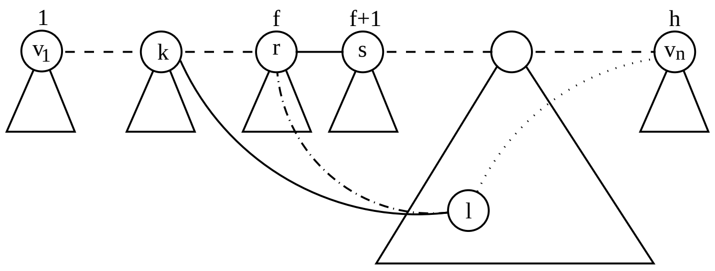


Fig. 2.3: The fi gure schematically shows an edge $\{k,l\} \in E$ that must exist according to Lemma 2.4. Solid lines mark edges in E, and dashed lines mark parts of the shortest path between v_1 and v_n. The dotted line between l and v_n indicates that there exists a path between the two nodes that does not visit the edge $\{r,s\}$. The alternating lines and dots between l and r indicate that the shortest path from l to v_n visits node r. The numbers on top of the nodes give their corresponding DFS numbers, and triangles mark DFS sub-trees.

Lemma 2.4 *(Compare with Figure 2.3.) Assume that all edges in G are part of at least one 2-admissible path. Let $\{r,s\} \in E$ denote an edge that must be visited by all admissible paths and that can be removed from G without disconnecting v_1 and v_n. Then there exists an edge $\{k,l\} \in E$ such that*

1. $\exists P \in \pi(v_1,v_n) : \{k,l\} \in P$ *and* $\{r,s\} \notin P$,
2. *k is a shortest path predecessor of r, and*
3. $\{r,s\} \in P(l,v_n)$.

Proof: Assume we compute a shortest path $P = (i_1,\ldots,i_h) \in \pi(v_1,v_n)$. Then $i_1 = v_1$, $i_h = v_n$ and $i_f = r$, $i_{f+1} = s$ for some $1 \leq f < h$. Next, we change the graph representation of G such that $\{i_g,i_{g+1}\}$ is the first outgoing edge of node i_g for all $1 \leq g < h$. For all nodes $j \in V$, let $d_j \in \{1,\ldots,n\}$ denote the ordering in which the nodes are first visited by a depth first search using the modified graph representation of G. Then $d_{i_g} = g$ for all $1 \leq g \leq h$. Since the removal of $\{r,s\}$ does not disconnect v_1 and v_n, there exists a forward edge $\{k,l\} \in E$ with $d_k < f$ and $d_l > f+1$. This implies Statements 1 and 2.

It remains to show that $\{r,s\} \in P(l,v_n)$. By assumption, there exists a 2-admissible path R through edge $\{k,l\}$. There are two possibilities: either R visits node k or node l first, which corresponds to:

a) $c(v_1,k) + c_{kl} + c(l,v_n) < B$, or

b) R visits l before k and $c(v_1,l) + c_{kl} + c(k,v_n) < B$.

In the first case, since $\{r,s\} \notin P(v_1,k)$ and $\{r,s\}$ must be visited by all admissible paths, it holds that $\{r,s\} \in P(l,v_n)$, and we are done.

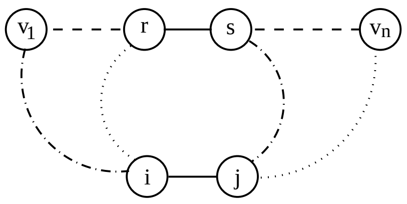


Fig. 2.4: The figure schematically shows an edge $\{i, j\} \in E$ that must exist according to Lemma 2.5. Solid lines mark edges in E, and dashed lines mark parts of the shortest path between v_1 and v_n. Alternating lines and dots indicate parts of the shortest path from v_1 to a node, and dotted lines indicate parts of the shortest path from a node to v_n. The proof of Theorem 2.2 shows that the path $(P(v_1,r),P(r,i),P(j,s),P(s,v_n))$ is 2-admissible and does not visit the edge $\{r,s\}$.

Let us consider the second case. Let $Q \in \pi(v_1,l)$ denote a shortest path from v_1 to l with $\{r,s\} \notin Q$. Without loss of generality, we may assume that k and l are chosen such that $\{k,l\} \in Q$. We observe that $\{r,s\} \in P(v_1,l)$, because otherwise this implies that $\{k,l\} \in Q = P(v_1,l)$ and the 2-admissible path visits node k before node l. Now, since k is a shortest path predecessor of r and $\{r,s\} \in P(v_1,l)$, it holds that $k \in P(v_1,l)$. Then,

$$\begin{aligned} c(v_1,k)+c_{kl}+c(l,v_n) &\leq c(v_1,k)+c_{kl}+c(l,k)+c(k,v_n) \\ &= c(v_1,k)+c(k,l)+c_{kl}+c(k,v_n) \\ &= c(v_1,l)+c_{kl}+c(k,v_n) \\ &< B, \end{aligned}$$

which reduces this case to (a). ∎

Lemma 2.5 *(Compare with Figure 2.4.) Assume that all edges in G are part of at least one 2-admissible path. Let $\{r,s\} \in E$ denote an edge that must be visited by all admissible paths and that can be removed from G without disconnecting v_1 and v_n. Then there exists an edge $\{i, j\} \in E$ such that*

1. *$\{r,s\} \in P(i,v_n)$ and $\{r,s\} \notin P(j,v_n)$, and*
2. *$\{r,s\} \notin P(v_1,i)$ and $\{r,s\} \in P(v_1, j)$.*

Proof: Denote an edge as in Lemma 2.4 by $\{k,l\} \in E$. Then there exists a path $P \in \pi(l,v_n)$ with $\{r,s\} \notin P$, and we may assume $\{r,s\} \in P(l,v_n)$.

1. Since $\{r,s\} \notin P(v_n,v_n)$, there exists an edge $\{i, j\} \in P$ such that $\{r,s\} \in P(i,v_n)$ and $\{r,s\} \notin P(j,v_n)$.
2. By assumption, there exists a 2-admissible path that visits j. Since $\{r,s\} \notin P(j,v_n)$, it follows that $\{r,s\} \in P(v_1, j)$, because $\{r,s\}$ must be visited by all admissible paths. Finally, assume that $\{r,s\} \in P(v_1,i)$. Then the shortest path visiting node i has costs

$$c(v_1,r)+c_{rs}+c(s,i)+c(i,r)+c_{rs}+c(s,v_n).$$

However, the path from v_1 via r, i and s to v_n has costs

$$c(v_1,r)+c(r,i)+c(i,s)+c(s,v_n),$$

which is lower or equal to the cost of the shortest path visiting i. This implies that it is also a shortest path visiting node i. It does not, however, visit some edges with zero costs. Particularly, it does not visit the edge $\{r,s\}$. Therefore, we may assume that $\{r,s\} \notin P(v_1,i)$.

■

Now, we have everything at hand to give the previously postponed

Proof of Theorem 2.2:

$\Leftarrow$ Let $\{r,s\}$ be a bridge on the shortest path $P \in \pi(v_1,v_n)$. Then the removal of $\{r,s\}$ disconnects the graph G. Since the node pairs (v_1,r) and (s,v_n) are still connected, the removal of $\{r,s\}$ also disconnects v_1 and v_n. Thus, for all $P \in \pi(v_1,v_n)$, it holds that $\{r,s\} \in P$. Therefore, also all admissible paths must visit $\{r,s\}$.

$\Rightarrow$ Obviously, if there exists any admissible path, then $P(v_1,v_n)$ is admissible, too. Thus, $\{r,s\} \in P(v_1,v_n)$. Now assume that the removal of $\{r,s\}$ does not disconnect v_1 and v_n. Then, according to Lemma 2.5, there exists an edge $\{i,j\} \in E$ such that $\{r,s\} \in P(i,v_n)$, $\{r,s\} \notin P(j,v_n)$, $\{r,s\} \notin P(v_1,i)$ and $\{r,s\} \in P(v_1,j)$. By assumption, there exists a 2-admissible path R visiting $\{i,j\}$. Without loss of generality, we may assume that R visits node i before node j, because

$$\begin{aligned} c(v_1,j)+c_{ij}+c(i,v_n) &= c(v_1,r)+c_{rs}+c(s,j)+c_{ij}+c(i,r)+c_{rs}+c(s,v_n) \\ &\geq c(v_1,r)+c(r,i)+c_{ij}+c(j,s)+c(s,v_n) \\ &\geq c(v_1,i)+c_{ij}+c(j,v_n). \end{aligned}$$

However, this implies that $\{r,s\} \notin R$, which is a contradiction to the assumption that every admissible path must visit $\{r,s\}$.

■

Using Theorem 2.2, after having removed all edges that cannot be part of any 2-admissible path, we can compute all edges that must be visited by all admissible paths in time $O(m+n)$: first, we compute a shortest path $P \in \pi(v_1,v_n)$ and mark all edges on this path. Then we compute all bridges in G and check which ones are visited by P.

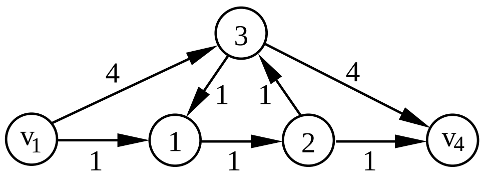


Fig. 2.5: A directed graph with non-negative arc weights. Assume we are given an upper bound $B = 8$. All arcs in the graph are part of an admissible path with costs lower than B, and every admissible path with costs lower than B must visit the arc $(1,2)$. However, there exists a path $(v_1, 3, v_4)$ that does not visit this arc.

The following theorem summarizes the results in the previous two sub-sections:

Theorem 2.3 *On undirected graphs with non-negative edge weights, relaxed L_1-consistency of a shortest path constraint can be achieved in time $O(m + n\log n)$, and in time $O(m + n)$ on a RAM.*

2.2.4 Shortest Path Problems on Directed Graphs

To complete our discussion on cost-based filtering for shortest path constraints, we finish with some results on shortest paths in general directed networks. We start by considering directed graphs with non-negative arc weights. In the end of this section, we will show how these results can be exploited to cope with negative arc weights as well.

As has been stated in the introduction to this section, achieving arc-consistency for shortest path constraints in general networks is NP-hard. Regarding the removal of arcs from the possible set, relaxed L_1-consistency on directed graphs with non-negative arc weights can be achieved in the same way as on undirected graphs. However, with respect to arcs that must be visited by all admissible paths, the situation is more complicated. Recall the result from Section 2.2.3: After having removed the infeasible edges, in undirected graphs, the edges that are required are exactly the ones on the shortest path that must be visited by *all* paths from v_1 to v_n.

Unfortunately, this classification does not hold for directed graphs as the example in Figure 2.5 shows. Thus, for all arcs $(i, j) \in P(v_1, v_n)$, we have to re-compute the shortest path value when removing (i, j) from E, which may require $n - 1$ shortest path computations in the worst case.

Theorem 2.4 *On directed graphs with non-negative arc weights, relaxed L_1-consistency can be achieved in time $O(n(m + n\log n))$.*

Since the computation time of the algorithm sketched previously may not be efficient enough to be of use when being applied in a tree search, in the following we consider another shortest path relaxation. Let $T \subseteq E$ denote a shortest path tree in G rooted at v_1. Without loss of generality, we may assume that every node in G can be reached from v_1, and thus that $V \subseteq T$. Obviously, when $e \in E$ is removed from T, the nodes in V are partitioned into two sets: the set $v_1 \in S_e \subset V$ of nodes that are still connected with v_1 in $T \setminus \{e\}$, and the complement of S_e in V, S_e^C. Obviously, $S_e^C \neq \emptyset$ iff $e \in T$. We set

$$\begin{aligned} J := \{P \mid\ & P \text{ is a 2-simple path from } v_1 \text{ to } v_n \text{ with } P \subseteq pos(Y) \\ & \text{or, if } e \in P \setminus pos(Y), \text{ then there exists an arc} \\ & (i,j) \in P \setminus T \text{ such that } i \in S_e \text{ and } j \in S_e^C\}. \end{aligned}$$

Moreover, we define

$$L_2(D(Y)) := \max\{\min\{cost(P) \mid P \in J\}, \max_{f \in req(Y)}\{\min\{cost(P) \mid P \in F_f\}\},$$

To understand the above shortest path relaxation better, we make the following observations:

- Obviously, since $H \subseteq J$, L_2 is dominated by L_1, i.e., $L_2 \leq L_1$. Therefore, L_2 is also a shortest path relaxation.
- The difference between relaxations L_1 and L_2 only consists in the set J that is used instead of H to determine the arcs that have to be required to achieve a state of relaxed consistency. In contrast to H, the set J also contains paths P that are not simple (i.e., paths that may visit some nodes more than just once) and that may visit arcs $e \notin pos(Y)$. However, if $e \in P \setminus pos(Y)$, then we enforce that P must also visit another arc $(i,j) \notin T$ that connects S_e with S_e^C. This implies $e \in T$, as otherwise $S_e^C = \emptyset$. Moreover, it holds that $cost(P) \geq \min\{c(v_1,i) + c_{ij} + c(j,v_n) \mid (i,j) \in S_e \times S_e^C \setminus T\}$.
- Like L_1, L_2 also does not force the 2-admissible paths to visit the nodes in $req(Y)$ simultaneously. Again we can improve the effectivity of the filtering algorithm by setting $c_{ij} := 0$ for all $(i,j) \in req(Y)$ and by subtracting $cost(req(Y))$ from B.
- A shortest path constraint is relaxed L_2-consistent, iff
  1. for all $f \in pos(Y)$, there exists a 2-admissible path $P \in F_f$, and
  2. for all $f \in pos(Y) \setminus req(Y)$, there exists a 2-admissible path $P \in J$ with $f \notin P$, or there exists an arc $e \in P \setminus T$ such that $e \in S_f \times S_f^C$.

We have seen that the relaxation L_2 is dominated by L_1. Nevertheless, cost-based filtering that achieves relaxed L_2-consistency is still stronger than ordinary reduced-cost filtering (see Section 2.1.2):

Lemma 2.6 *If a shortest path constraint is relaxed L_2-consistent, reduced-cost filtering is ineffective.*

Proof: Let $(reqY, pos(Y))$ such that the shortest path constraint is L_2-consistent. Furthermore, denote the reduced costs of $(i,j) \in pos(Y)$ by $\overline{c}_{ij} \geq 0$.

- By assumption, for all $(i,j) \in pos(Y)$, it holds that there exists a 2-admissible path that visits (i,j). Particularly, the shortest path in $F_{(i,j)}$ is 2-admissible, i.e., $c(v_1,i) + c_{ij} + c(j,v_n) < B$. Reduced-cost filtering removes an arc $(i,j) \in pos(Y)$ from the possible set iff

$$c(v_1,v_n) + c_{ij} + c(v_1,i) - c(v_1,j) = c(v_1,v_n) + \overline{c}_{ij} \geq B.$$

  However, since $c(v_1,j) + c(j,v_n) \geq c(v_1,v_n)$, it holds that

$$c(v_1,v_n) + c_{ij} + c(v_1,i) - c(v_1,j) \leq c_{ij} + c(v_1,i) + c(j,v_n) < B.$$

- Reduced-cost filtering adds $f = (i,j) \in pos(Y) \setminus req(Y)$ to $req(Y)$ iff

$$c(v_1,v_n) + \min\{\overline{c}_{gh} \mid (g,h) \in S_f \times S_f^C\} \geq B.$$

  By assumption, for all $f = (i,j) \in pos(Y) \setminus req(Y)$, there exists a 2-admissible path P in G such that either $(i,j) \notin P$ or there exists an arc $(r,s) \in P \setminus T$ such that $(r,s) \in S_f \times S_f^C$:

  a) Let $f \notin P$. Since P is 2-admissible, it implies that there exists an admissible path (that can be constructed by removing all loops in P) that does not visit f. Thus, f must not be required.

  b) Now, let $(r,s) \in P \setminus T$ such that $(r,s) \in S_f \times S_f^C$, and $(g,h) \in S_f \times S_f^C \setminus T$ the arc with minimal reduced costs $\overline{c}_{gh} \geq 0$. Then,

$$\begin{array}{rcl} c(v_1,v_n) + \overline{c}_{gh} & \leq & c(v_1,v_n) + \overline{c}_{rs} \\ & = & c(v_1,v_n) + c_{rs} + c(v_1,r) - c(v_1,s) \\ & \leq & c_{rs} + c(v_1,r) + c(s,v_n) \\ & = & cost(P) \\ & < & B, \end{array}$$

  because $c(v_1,s) + c(s,v_n) \geq c(v_1,v_n)$.

■

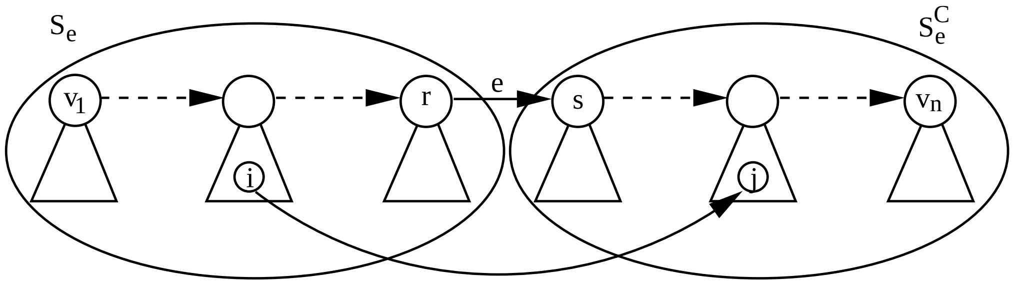


Fig. 2.6: The figure schematically shows a shortest path tree T rooted at v_1. Solid lines denote arcs in G, dashed lines mark parts of the shortest path $P(v_1,v_n)$ from v_1 to v_n. The triangles symbolize shortest path sub-trees. For an edge $e=(r,s)\in P(v_1,v_n)$, the nodes in V are partitioned into two non-empty sets S_e and S_e^C. If e is removed from the graph, the shortest path from v_1 to v_n must visit an edge $(i,j)\in S_e\times S_e^C\setminus T$.

2.2.4.1 Relaxed L_2-Consistency

As relaxations L_1 and L_2 do not differ with respect to the definition of F_f, $f\in E$, to remove arcs from $pos(Y)$ we can simply follow the procedure sketched in Section 2.2.3.

Regarding the identification of arcs that have to be added to $req(Y)$ to achieve relaxed L_2-consistency, for all $e\in pos(Y)\setminus req(Y)$, we have to compute the cost of the shortest 2-simple path P from v_1 to v_n such that $e\notin P$ or such that there exists an edge $(i,j)\in P\setminus T$ with $(i,j)\in S_e\times S_e^C$, where T is a shortest path tree in G rooted at v_1.

First, we compute the shortest paths from v_1 to v_n and v_n to v_1 in the reverse of G in time $O(m+n\log n)$. As a byproduct, we get T and shortest path distances $c(v_1,i)$, $c(i,v_n)$ for all $i\in V$. If $c(v_1,v_n)\geq B$, the current choice point is inconsistent, and we can backtrack. Otherwise, candidates to be added to $req(Y)$ are only the arcs $e\in P(v_1,v_n)$. Since $v_1\in S_e$ and $v_n\in S_e^C$, the shortest 2-simple path P from v_1 to v_n with $e\notin P$ must contain an arc $(i,j)\in S_e\times S_e^C$. Moreover, since $T\cap S_e\times S_e^C=\{e\}$, we have that $(i,j)\notin T$ (see Figure 2.6). Therefore, it is sufficient to compute, for all $e\in P(v_1,v_n)$, the costs of the shortest 2-simple path P from v_1 to v_n that contains some $(i,j)\in S_e\times S_e^C\setminus T$.

Let $P(v_1,v_n)=(r_1,r_2,\dots,r_h,r_{h+1})$, $h\in\mathbb{N}$, $r_1=v_1$ and $r_{h+1}=v_n$, and denote the sequence of arcs that $P(v_1,v_n)$ visits by $(e_1,\dots,e_h)$, whereby $e_k=(r_k,r_{k+1})$ for all $1\leq k\leq h$. Furthermore, for all $1\leq k\leq h$, let Q_k denote a shortest 2-simple path from v_1 to v_n with $(i,j)\in Q_k$ for some $(i,j)\in S_{e_k}\times S_{e_k}^C\setminus T$. Then,

$$cost(Q_k)=\min\{c(v_1,i)+c_{i,j}+c(j,v_n)\mid (i,j)\in S_{e_k}\times S_{e_k}^C\setminus T\}.$$

A brute force approach requires time $\Theta(nm)$ to determine these values. However, we can do better when we compute the values $cost(Q_k)$ for all $1\leq k\leq h$ sequentially. Note that

$$S_{e_h}^C\subseteq\cdots\subseteq S_{e_1}^C.$$

We keep the nodes j in the current set $S^C_{e_k}$ in a min-heap, whereby the associated value of j in the heap is defined as

$$x_j := \min\{c(v_1,i)+c_{i,j}+c(j,v_n) \mid i \in S_{e_k} \text{ and } (i,j) \in E \setminus T\}.$$

Obviously, the smallest x_j in the heap determines $cost(Q_k)$. In the transition from one shortest-path arc e_k to the next e_{k+1}, the nodes $i \in S_{e_k} \setminus S_{e_{k+1}}$ have to be removed from the heap, and the values x_j must be updated. For each node $i \in S_{e_k} \setminus S_{e_k+1}$, we iterate over all outgoing arcs and perform a *decrease-key* on the adjacent nodes if necessary. Then i is removed from the heap. Since every node in V leaves the heap at most once and never re-enters it, for all $1 \leq k \leq h$, this procedure requires at most m *decrease-key* operations and n *delete-min* operations. Therefore, when using a Fibonacci heap, the values $cost(Q_k)$ for all $1 \leq k \leq h$ can be determined in time $O(m+n\log n)$. Then e_k is added to $req(Y)$ iff $cost(Q_k) \geq B$. It follows:

Theorem 2.5 *On directed graphs with non-negative arc weights, relaxed L_2-consistency of a shortest path constraint can be achieved in time $O(m+n\log n)$.*

Finally, we consider the general case of directed graphs with integer arc weights that do not contain negative weight cycles. On such graphs, the Bellman-Ford algorithm computes a single source shortest path in time $O(nm)$. The shortest path distance from source to sink can be used to prune the search if that value exceeds the given bound B. However, for the purpose of cost-based filtering with respect to the relaxations L_1 or L_2, we need to compute the shortest path distances from the source and to the sink for all nodes.

Of course, we could apply the Bellman-Ford algorithm with v_1 as root in G and v_n in the reverse of G to obtain these values. To achieve relaxed L_1-consistency, this procedure would require time $\Theta(n^2m)$ in the worst case. We can do much better though, especially when taking into account that within a tree search, many similar Shortest Path Problems have to be solved. We can speed up the computation by using node potentials h_v for all $v \in V$. It is a well-known fact, that the shortest path structure of a graph is maintained when the arc weights are changed to $\overline{c}_{ij} = c_{ij} + h_i - h_j$ [1]. We aim at finding node potentials h such that $\overline{c} \geq 0$. Then, even after arcs have been removed from the graph or the shortest path is required to visit certain arcs, we can simply apply the algorithms that we developed for directed graphs with non-negative arc weights. The only necessary modification is to compute $c(i,j) = \overline{c}(i,j) - h_i + h_j$.

In order to compute the desired node potentials, we use a method that has been developed for the computation of all pairs shortest paths by Johnson [43]: we add an artificial source node s and arcs (s,i) for all $i \in V$, and we set $c_{si} := 0$. If the given graph does not contain negative weight cycles, the Bellman-Ford algorithm produces shortest path distances $c(s,i)$. For all arcs $(i,j) \in E$, we have that $c(s,j) \leq c(s,i) + c_{ij}$. Thus, when setting $h_i := c(s,i)$ we get

$$\overline{c}_{ij} = c_{ij} + h_i - h_j = c_{ij} + c(s,i) - c(s,j) \geq 0.$$

	Degree of Consistency			
Graph Type	ArcCon	L_1	L_2	RedCost
DAG	$O(m+n)$			
undirected, $c \geq 0$	NP-hard	$O(m+n\log n)$, $[RAM]O(m+n)$		
directed, $c \geq 0$	NP-hard	$O(n(m+n\log n))$	$O(m+n\log n)$	
directed, no negative cycles	NP-hard	$O(n(m+n\log n))$	O(nm) amort.$[\Omega(n)]$: $O(m+n\log n)$	

Tab. 2.1: The table gives an overview of the fi ndings in this section.

The following two theorems follow directly from the discussion:

Theorem 2.6 *On directed graphs, relaxed L_1-consistency of a shortest path constraint can be achieved in time $O(n(m+n\log n))$.*

Theorem 2.7 *On directed graphs, relaxed L_2-consistency of a shortest path constraint can be achieved in time $O(nm)$. Within a tree search, relaxed L_2-consistency can be achieved in amortized time $O(m+n\log n)$ for $\Omega(n)$ calls of the filtering procedure.*

2.2.5 Summary

Before we proceed, we summarize the results that we achieved in this section (see Table 2.1): On DAGs, arc-consistency for a shortest path constraint can be achieved in linear time by exploiting topological orderings. On general directed and on undirected graphs, achieving arc-consistency is an NP-hard task. We developed two shortest path relaxations L_1 and L_2 both based on the class of 2-simple paths. We showed that L_1 dominates L_2, and cost-based filtering based on L_2 is superior to reduced-cost filtering. On undirected graphs with non-negative edge weights, relaxed L_1-consistency (and therefore also relaxed L_2-consistency) can be achieved in time $O(m+n\log n)$ and in time $O(m+n)$ on a RAM. On directed graphs with non-negative arc weights, relaxed L_1-consistency can be obtained in time $O(n(m+n\log n))$, and a state of relaxed L_2-consistency can be achieved in time $O(m+n\log n)$. Finally, in the presence of negative arc weights, we use the Bellman-Ford algorithm just once for the computation of node potentials that allow us to solve the Shortest Path Problems on graphs with non-negative arc weights. Therefore, we achieve relaxed L_1-consistency in time $O(n(m+n\log n))$, and L_2-consistency in time $O(nm)$ or $O(m+n\log n)$ for $\Omega(n)$ calls of the filtering algorithm.

2.3 Weighted Stable Set Constraints on Interval Graphs

Real-world scheduling problems often require the selection of temporally non-overlapping tasks, as one machine, processor or person can only work on one job at a time. E.g., if one wants to record movies from TV, no two temporally overlapping broadcasts can be taped. Thus, when given a set of weighted tasks with starting and ending times, we try to find a selection of non-overlapping tasks such that their weighted sum is minimized [43][2].

Frequently, the problem evolves only as relaxation or sub-problem of a real-world application. For instance, in a realistic model of the above TV recording example, the storage capacity of the recording device is limited (see Chapter 6). The problem can viewed as an augmented Knapsack Problem, which is NP-hard. Exact algorithms to compute and prove an optimal solution for such problems are often based on enumeration approaches. The tightening of sub-problems can help greatly to improve the performance of a tree search approach. Therefore, in this section we develop an efficient cost-based filtering algorithm that exploits the special structure of non-overlapping constraints.

During a tree search, many similar problem instances have to be solved, whereby the instances from one iteration to the other only differ with respect to necessarily included and excluded tasks and, as we shall see later, possibly changes in the objective. Therefore, the development of an incremental algorithm is desirable [8, 54, 177, 178]. The algorithm we develop works in two phases: a preprocessing phase using time $\Theta(n \log n)$, and an optimization and filtering phase using linear time. The data structure established in the first phase is independent of the objective function and can be adapted in linear time to reflect decisions on necessarily included and excluded tasks. Thus, we achieve an amortized linear time algorithm for $\Omega(\log n)$ incremental calls with changing variable domains and different objectives.

Repeated computations with changing objective functions are important when solving Lagrangian relaxations for example. In Section 3.2, we develop a method to link linear optimization constraints that is based on Lagrangian relaxation, and that makes use of dual and reduced-cost information while solving the Lagrangian dual. Therefore, as a major objective in this section, we develop an efficient algorithm that, on top of an optimal selection of non-overlapping tasks, provides dual and reduced-cost data as a byproduct.

The work presented in this section was published in [189]. It is structured as follows: In Section 2.3.1, we define the weighted stable set constraint formally. Then, in Section 2.3.2, we develop an algorithm based on mathematical programming that computes minimum weighted stable sets on interval graphs and provides dual and reduced-costs information as a byproduct.

[2]In contrast to the common defi nition of weighted stable set problems, we state the problem not as a maximization but as a minimization problem. We do this because the latter problem has a one-to-one correspondence with a shortest-path problem in the complement graph that we will use for fi ltering purposes later. Note that we allow negative node weights such that a maximization problem can easily be transformed into a minimization instance.

Finally, in Section 2.3.3, we give a cost-based filtering algorithm for weighted stable set constraints on interval graphs.

2.3.1 The Weighted Stable Set Constraint

A natural way of modeling the problem of finding a selection of non-overlapping tasks is to consider an interval graph [99]: the tasks are represented by the nodes, and an edge connects two nodes iff the corresponding tasks are in conflict, i.e., iff the corresponding intervals are overlapping.

Definition 2.6 *A graph $G = (V,E)$ is called an* interval graph *iff there exist intervals $I_1,\dots,I_{|V|} \subset \mathbb{Q}$ such that $\forall\, v_i, v_j \in V : \{v_i, v_j\} \in E \iff I_i \cap I_j \neq \emptyset$.*

Then the problem consists in finding a minimum weighted stable set (WSSP) in an interval graph. We generalize the use of conflict graphs and define:

Definition 2.7 *Given an undirected graph $G = (V,E)$, $n = |V|$, $V = \{1,\dots,n\}$, denote the node weights in G by $c \in \mathbb{Q}^n$, and let $B \in \mathbb{Q}$. Let $X_i \in \{0,1\}$ denote binary variables for all $1 \leq i \leq n$. Then a* weighted stable set constraint *has variables $\{X_1,\dots,X_n\}$, and an instantiation $X_i = x_i$ for all $1 \leq i \leq n$ is true, iff*

- *for all $1 \leq i < j \leq n$, it holds that $x_i = 1 = x_j$ implies $\{i,j\} \notin E$, and*
- *$\sum_{x_i=1} c_i < B$.*

Obviously, the weighted stable set constraint is a minimization constraint. On general graphs, the computation of a minimum stable set is an NP-hard task. Therefore, achieving arc-consistency for the weighted stable set constraint on general graphs is also NP-hard. However, on interval graphs minimum weighted stable sets can be computed in time $O(n\log n)$ [115]. The existing algorithms for the WSSP on interval graphs are based on sweep line or dynamic programming approaches and neither provide dual values and reduced-cost information, nor do they suggest how cost-based filtering could be performed efficiently.

We will show that a state of arc-consistency can be achieved in amortized linear time for $\Omega(\log n)$ incremental calls for weighted stable set constraints on interval graphs. In the following, we assume that we are given a number $n \in \mathbb{N}$, intervals $I_i = [start(i), end(i)]$ for all $1 \leq i \leq n$, task weights $c \in \mathbb{Q}^n$ and an upper bound B on the objective. We refer to the corresponding interval graph with $G = (V,E)$ whereby $V = \{1,\dots,n\}$ and $E = \{\{i,j\} \mid I_i \cap I_j \neq \emptyset\}$.

2.3.2 A Mathematical Programming Approach

We present an algorithm based on mathematical programming for the Minimum Weighted Stable Set Problem on interval graphs that provides us with dual and reduced-cost information as a byproduct, and that will be extended to an efficient cost-based filtering for the problem later.

Obviously, the following Integer Program solves the WSSP on interval graphs:

$$\begin{array}{lrcll} \text{Minimize} & \multicolumn{3}{l}{IP_1 = \sum_{1\le i\le n} c_i x_i} & \\ \text{subject to} & x_i + x_j & \le & 1 & \forall\, 1 \le i < j \le n,\ I_i \cap I_j \neq \emptyset \\ & x & \in & \{0,1\}^n & \end{array}$$

In this formulation, an LP relaxation does not necessarily yield an integer solution. However, we can tighten the problem formulation such that every LP-solution is already integer. To achieve that formulation, we introduce a few more definitions:

Definition 2.8 *A set $C \subseteq V$ is called a* conflict clique, *iff $I_i \cap I_j \neq \emptyset\ \forall\, i, j \in C$. A conflict clique C is called* maximal, *iff $\forall\, D \subseteq V$, 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 in G. We set $max\_start : M \to \mathbb{N}$, $max\_start(C) := \max_{i\in C}\{start(i)\}$.*

Remark 2.1 *$\bigcup_{1\le p\le m} C_p = V$, because $\{i\}$ is a conflict clique $\forall\ i \in V$. Thus, there exists a maximal conflict clique C_p, $1 \le p \le m$, such that $i \in C_p$.*

Lemma 2.7 *The function max_start is injective.*

Proof: Assume $max\_start(C_p) = max\_start(C_q)$ for some $1 \le p, q \le m$. Then there exist nodes $s_p \in C_p$ and $s_q \in C_q$ such that

$$start(s_p) = max\_start(C_p) = max\_start(C_q) = start(s_q),$$

and $\forall\ i \in C_p, j \in C_q$:

$$end(i) \ge start(s_p) = start(s_q) \ge start(j)$$

and

$$start(i) \le start(s_p) = start(s_q) \le end(j).$$

Thus, all nodes in C_p and C_q are pairwise overlapping. Therefore, $C_p \cup C_q$ is a conflict clique. As C_p and C_q are maximal, we have $C_p = C_p \cup C_q = C_q$. However, as $p \neq q$ implies $C_p \neq C_q$, we have $p = q$. ■

Thus, without loss of generality, in the following we assume that the conflict cliques are ordered with respect to *max_start*, i.e.

$$max\_start(C_p) < max\_start(C_q) \qquad \forall\, 1 \le p < q \le m.$$

Lemma 2.8 *Let $1 \leq p < r \leq m$ and $i \in C_p \cap C_r$. Then $i \in C_q \; \forall \; p < q < r$.*

Proof: Let $s_p \in C_p, s_q \in C_q, s_r \in C_r$, such that $start(s_t) = max\_start(C_t) \; \forall \; t \in \{p,q,r\}$. Furthermore, let $j \in C_q$. Then,

$$end(i) \geq start(s_r) > start(s_q) \geq start(j)$$

and

$$start(i) \leq start(s_p) < start(s_q) \leq end(j).$$

Therefore, $C_q \cup \{i\}$ is a conflict clique. As C_q is maximal, $C_q = C_q \cup \{i\}$, i.e. $i \in C_q$. ■

Corollary 2.2 *$m \leq n$.*

Proof: Let $1 \leq p < m$. $\exists \; i \in C_p$ such that $i \notin C_{p+1}$, as otherwise $C_p \subseteq C_{p+1}$, which contradicts the maximality of C_p or $C_p \neq C_{p+1}$. Thus, with Lemma 2.8, we have $i \notin C_q \; \forall \; p < q \leq m$. Therefore, $|C_q| \leq n+1-q \; \forall \; 1 \leq q \leq m$, and thus $m \leq n$. ■

Definition 2.9 *We set $R_p := C_p \setminus C_{p+1} \; \forall \; 1 \leq p < m$ and $R_m := C_m$, and call every such R_p a* (max_start) rest clique.

Remark 2.2 *The rest cliques form a partition of V: Let $1 \leq i \leq n$. Remark 2.1 states that $i \in C_p$ for some $1 \leq p \leq m$. Let $q := \max\{p \mid 1 \leq p \leq m, \; i \in C_p\}$. Then, $i \in R_q$.*

On the other hand, let $i \in R_p \cap R_q$ with $1 \leq p < q \leq m$. Then, $i \in C_p \cap C_q$. Therefore, with Lemma 2.8, we have $i \in C_{p+1}$, which is a contradiction to $i \in R_p$.

Let $C_1, \ldots, C_m$ denote the maximal conflict cliques of G ordered according to max_start, and consider the following integer program:

$$\begin{array}{lrcll} \text{Minimize} & \multicolumn{3}{l}{IP_2 = \sum_{1 \leq i \leq n} c_i x_i} & \\ \text{subject to} & \sum_{i \in C_p} x_i & \leq & 1 & \forall \; 1 \leq p \leq m \\ & x & \in & \{0,1\}^n & \end{array}$$

The maximal conflict clique restrictions 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 I_i and I_j, it is also true that $\sum_{i \in C_p} x_i \leq 1 \; \forall \; 1 \leq p \leq m$. Thus, the above IP solves the WSSP on interval graphs.

In the following, by $A \in \{0,1\}^{m \times n}$ we denote the corresponding matrix to IP_2, i.e. $A = (a_{pi})_{1 \leq p \leq m, 1 \leq i \leq n}$ with $a_{pi} = 1$ iff $i \in C_p$.

Theorem 2.8 *The corresponding matrix A of IP_2 is an interval matrix.*

Proof: We have to show that $a_{pi} = a_{ri} = 1$ implies that $a_{qi} = 1 \; \forall \; p < q < r$, $1 \leq i \leq n$. By the construction of A, this is equivalent to showing that $i \in C_p \cap C_r$ implies $i \in C_q \; \forall \; p < q < r$. However, this is true according to Lemma 2.8. ■

Corollary 2.3 *IP_2 is totally unimodular.*

Proof: Interval matrices are totally unimodular [158]. ■

Corollary 2.3 now allows to solve the WSSP on interval graphs as a linear program:

$$\begin{array}{lrcll} \text{Minimize} & LP_3 = \sum_{1 \leq i \leq n} c_i x_i & & & \\ \text{subject to} & \sum_{i \in C_p} x_i & \leq & 1 & \forall \, 1 \leq p \leq m \\ & x & \geq & 0 & \end{array}$$

Notice that, with Remark 2.1, the maximal conflict clique restrictions imply that $x \leq 1$.

2.3.2.1 A Pivot Selection Strategy

We use the simplex method to solve LP_3. Let $R_1, \ldots, R_m$ denote the (max_start) rest cliques of G. In iteration $1 \leq t \leq m$, we choose $q := t$ as pivot row and $j \in R_q$ with the smallest reduced costs as pivot column. If the reduced costs of j are less than 0, we perform a pivot step. Otherwise we proceed with the next iteration immediately.

Theorem 2.9 *After m such iterations, the simplex tableau is primal and dual feasible.*

The proof of Theorem 2.9 will be given later in this section. In the following, we refer to the simplex tableau with the following identifiers: elements of the matrix $A^t \in \{-1, 0, 1\}^{m \times n}$ after $0 \leq t \leq m$ simplex iterations are denoted by $(a^t_{pi})_{1 \leq p \leq m, 1 \leq i \leq n}$, entries in the right hand side $b^t \in \mathbb{Q}^m$ are referred to by $(b^t_p)_{1 \leq p \leq m}$, and the reduced costs $\overline{c}^t$ are denoted by $(\overline{c}^t_i)_{1 \leq i \leq n}$.

First, we prove that our pivot selection preserves primal feasibility. We observe that $x = 0$ is primal feasible as $b^0_p = 1 \geq 0$, $\forall \, 1 \leq p \leq m$. To assure the maintenance of primal feasibility, we must show that $b^t \geq 0$, $\forall \, 0 \leq t \leq m$. To do so, we prove the following

Lemma 2.9 *Let $0 \leq t \leq m$, $1 \leq p \leq m$, $1 \leq i \leq n$. Then,*

(a) $p \geq t$ implies that $a^t_{pi} = a^0_{pi}$ and $b^t_p = b^0_p = 1$.

(b) $b^t_p = 0$, $i \in \bigcup_{r>t} R_r$ implies $a^t_{pi} \in \{-1, 0\}$.

(c) $b^t_p = 1$, $i \in \bigcup_{r>t} R_r$ implies $a^t_{pi} \in \{0, 1\}$.

(d) $b^t_p \in \{0, 1\}$.

Proof: We induce over t:
$\mathbf{t = 0}$: $b_p^0 = 1$ and $a_{pi}^0 \in \{0,1\}\ \forall\ 1 \le p \le m$ and $1 \le i \le n$.
$\mathbf{t \to t+1}$: Let $t < m$, and denote the pivot column in iteration $t+1$ by $j \in R_{t+1}$. If the reduced costs of column j are greater or equal zero, then we are done since $b^{t+1} = b^t$ and $A^{t+1} = A^t$. Otherwise, we set $q := t+1$ and choose a_{qj}^t as pivot element. By induction hypothesis (a), we know that $b_q^t = b_q^0 = 1$, and that $a_{qj}^t = a_{qj}^0$. Now, since $j \in R_q \subseteq C_q$, $a_{qj}^0 = 1$. Thus our pivot element is equal to 1.

(a) Therefore, $a_{qi}^{t+1} = a_{qi}^t = a_{qi}^0$ and $b_q^{t+1} = b_q^t = b_q^0 = 1$ for all $1 \le i \le n$. Now let $t+1 < p \le m$. According to our pivot selection strategy,

$$j \in R_q = C_q \setminus C_{q+1} \Rightarrow j \notin C_{q+1}.$$

This, and the interval property of the matrix A imply

$$a_{pj}^t = a_{pj}^0 = 0 \qquad \text{for } p = t+2, \text{ and thus for all } p > t+1.$$

Thus, in iteration $t+1$ the rows $p > t+1$ do not change, i.e.

$$a_{pj}^{t+1} = a_{pj}^t = a_{pj}^0 \qquad \text{and} \qquad b_p^{t+1} = b_p^t = b_p^0.$$

For $p \ge t+1$, (a) implies (b)–(d). Thus, in the following we assume $p \le t$. Since, for all $p \le t$ with $a_{pj}^t = 0$, row p does not change in iteration $t+1$, we only need to consider $p \le t$ with $a_{pj}^t \ne 0$. Then, as the matrix A is totally unimodular, it holds that $a_{pj}^t \in \{-1,1\}$.

Finally, let $i \in \bigcup_{r>t} R_r$ with $a_{qi}^t = a_{qi}^0 = 0$. Due to the interval matrix property of A and $a_{ri}^0 = 1$ for some $r > t+1 = q$, we know then that $a_{ri}^0 = 0\ \forall\ r \le q$. Moreover, as all pivot elements up to step t were chosen from rows lower than q, it holds that $a_{ri}^t = a_{ri}^0 \in \{0,1\}$, and $b_r^t = b_r^0 = 1$ for all $q < r \le m$. Therefore, we only need to consider $a_{qi}^t = a_{qi}^0 = 1$.

(b–d) Let $p \le t$, $i \in \bigcup_{r>t} R_r$, $a_{qi}^t = 1$ and $a_{pj}^t \in \{-1,1\}$. Using induction hypothesis (d), we know that $b_p^t \in \{0,1\}$. First, assume that $b_p^t = 0$. By induction hypothesis (b), we know that $a_{pj}^t = -1$ and $a_{pi}^t \in \{-1,0\}$. Thus,

$$b_p^{t+1} = b_p^t + 1 = 1 \in \{0,1\} \qquad \text{and} \qquad a_{pi}^{t+1} = a_{pi}^t + 1 \in \{0,1\}.$$

Now let us assume $b_p^t = 1$. Then, by induction hypothesis (c), we know that $a_{pj}^t = 1$, and $a_{pi}^t \in \{0,1\}$. Thus,

$$b_p^{t+1} = b_p^t - 1 = 0 \in \{0,1\} \qquad \text{and} \qquad a_{pi}^{t+1} = a_{pi}^t - 1 \in \{-1,0\}.$$

■

Now we show that after m iterations we achieve dual feasibility.

Lemma 2.10 *Let $1 \le t \le m$. Then,*

(a) $\overline{c}_i^t \ge 0$ for all $i \in R_t$.

(b) If $t < m$, then $i \in \bigcup_{1 \le p < t} R_p$ implies $\overline{c}_i^{t+1} = \overline{c}_i^t$.

Proof: (a) Let $j \in R_t$ denote the pivot column in iteration t. If $\overline{c}_j^{t-1} \ge 0$ we are done, as then $\overline{c}_i^t = \overline{c}_i^{t-1} \ge \overline{c}_j^{t-1} \ge 0$ for all $i \in R_t$. So let us assume $\overline{c}_j^{t-1} < 0$. In Lemma 2.9, we have already shown that the pivot element is $a_{tj}^{t-1} = 1$, and

$$a_{ti}^{t-1} = a_{ti}^0 \qquad \forall\, 1 \le i \le n.$$

In particular, we know that

$$a_{ti}^{t-1} = a_{ti}^0 = 1 \qquad \forall\, i \in R_t \subseteq C_t.$$

We conclude that

$$\overline{c}_i^t = \overline{c}_i^{t-1} - \overline{c}_j^{t-1} \ge 0.$$

(b) Let $1 \le p \le t < m$ and $i \in R_p$. Lemma 2.9 states that $a_{ri}^t = a_{ri}^0$ for all $t < r \le m$. Then, due to the interval matrix property of A, and $i \in R_p = C_p \setminus C_{p+1}$, we know that $a_{t+1,i}^t = a_{t+1,i}^0 = 0$. Therefore, $\overline{c}_i^{t+1} = \overline{c}_i^t$. ■

Corollary 2.4 *After m iterations we achieve dual feasibility.*

Proof: Let $1 \le i \le n$, and $1 \le t \le m$ such that $i \in R_t$. With Lemma 2.10, it holds that:

$$0 \le \overline{c}_i^t = \overline{c}_i^{t+1} = \cdots = \overline{c}_i^m.$$

■

Now, we give the previously postponed

Proof of Theorem 2.9: In Lemma 2.9 and Corollary 2.4, we have shown that after $m \le n$ iterations the simplex tableau is primal and dual feasible. ■

2.3.2.2 An Efficient Simplex Realization

We have shown how the WSSP on interval graphs can be stated as a totally unimodular LP. Moreover, we have proven a feasible pivot selection strategy that yields an optimal tableau after at most n simplex iterations.

In the following, we develop an efficient $\Theta(n \log n)$-time algorithm to compute a set $Q \subseteq \{1,\dots,n\}$ with $I_i \cap I_j = \emptyset\ \forall\ i,j \in Q$, such that $cost(Q) := \sum_{i\in Q} c_i$ is minimal. Most importantly, the algorithm also provides us with dual and reduced-cost information as a byproduct. To establish that algorithm, we show how the simplex computations according to the pivot strategy developed in the previous section can be performed efficiently.

Theorem 2.10 *Let $(j_1,\dots,j_m) \in R_1 \times \cdots \times R_m$ denote the sequence of pivot columns according to Section 2.3.2.1, and let $d : \{1,\dots,m\} \to \{0,1\}$ with $d(t) := 1$ iff $\overline{c}^t_{j_t} < 0$ for all $1 \le t \le m$.*

Then the set $Q := \{j_t \mid 1 \le t \le m$ with d(t)=1 and $I_{j_t} \cap I_{j_r} = \emptyset\ \forall\ t < r \le m\}$ is a stable set in V with minimal costs.

Proof: If no pivoting is taking place ($d(t) = 0$ for all $1 \le t \le m$), the initial tableau is optimal with $x = 0$. Therefore, $Q = \emptyset$ is an optimal solution to the problem. So let us assume that $D := \{t \mid 1 \le t \le m \text{ and } d(t) = 1\} \neq \emptyset$.

We induce over m:
$m = 1$: If $D \neq \emptyset$ and there exists only one row, exactly one pivot step is being carried out. Then x_{j_1} is the only basic variable, and according to Lemma 2.9 it holds that: $x_{j_1} = b^1_1 = b^0_1 = 1$. Thus, $Q = \{j_1\}$ is an optimal solution.
$m \to m+1$: Now assume that there are $m+1$ maximal conflict cliques. We set $l := \max_{t\in D} t$. Again, by applying Lemma 2.9, we know that $b^{m+1}_l = b^l_l = b^0_l = 1$. Therefore, there exists an optimal solution

$$Q \subseteq \{j_t \mid 1 \le t \le l \text{ with } d(t) = 1\}, \quad \text{with} \quad j_l \in Q.$$

Let $k := \min\{t \mid 1 \le t \le l,\ j_l \in C_t \text{ and } d(t) = 1\}$. When setting $N := \bigcup_{k\le t\le m} C_t$, we know that $N \cap Q = \emptyset$. Thus, there exists an optimal solution $Q = \{j_l\} \cup S$, where

$$S \subseteq V \setminus N = \textstyle\bigcup_{1\le t<k} R_t, \quad \text{and it holds that} \quad I_i \cap I_j = \emptyset \qquad \forall\ i,j \in S.$$

Since $cost(Q) = c_{j_l} + cost(S)$, we can construct an optimal solution by finding such a set S with a minimal value $cost(S)$. If $k = 1$, it holds that $S = \emptyset$, and we are done. Otherwise, we have to solve a WSSP on an interval graph with $k-1 \le m$ maximal conflict cliques to find such a set S. By setting up the corresponding LP, we find that the sequence of pivot elements to solve this problem is exactly $(j_1,\dots,j_{k-1})$, and pivot steps are carried out for all $1 \le t < k$ with $d(t) = 1$. We apply our induction hypothesis and achieve

$$S = \{j_t \mid 1 \le t < k \text{ with } d(t) = 1 \text{ and } I_{j_t} \cap I_{j_s} = \emptyset \; \forall\, t < s < k\}.$$

Thus, $Q = \{j_l\} \cup S = \{j_t \mid 1 \le t \le m+1 \text{ with } d(t) = 1 \text{ and } I_{j_t} \cap I_{j_s} = \emptyset \; \forall\, t < s \le m+1\}$ is an optimal solution to the WSSP on interval graphs. ■

The above Theorem 2.10 allows us to construct an optimal solution if we know the sequence of pivot elements: All we have to do is start with $Q = \emptyset$. Then we visit all pivot elements in the reverse order. An element j_t is added to Q iff $d(t) = 1$ and its corresponding interval does not overlap with any corresponding interval of an element in Q. This last check can be performed efficiently by maintaining the value $\min_{j \in Q} f_j$, whereby $f_i := \min\{p \mid 1 \le p \le m,\ i \in C_p\} \;\forall\, 1 \le i \le n$ denotes the index of the first maximal conflict clique that node i belongs to.

It remains to compute the sequence of pivot columns for which a pivot step is being carried out. However, according to Section 2.3.2.1, this is an easy task if we can only determine the reduced costs quickly:

Lemma 2.11 *Let $1 \le t \le m$, $(j_1, \ldots, j_m) \in R_1 \times \cdots \times R_m$ be the sequence of pivot columns according to our pivot selection strategy, and let $d : \{1, \ldots, m\} \to \{0,1\}$ with $d(t) := 1$ iff $\overline{c}^t_{j_t} < 0$. Furthermore, we set $q_t := t$. Let $z^t \in \mathbb{Q}$ denote the objective function value after iteration t ($z^0 := 0$), and $a^{t-1}_{q_t j_t} = 1$ be the pivot element in iteration t. Finally, for all $1 \le i \le n$, we set $g^t_i := z^{f_i - 1} - z^t$ if $f_i \le t$, and $g^t_i := 0$ otherwise. Then,*

(a) $z^t = z^{t-1} + \overline{c}^{t-1}_{j_t} \cdot d(t) \le z^{t-1}$.

(b) $z^t = \sum_{1 \le r \le t} \overline{c}^{r-1}_{j_r} \cdot d(r)$.

(c) $\overline{c}^t_i = c_i + g^t_i \qquad \forall\, i \in \bigcup_{t \le p \le m} R_p$.

Proof: (a) If no pivoting takes place in iteration t, then $\overline{c}^{t-1}_{j_t} \ge 0$ and $d(t) = 0$, and $z^t = z^{t-1}$. Otherwise, $\overline{c}^{t-1}_{j_t} < 0$ and $d(t) = 1$. According to Lemma 2.9, $a^{t-1}_{q_t j_t} = a^0_{q_t j_t} = 1$, and $b^{t-1}_{q_t} = b^0_{q_t} = 1$. Thus, $z^t = z^{t-1} + \overline{c}^{t-1}_{j_t} \cdot b^{t-1}_{q_t} / a^{t-1}_{q_t j_t} = z^{t-1} + \overline{c}^{t-1}_{j_t} < z^{t-1}$.
(b) With $z^0 = 0$, (b) is a simple implication of (a).
(c) Let $t \le p \le m$ and $i \in R_p$. First, assume that $f_i > t$. Then, $a^0_{ri} = 0 \;\forall\, 1 \le r \le t$. As all pivot elements up to step t were chosen from rows $1 \le r \le t$, we have that $\overline{c}^t_i = \overline{c}^0_i = c_i + g^t_i$.

So let $f_i \le t$. Using (b), we see that $g^t_i = z^{f_i - 1} - z^t = -\sum_{f_i \le r \le t} \overline{c}^{r-1}_{j_r} \cdot d(r)$. As $a^0_{ri} = 0 \;\forall\, 1 \le r < f_i$, we know that $\overline{c}^{f_i - 1}_i = c_i$. Now let $f_i \le r \le t \le p$. With Lemma 2.9, it holds that $a^{r-1}_{q_r j_r} = a^0_{q_r j_r} = 1$, and also $a^{r-1}_{q_r i} = a^0_{q_r i} = 1$. Thus, $\overline{c}^r_i = \overline{c}^{r-1}_i - \overline{c}^{r-1}_{j_r} \cdot d(r)$, and hence $\overline{c}^t_i = \overline{c}^{f_i - 1}_i - \sum_{f_i \le r \le t} \overline{c}^{r-1}_{j_r} \cdot d(r) = c_i + g^t_i$. ■

2.3.2.3 An Algorithm providing Dual Information

With Theorem 2.10 and Lemma 2.11, we can formulate an efficient algorithm solving the WSSP on interval graphs that provides us with dual values as a byproduct. In phase 1, we determine the (max_start) rest cliques R_p, with $1 \leq p \leq m$, and the corresponding values $f_i \; \forall \; 1 \leq i \leq n$. This can be done in time $\Theta(n \log n)$.

Phase 2 consists of m iterations: First, we set $z^0 := 0$. In each iteration $1 \leq t \leq m$ we compute $\overline{c}_i^{t-1} = c_i + z^{f_i - 1} - z^{t-1} \; \forall \; i \in R_t$, and $j_t \in R_t$ with $\overline{c}_{j_t}^{t-1} = \min_{i \in R_t}\{\overline{c}_i^{t-1}\}$. If $\overline{c}_{j_t}^{t-1} \geq 0$, we set $z^t := z^{t-1}$, otherwise $z^t := z^{t-1} + \overline{c}_{j_t}^{t-1}$. Finally, we set $t := t+1$ and proceed to the next iteration.

With Remark 2.2, we know that the sets R_p form a partition of V. Thus, all nodes $1 \leq i \leq n$ are being looked at exactly once to compute the reduced costs. Also, in all computations of the pivot columns, each node is incorporated only once. Therefore, phase 2 takes time $\Theta(n)$.

After m iterations, we know the value z^m, as well as the sequence $(j_1, \ldots, j_m) \in R_1 \times \cdots \times R_m$ of pivot columns and the function d. By applying Theorem 2.10, we can construct a stable set out of this information in linear time. Since the rest cliques of the underlying interval graph are independent of the objective function, we achieve an incremental linear time algorithm for $\Omega(\log n)$ calls with different objectives.

Most importantly, we get dual values as a byproduct. By looking at the optimal tableau, we find that the optimal dual variable for each maximal clique constraint $1 \leq t \leq m$ has the value $-\overline{c}_{j_t}^m \cdot d(t)$.

2.3.3 Cost-based Filtering

After having developed an algorithm that solves the WSSP on interval graphs, we now give an efficient filtering algorithm for the corresponding constraint. Unlike the Shortest Path Problem, that can also be solved in polynomial time, but for which achieving a state of arc-consistency is NP-hard, the Weighted Stable Set Problem exhibits a stable substructure: when any node is removed from or added to the stable set, the remaining problem is again a Weighted Stable Set Problem. In our case, the sub-problem can even be represented as a WSSP on a (modified) interval graph. Therefore, a simple arc-consistency algorithm can be obtained by probing all variable values using the previously developed algorithm, which requires time $\Theta(n^2)$.

In the following, we develop a cost-based filtering algorithm for the WSSP on interval graphs that achieves a state of arc-consistency in amortized linear time for $\Omega(\log n)$ calls to the routine with changing objectives and/or variable domains. To develop that algorithm, we re-interpret the problem as finding a shortest path in a directed, acyclic and node-weighted co-interval graph:

We introduce an artificial source σ and an artificial sink τ with corresponding intervals before and after all other nodes, and with $c_\sigma := 0$ and $c_\tau := 0$. Set $\overline{G} = (N, A)$ with $N := V \cup \{\sigma, \tau\}$ and $A := \{(i,j) \mid i,j \in N, \; end(i) < start(j)\}$. We then define $\pi(\sigma, \tau)$ as the set of simple paths from

σ to τ in $\overline{G}$. The cost of a path $P \in \pi(\sigma,\tau)$ is defined as $cost(P) := \sum_{i \in P} c_i$.

Remark 2.3 *There is a one-to-one correspondence between stable sets in G and paths $P \in \pi(\sigma,\tau)$ in $\overline{G}$:*

- *Let $Q = \{i_1,\dots,i_l\} \subseteq V$ denote a stable set in G. Without loss of generality, we may assume that $start(i_j) < start(i_k)$ for all $1 \le j < k \le l$. Then, since Q is a stable set, it even holds that $end(i_j) < start(i_k)$ for all $1 \le j < k \le l$. Therefore, $P := (\sigma, i_1,\dots,i_l,\tau)$ is a simple path from σ to τ in $\overline{G}$ with $cost(P) = \sum_{h \in P} c_h = 0 + \sum_{1 \le j \le l} c_{i_l} + 0 = cost(Q)$.*
- *On the other hand, if $P = (\sigma, i_1,\dots,i_l,\tau) \in \pi(\sigma,\tau)$, then $end(i_j) < start(i_k)$ for all $1 \le j < k \le l$. Therefore, the set $Q := \{i_1,\dots,i_l\}$ is a stable set in G, and $cost(Q) = \sum_{1 \le j \le l} c_{i_l} = \sum_{h \in P} c_h = cost(P)$.*

Therefore, a minimum weighted stable set in G corresponds to a shortest path from σ to τ in $\overline{G}$, and both have the same costs.

Given an upper bound $B \in \mathbb{Q}$, we define

$$\begin{aligned} Rem(B) &:= \{1 \le i \le n \mid \forall\, P \in \pi(\sigma,\tau), i \in P:\ cost(P) \ge B\}, \text{ and} \\ Req(B) &:= \{1 \le i \le n \mid \forall\, P \in \pi(\sigma,\tau), i \notin P:\ cost(P) \ge B\}. \end{aligned}$$

Then, with Remark 2.3, to achieve a state of arc-consistency for a weighted stable set constraint on an interval graph, we need to remove the value 1 from the domain of X_i iff $i \in Rem(B)$, and we have to remove the value 0 from the domain of X_i iff $i \in Req(B)$. Since the variables are all binary, this corresponds to setting $X_i = 0$ iff $i \in Rem(B)$, and $X_i = 1$ iff $i \in Req(B)$.

2.3.3.1 Removing Nodes

To compute $Rem(B)$, it is sufficient to determine the values of the shortest paths from the source σ via node j to the sink τ for all $j \in \{1,\dots,n\}$. This can be done by computing the shortest-path distances from the source and to the sink (compare with Section 2.2.2). With Remark 2.3, the shortest path from σ to a node $j \in \{1,\dots,n\}$ can be determined by solving a WSSP on the reduced interval graph with node set $\bigcup_{1 \le p < f_j} C_p$, i.e., by solving the following LP:

$$\begin{array}{lrcl} \text{Minimize} & LP_4^j = \sum_{i \in C_p,\, p < f_j} c_i x_i & & \\ \text{subject to} & \sum_{i \in C_p} x_i \ \le & 1 & \forall\, 1 \le p < f_j \\ & x \ \ge & 0 & \end{array}$$

Then the shortest-path value from σ to j is $c(\sigma, j) = LP_4^j + c_j$. According to the previously developed theory, the minimal objective for the above LP_4^j is exactly $z^{f_j - 1}$. Thus, the shortest-path distance of node j is $c(\sigma, j) = z^{f_j-1} + c_j$.

A similar theory as presented before shows that the shortest-path distances to the sink can be determined by applying the algorithm of Section 2.3.2.3 using the last clique belongings

$$l_i := \max\{p \mid 1 \leq p \leq m,\ i \in C_p\} \qquad \forall\, 1 \leq i \leq n,$$

and the *min_end* rest cliques, where

$$min\_end : M \to \mathbb{N},\ min\_end(C) := \min_{i \in C}\{end(i)\}.$$

Solving LP_4^j in this inverse manner yields objective function values ${}^{\tau}z^t$ for all iterations $0 \leq t \leq m$. Then the shortest-path distance to the sink is $c(j,\tau) = {}^{\tau}z^{m-l_j} + c_j$. With those values at hand, we can determine the shortest-path value through node $1 \leq j \leq n$ by $c(\sigma, j) + c(j,\sigma) - c_j = z^{f_j-1} + c_j + {}^{\tau}z^{m-l_j}$. Then,

$$Rem(B) = \{1 \leq j \leq n \mid z^{f_j-1} + c_j + {}^{\tau}z^{m-l_j} \geq B\}.$$

The algorithm sketched above determines the set of nodes that have to be removed from the graph to achieve a state of arc-consistency. Of course, other constraints may also remove nodes. In both cases, we must be able to handle these changes efficiently for the next call to our routine. Without going into implementation details, we note that the removal of nodes does not affect the interval structure of the graph, and that the data structures storing the *max_start* and *min_end* rest cliques as well as the first and last clique belongings can be compressed in linear time to delete any number of nodes from the graph.

We conclude that the members of $Rem(B)$ can be computed and deleted in time $\Theta(n \log n)$ and in amortized linear time for $\Omega(\log n)$ calls of the filtering algorithm.

2.3.3.2 Requiring Nodes

To compute $Req(B)$, we need to identify all nodes that must be an element of any path having a value lower than B. Obviously, only nodes on the shortest path S can have this property. Thus, for every node $j \in S$ we need to find out whether the value of a shortest path P with $j \notin P$ is still lower than B.

Remark 2.4 *Let $1 \leq f_j \leq l_j \leq m$ denote the first and the last clique belongings of j. Furthermore, let P be the shortest path with $j \notin P$. Obviously, it either holds that $I_j \cap I_i = \emptyset\ \forall\ i \in P$, or there exists a node $i \in P$ such that $I_j \cap I_i \neq \emptyset$. In the first case, we know that the value of the shortest path not using the time interval I_j has the value*

$$cost(P) = z^m - c_j.$$

In the second case, after having determined and deleted $Rem(B)$ from the graph, we only have to check if there exists any node $i \in \{1,\ldots,n\}$, $i \neq j$, with $I_j \cap I_i \neq \emptyset$. For when such a node i exists, there also exists a path $\overline{P}$ with $i \in \overline{P}$ and $cost(\overline{P}) < B$ (otherwise i would have been deleted before). Since i and j are overlapping, we also know that $j \notin \overline{P}$. Thus, in $\overline{P}$ we have found a path not covering j with a value lower than B. Therefore, $j \notin Req(B)$. On the other hand, if such a node i does not exists, the second case is obsolete, and we only need to consider the first case.

By making that observation, we can determine $Req(B)$ in amortized linear time: First, for all $j \in P$, we check whether there exists a node in the shrunken graph that overlaps with j. Without specifying the implementation details here, we just note that this can be done in linear time for all nodes j. If no overlapping node exists, we compute $z^m - c_j$ and check whether this value is lower than B. If not, we add j to $Req(B)$, otherwise we do not add it to $Req(B)$.

Now, we have an efficient algorithm at hand to compute $Req(B)$. Obviously, other constraints and branching decisions must be taken into account when our procedure is being called next. Thus, we have to be able to transform our graph in such a way that from now on, every path must visit the new required nodes. At first glance this sounds problematic, as a naive approach would delete all arcs going around the required nodes, but this procedure would cause the resulting graph to not have the co-interval property anymore.

We can force the admissible paths (that is, paths with costs lower than B) to visit the required nodes by making them extremely cheap: Let $Req \subseteq \{1,\ldots,n\}$ the set of (currently) required nodes. Furthermore, let $T \gg 0$ be sufficiently large[3]. Then we set $\hat{c}_j := c_j - T \;\forall\; j \in Req$, and $\hat{c}_j := c_j \;\forall\; j \notin Req$. We use $\hat{c}$ instead of c as our objective and check whether the shortest-path value is lower than $B - |Req| \cdot T$. If not, either two required nodes overlap, or the shortest-path value in the original graph exceeds B. Moreover, by determining $Rem(B - |Req| \cdot T)$, we find all nodes that overlap with some required node plus all nodes that would cause the shortest path in the original graph to exceed the threshold B.

We summarize the results from the previous sections in

Theorem 2.11 *Arc-consistency for a weighted stable set constraint on an interval graph can be achieved in time $\Theta(n \log n)$ or in amortized linear time for $\Omega(\log n)$ incremental calls of the filtering algorithm.*

[3]Assuming that $\min_{i \in V}\{c_i\} < 0$, a valid setting for T is for example $T := n \cdot (1 + \max_{i \in V}\{c_i\} - \min_{i \in V}\{c_i\})$

2.4 Weighted All Different Constraints

The constraint structure of many discrete optimization problems can be modeled efficiently using all-different constraints. As a matter of fact, the all-different constraint was one of the first global constraints that were considered in the literature [179]. Regarding the combination of the all-different constraint and a linear objective, in [34] Caseau and Laburthe introduced the *MinWeightAllDiff* constraint. In first applications [35], it was used for pruning purposes only. In [77, 78], Focacci et al. showed how the constraint (the authors refer to it as the *IlcAllDiffCost constraint*) can also be used for domain filtering by exploiting reduced-cost information.

In this section, we present an arc-consistency algorithm for the minimum weight all-different constraint. It is based on standard operations research algorithms for the computation of minimum weight bipartite matchings and shortest paths with non-negative edge weights. We show that arc-consistency can be achieved in time $O(n(d+m\log m))$, where n denotes the number of variables, m is the cardinality of the union of all variable domains, and d denotes the sum of the cardinalities of the variable domains.

The work presented in this section was published in [187]. It is structured as follows: In Section 2.4.1, we formally define the minimum weight all-different constraint. The arc-consistency algorithm for the constraint is presented in Section 2.4.2.

2.4.1 The Minimum Weight All-Different Constraint

Given a natural number $n \in \mathbb{N}$ and variables $X_1,\dots,X_n$, we denote the domains of the variables by $D_1 := D(X_1),\dots,D_n := D(X_n)$, and let $D := \{x_1,\dots,x_m\} = \bigcup_i D_i$ denote the union of all domains, whereby $m = |D|$. Furthermore, given costs $c_{ij} \geq 0$ for assigning value x_j to variable X_i (whereby c_{ij} may be undefined if $x_j \notin D_i$), we add a variable for the objective $Z = Z(X,c) = \sum_{i,X_i=x_j} c_{ij}$ to be minimized. Note that the non-negativity restriction on c can always be achieved by setting $\hat{c}_{ij} := c_{ij} - \min_{i,j} c_{ij}$, which will change the objective by the constant $n \min_{i,j} c_{ij}$.

In the course of optimization, once we have found a feasible solution with associated objective value B, we are then only searching for improving solutions, thus requiring $Z < B$. Then, we define:

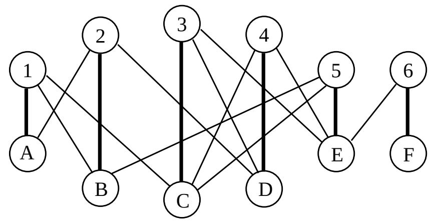
Defi nition 2.10 *The* minimum weight all-different constraint *is the conjunction of an all-different constraint on variables $X_1,\dots,X_n$ and a bound constraint on the objective Z, i.e.:*

$$\text{MinWeightAllDiff}(X_1,\dots,X_n,c,B) := \vartheta_{\text{AllDiff}(X_1,\dots,X_n),Z}[B] = \text{AllDiff}(X_1,\dots,X_n) \wedge (Z < B).$$

Consider the following example: Given variables $X_1,\dots,X_6$ with domains $D_1 = \{\mathcal{A},\mathcal{B},\mathcal{C}\}$, $D_2 = \{\mathcal{A},\mathcal{B},\mathcal{D}\}$, $D_3 = \{\mathcal{C},\mathcal{D},\mathcal{E}\}$, $D_4 = \{\mathcal{C},\mathcal{D},\mathcal{E}\}$, $D_5 = \{\mathcal{B},\mathcal{C},\mathcal{E}\}$, and $D_6 = \{\mathcal{E},\mathcal{F}\}$. In Figure 2.7, we complete the example by specifying a cost matrix c.

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{E}$	$\mathcal{F}$
1	**5**	7	2			
2	2	**3**		4		
3			**4**	8	5	
4			3	**6**	2	
5		8	7		**2**	
6					3	**6**

(a)



(b)

Fig. 2.7: (a) The table gives the costs c_{ij} of assigning a value x_j to a variable X_i. (b) A bipartite graph links variables to values that they can take. Bold numbers and lines mark the optimal solution with objective value 26.

In the following, we will assume $m \geq n$, since otherwise there exists no feasible assignment. Figure 2.7 shows that there is a tight correlation between the minimum weight all-different constraint and the *Weighted Bipartite Perfect Matching Problem* that can be formalized by setting $G := G(X,D,c) := (V_1,V_2,E,c)$ where $V_1 := \{X_1,\ldots,X_n\}$, $V_2 := \{x_1,\ldots,x_m\}$ and $E := \{\{X_i,x_j\} \mid x_j \in D_i\}$. It is easy to see that any perfect matching (that is, a subset of pairwise non-adjacent edges of cardinality n) in G defines a feasible assignment of all-different values to the variables. Therefore, there is also a one-to-one correspondence of cost-optimal variable assignments and minimum weight perfect matchings in G.

For the latter problem, a series of efficient algorithms have been developed. Using the *Hungarian method* or the *successive shortest path algorithm*, it can be solved in time $O(n(d + m\log m))$, where $d := \sum_i |D_i|$ denotes the number of edges in the given bipartite graph. For a detailed presentation of approaches for the Weighted Bipartite Matching Problem, we refer the reader to [1].

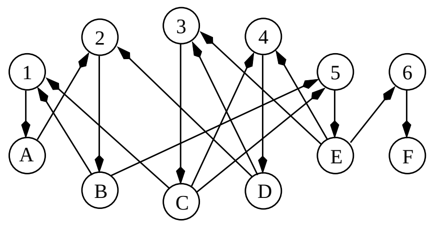
Since there are efficient algorithms available, there is no need to apply a tree search to compute an optimal variable assignment if the minimum weight all-different constraint is the only constraint of a discrete optimization problem. However, the situation changes when the problem consists of more than one minimum weight all-different constraint or a combination with other constraints. Then a tree search may very well be the favorable algorithmic approach to tackle the problem [34].

In such a scenario, we can exploit the algorithms developed in the OR community to compute a bound on the best possible variable assignment that can still be reached in the sub-tree rooted at the current choice point. Also, it has been suggested to use reduced-cost information to perform cost-based filtering at essentially no additional computational cost [78].

In the following, we describe an algorithm that achieves arc-consistency in the same worst

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{E}$	$\mathcal{F}$
1	-5	7	2			
2	2	-3		4		
3			-4	8	5	
4			3	-6	2	
5		8	7		-2	
6					4	-6

(a)



(b)

Fig. 2.8: (a) The new cost matrix c^M, and (b) the network N^M for the optimal matching from Figure 2.7.

case running time as is needed to compute a minimum weight perfect matching when using the Hungarian method or the successive shortest path algorithm.

2.4.2 An Arc-Consistency Algorithm

To achieve arc-consistency of the minimum weight all-different constraint, we need to remove all values from variable domains that cannot be part of any feasible assignment of values to variables with associated costs $Z < B$. That is, in the graph interpretation of the problem, we need to compute and remove the set of edges that cannot be part of any perfect matching with costs less than B.

For any perfect matching M, we set $cost(M) := \sum_{\{X_i,x_j\}\in M} c_{ij}$. Furthermore, we define the *corresponding network* $N^M := (V_1, V_2, A, c^M)$ whereby

$$A := \{(X_i,x_j) \mid \{X_i,x_j\} \in M\} \cup \{(x_j,X_i) \mid \{X_i,x_j\} \notin M\},$$

and $c^M_{ij} := -c_{ij}$ if $\{X_i,x_j\} \in M$, and $c^M_{ij} := c_{ij}$ otherwise. That is, we transform the graph G into a directed network by directing matching edges from V_1 to V_2 and all other edges from V_2 to V_1. Furthermore, the cost of arcs going from V_1 to V_2 is multiplied by -1. Figure 2.8 shows the directed network N^M for our example.

In the following, we will make some key observations that we will use later to develop an efficient arc-consistency algorithm. For a cycle C in N^M, we set $cost(C) := \sum_{e\in C} c^M_e$. Let M denote a perfect matching in G.

Lemma 2.12 *Given an edge $e \notin M$, assume that there exists a minimum-cost cycle C_e in N^M that contains e.*[4]

a) There is a perfect matching M_e in G that contains e, and it holds that $cost(M_e) = cost(M) + cost(C_e)$.

b) The set M is a minimum weight perfect matching in G, iff there is no negative cycle in N^M.

c) If M is of minimum weight, then for every perfect matching M_e that contains e, it holds that $cost(M_e) \geq cost(M) + cost(C_e)$.

Proof:

a) Let C_e^+ and C_e^- denote the edges in E that correspond to arcs in C_e that go from V_2 to V_1, or from V_1 to V_2, respectively. We define $M_e := (M \setminus C_e^-) \cup C_e^+$. Obviously, $e \in M_e$, and since $|C_e^+| = |C_e^-|$, M_e is a perfect matching in G. It holds that:

$$cost(M_e) = cost(M) - cost(C_e^-) + cost(C_e^+) = cost(M) + cost(C_e).$$

b) Follows directly from (a).

c) It is easy to see that the symmetric difference $M \oplus M_e = M \setminus M_e \cup M_e \setminus M$ forms a set of cycles $C_1, \ldots, C_r$ in G that also correspond to cycles in N^M. Moreover, it holds that

$$cost(M_e) = cost(M) - cost(M \setminus M_e) + cost(M_e \setminus M),$$

and thus

$$cost(M_e) = cost(M) + \sum_i cost(C_i).$$

Without loss of generality, we may assume that $e \in C_1$. Then, due to (b) and $cost(C_e) \leq cost(C_1)$, we have that

$$cost(M_e) \geq cost(M) + cost(C_1) \geq cost(M) + cost(C_e).$$

■

Theorem 2.12 *Let M denote a minimum weight perfect matching in G, and $e \in E \setminus M$. There exists a perfect matching M_e with $e \in M_e$ and $cost(M_e) < B$, iff there exists a cycle C_e in N^M that contains e with $cost(C_e) < B - cost(M)$.*

[4]Here and in the following we identify an edge $e \in G$ and its corresponding arc in the directed network N^M.

Proof: Let C_e denote the cycle in N^M with $e \in C_e$ and minimal costs.

$\Rightarrow$ Assume that there is no such cycle. Then either there is no cycle in N^M that contains e, or $cost(C_e) \geq B - cost(M)$. In the first case, there exists no matching M_e that contains e.[5] In the latter case, with Lemma 2.12(c), we have that $cost(M_e) \geq cost(M) + cost(C_e) \geq B$, which is a contradiction.

$\Leftarrow$ We have that $cost(C_e) < B - cost(M)$. With Lemma 2.12(a) this implies that there exists a perfect matching M_e that contains e, and for which it holds that $cost(M_e) = cost(M) + cost(C_e) < B$.

■

With Theorem 2.12, we can now characterize values that have to be removed from variable domains in order to achieve arc-consistency. Given a minimum weight perfect matching M in G, infeasible assignments simply correspond to arcs e in N^M that are not contained in any cycle C_e with $cost(C_e) < B - cost(M)$.

Of course, if $cost(M) \geq B$ we know from Lemma 2.12(b) that the current choice point is inconsistent, and we can backtrack right away. So let us assume that $cost(M) < B$. Then, using empty cycles C_e with $cost(C_e) = 0 < B - cost(M)$, we can show that all edges $e \in M$ are valid assignments. Thus, we only need to consider $e \notin M$. By construction, we know that the corresponding edge in N^M is directed from V_2 to V_1, i.e. $e = (x_j, X_i)$. Denote the shortest-path distance from X_i to x_j in N^M by $dist(X_i, x_j, c^M)$. Then, for the minimum weight cycle C_e with $e \in C_e$, it holds that: $cost(C_e) = c_{ij} + dist(X_i, x_j, c^M)$. Thus, it is sufficient to compute the shortest-path distances from V_1 to V_2 in N^M.

We can ease this work by eliminating negative edge weights in N^M. Consider node potential functions $\pi^1 : V_1 \to \mathbb{Q}$ and $\pi^2 : V_2 \to \mathbb{Q}$. It is a well-known fact that the shortest-path structure of the network remains intact if we change the cost function by setting $\overline{c}^M_{ij} := c^M_{ij} + \pi^1_i - \pi^2_j$ for all $(i, j) \in M$, and $\overline{c}^M_{ij} := c^M_{ij} - \pi^1_i + \pi^2_j$ for all $(i, j) \notin M$. Then,

$$dist(X_i, x_j, c^M) = dist(X_i, x_j, \overline{c}^M_{ij}) - \pi^1_i + \pi^2_j.$$

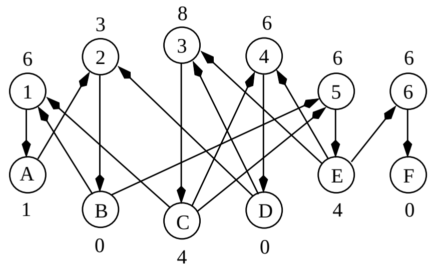
If the network does not contain negative weight cycles (which is true because M is a perfect matching of minimum weight, see Lemma 2.12(b)), we can choose node potentials such that $\overline{c}^M \geq 0$. This idea has been used before in the all-pairs shortest path algorithm by Johnson [43].

In our context, after having computed a minimum weight perfect matching, we get the node potential functions π^1 and π^2 for free by using the dual and negative dual values corresponding to the nodes in V_1 and V_2, respectively. As a matter of fact, the resulting cost vector $\overline{c}^M$ is exactly the vector of reduced costs $\overline{c}$: If $e = (i, j) \in V_1 \times V_2$, then $0 = \overline{c}_{ij} = c_{ij} - \pi^1_i - (-\pi^2_j)$, and thus $\overline{c}^M_{ij} = -c_{ij} + \pi^1_i - \pi^2_j = -\overline{c}_{ij} = 0 = \overline{c}_{ij}$. Otherwise, $\overline{c}^M = c^M_{ij} - \pi^1_i + \pi^2_j = c_{ij} - \pi^1_i - (-\pi^2_j) = \overline{c}_{ij}$ (see Figure 2.9).

[5] Note that this observation is commonly used in domain fi ltering algorithms for the all-different constraint [179].

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{E}$	$\mathcal{F}$
1	0	1	0			
2	0	0		1		
3			0	0	1	
4			1	0	0	
5		**2**	**5**		0	
6					1	0

(a)



(b)

Fig. 2.9: (a) The changed cost vector $\bar{c} = \bar{c}^M$, and (b) the network N^M with node potentials π^1 and π^2. Bold numbers show those assignments that can be eliminated by simple reduced-cost propagation in the presence of a solution with value $B = 28$.

We summarize: To achieve arc-consistency, we first compute a minimum weight perfect matching in a bipartite graph in time $O(n(d + m\log m))$. We obtain an optimal matching M, dual values π^1, π^2, and reduced costs $\bar{c}$. If $cost(M) \geq B$, we can backtrack. Otherwise, we set up a network $N = (V_1, V_2, A, \bar{c})$ and compute n single source shortest paths with non-negative edge weights, each of them requiring time $O(d + m\log m)$ when using Dijkstra's algorithm in combination with Fibonacci heaps [43]. We obtain distances $dist(X_i, x_j, \bar{c})$ for all variables and values. Finally, we remove value x_j from the domain of X_i, iff

$$\begin{aligned}
\bar{c}_{ij} + dist(X_i, x_j, \bar{c}) &= c_{ij} + dist(X_i, x_j, \bar{c}) - \pi^1_i + \pi^2_j \\
&= c_{ij} + dist(X_i, x_j, c^M) \\
&= cost(C_{\{i,j\}}) \\
&\geq B - cost(M),
\end{aligned}$$

where $C_{\{i,j\}}$ is the shortest cycle in N^M that contains $\{i, j\}$. Obviously, this entire procedure runs in time $O(n(d + m\log m))$. The entire domain filtering process is visualized for our example in Figure 2.10.

Interestingly, the idea of using reduced-cost shortest-path distances has been considered before to strengthen reduced-cost propagation [78]. For an experimental evaluation of this idea, we refer the reader to that paper. Now we have shown that this enhanced reduced-cost propagation is powerful enough to guarantee arc-consistency for the minimum weight all-different constraint.

The algorithm we introduced achieves arc-consistency in time $O(n(d + m\log m))$. At first sight this sounds optimal, because it is the same time that is needed by algorithms for the Weighted Bipartite Perfect Matching Problem such as the Hungarian method or the successive shortest path algorithm. However, two questions remain open: First, can we derive a cost-based

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{E}$	$\mathcal{F}$
1	0	0	2	2	2	3
2	1	0	2	2	2	3
3	0	0	0	1	2	3
4	0	0	0	0	2	3
5	0	0	0	0	0	1
6	∞	∞	∞	∞	∞	0

(a)

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{E}$	$\mathcal{F}$
1	-5	-6	0	-4	0	-3
2	-1	-3	3	-1	3	0
3	-7	-8	-4	-7	-2	-5
4	-5	-6	-2	-6	0	-3
5	-5	-6	-2	-6	-2	-5
6	∞	∞	∞	∞	∞	-6

(b)

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{E}$	$\mathcal{F}$
1	0	1	**2**			
2	1	0		**3**		
3			0	1	**3**	
4			1	0	**2**	
5		**2**	**5**		0	
6					∞	0

(c)

Fig. 2.10: (a) Shortest paths from nodes in V_1 to nodes in V_2 with respect to the reduced costs $\bar{c}$. (b) The same shortest paths using the original cost vector c^M. (c) The additional costs imposed by an assignment $X_i = x_j$. Bold numbers show those assignments that can be eliminated in the presence of a solution with value $B = 28$.

filtering algorithm from the cost-scaling algorithm that gives the best known time bound for Assignment Problems that satisfy the similarity assumption? Second, can the above filtering method be implemented to run incrementally faster?

2.5 Knapsack Constraints

Based on reduction techniques for the Knapsack Problem (KP), we develop cost-based filtering routines for knapsack constraints. We present several algorithms using bounds of different quality. The method that we consider the most interesting in theory and practice is based on a bound proposed by Martello and Toth in [147]. 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 compare two of the new algorithms numerically with two other reduction algorithms that have been proposed earlier in the KP literature.

The work presented in this section was published in [61, 64, 65]. It is organized as follows: First, we motivate the development of an efficient cost-based filtering algorithm for knapsack constraints in Section 2.5.1. In Section 2.5.2, we present existing upper bounds and reduction techniques for the KP. Then, in Section 2.5.3, we develop algorithms for the quick propagation of knapsack constraints. An experimental evaluation of these algorithms, as well as a comparison with alternative approaches is presented in Section 2.5.4. Finally, we discuss generalizations for knapsack related problems in Section 2.5.5.

2.5.1 Defi nition and Applications

Cost-based domain filtering algorithms for knapsack optimization constraints are relevant in various application areas. First of all, capacity constraints are the basic building blocks for linear programs. Therefore, knapsack constraints may very well be viewed as a standard modeling element when tackling problems in this large class of problems. Consider the following example:

Automatic Recording The *Automatic Recording Problem (ARP)* consists in finding an optimal selection of items, each of them associated with a weight, an interval, and a profit value, such that

- the total weight of the selection does not exceed a given capacity,
- all intervals associated with all selected items are pairwise non-overlapping, and
- the profit is maximized.

Thus, the problem consists of a knapsack constraint accompanied by an independent set constraint. It models the automatic selection of TV broadcasts for video recording. The independent set constraint ensures that only non-overlapping broadcasts can be recorded, whereas the knapsack constraint models the limited storage capacity of the recording device. The objective is to maximize user's satisfaction. In Chapter 6, we develop an algorithm that solves the ARP by Lagrangian relaxation using the filtering algorithm presented in Section 2.5.3.1.

Quadratic Knapsack Problems Knapsack constraints can also be used profitably when tackling the *Quadratic Knapsack Problem (QKP)*. It calls for maximizing a quadratic boolean objective function subject to a linear capacity constraint. Filtering algorithms for KP are often used to reduce the size of the given QKP [31]. Consider the relax and cut algorithm of Porto, de Moraes, and Lucena [170] as an example. It 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. To solve the Lagrangian dual, a series of KPs has to be solved in every search node. The authors stress that knapsack variable fixing algorithms are vital ingredients of their approach. The algorithms proposed in Section 2.5.3 may help to increase the overall performance.

In our last example, we show that knapsack constraints may also be relevant for the solution of sub-problems when using decomposition techniques on eligible optimization problems:

CP-based Column Generation In Section 3.1, we develop a method called *constraint programming based column generation*. It implies that a constraint satisfaction problem is set up to generate columns in a column generation framework. When applying that approach to appropriate optimization problems, augmented Knapsack Problems emerge as sub-problems. As an example, consider the *Constrained Cutting Stock Problem* that is a Cutting Stock Problem with additional constraints on the cutting patterns. Using a column generation approach, the sub-problem is a Constrained Bounded Knapsack Problem: the length of the rolls determines the capacity for the cutting patterns, and the objective is used to search for columns with negative reduced costs only. 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 sub-problem is to minimize $1-\pi^T X$ (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. The KP objective then is to maximize $\pi^T X$ with an initial lower bound of 1. Additional constraints usually stem from real-world applications and may be non-linear. Some examples for real-world constraints are given in [38].

2.5.1.1 Constrained Knapsack Problems

The examples given above show that knapsack constraints are often accompanied by other constraints when modeling real-life problems. Therefore, we introduce the definition of *Constrained Knapsack Problems (CKPs)* which are Knapsack Problems with additional constraints, whereby objective function and the capacity constraint have to be linear.

Defi nition 2.11 *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* $G := \{x \in \mathbb{B}^n \mid w^T x \leq C\}$.

2. *Let* $k \in \mathbb{N}$*, and* $R := \{r_1, \ldots, r_k \mid r_j : \mathbb{B}^n \to \mathbb{B} \quad \forall\, 1 \leq j \leq k\}$*. 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* $r(x) = 1 \;\forall\; r \in R$*.* $F(R) := \{x \in G \mid x \text{ is feasible}\}$ *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*

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

Note that, for the unconstrained KP, it holds that $F = G$. For such 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 [146, 167]. For these algorithms, it is not straightforward to see how the reduction algorithms we present in the following could be integrated efficiently.

However, algorithms tailored for the special case of pure KP are usually not able to solve general CKPs, because they do not allow to incorporate additional constraints. One reason is that algorithms designed to solve pure KPs make certain assumptions that do not hold for CKPs. For example, it is not clear for the CKP that we can require the profits to be non-negative (as it is the case for KP), because the strategy of omitting items with positive weight and negative profit [149] may not yield feasible solutions at all. Thus, in general a tree search will be necessary to solve CKPs, and cost-based domain filtering algorithms for knapsack constraints may help to improve the performance of such an approach.

For the remainder of this section, with identifiers $\mathbb{B}, C, n, w, p, G, R$, and F we refer to Definition 2.11. We will sometimes need to refer to reduced (C)KPs where an item $i \in \{1, \ldots, n\}$ is either included or excluded in any feasible solution. We refer to those problems with (C)KP$[X_i = 1]$ or (C)KP$[X_i = 0]$.

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 capacity 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 $Z : \mathbb{B}^n \to \mathbb{Q}$ with $Z(X) = Z(X_1, \ldots, X_n) := p^T X$.

Definition 2.12 *Given any lower bound* $B \geq 0$*, we call the maximization constraint* $\vartheta_{\omega,P}[B]$ *a* knapsack constraint.

Items of a 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.
$$Z(x) \leq B \quad \forall\, w^T x \leq C,\ x_i = 1 \tag{2.2}$$

- items i that can be included into the knapsack as they must be part of any improving solution, i.e.
$$Z(x) \leq B \quad \forall\, w^T x \leq C,\ x_i = 0 \tag{2.3}$$

- items that cannot be decided at the moment.

A filtering algorithm that achieves (hyper-)arc-consistency for the knapsack constraint has to include and to remove items that do not fall into the last class. Since showing that either (2.2) or (2.3) holds for an item i (i.e. to check the arc-consistency of $\vartheta_{\omega,P}[B]$) generally requires to solve a KP itself, complete propagation here is an NP-hard task. One way to cope with the situation is to develop pseudo-polynomial filtering algorithms. For example, in [209] a reduction algorithm for subset-sum knapsack constraints is developed that has pseudo-polynomial run-time.

We propose another way by checking if the inequality holds for an upper bound U on CKP$[X_i = b]$, $b = 0$ or $b = 1$, i.e., we check $U(\mathbb{B} \times \cdots \times \{b\} \times \cdots \times \mathbb{B}) \leq B$. Then we write $U(\text{CKP}[X_i = b]) \leq B$.[6]

2.5.2 Knapsack Relaxations

2.5.2.1 Upper Bounds for Knapsack Problems

The effectiveness of a domain filtering algorithm that achieves relaxed consistency is determined by the relaxation quality, i.e. the bounds used for cost-based filtering. Following the presentation given in chapter 2 of [149], 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 often does not yield tight bounds on the objective. However, if the additional constraints satisfy certain properties, they can be incorporated in the objective function of a pure KP using Lagrangian relaxation. For additional linear constraints, there are ways of how this can be done effectively (see [80, 188] and Chapter 3). 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

[6]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 i-th factor.

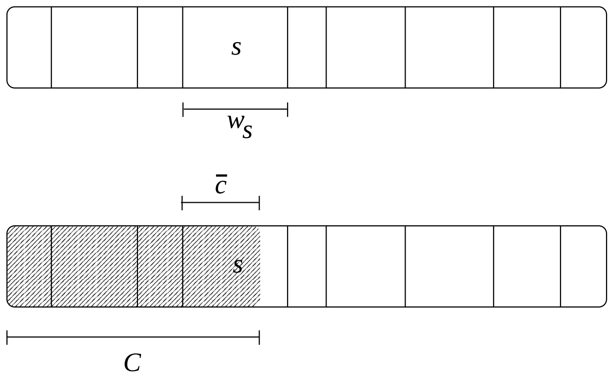


Fig. 2.11: The width of each element is proportional to its weight. The elements are ordered with respect to the effi ciencies p_i/w_i. The leftmost element has the biggest effi ciency, and the rightmost the smallest one. s marks the critical item in U_1.

here where no such s exists). Dantzig [50] 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 $\emptyset \neq M_1, \ldots, M_n \subseteq \mathbb{B}$. Let $l_i := \min(M_i)$ denote the minimum, and $r_i := \max(M_i)$ denote 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

$$U_1(M_1 \times \cdots \times M_n) := \max\{Z(X_1, \ldots, X_n) \mid \omega(X_1, \ldots, X_n) = 1,\ X_i \in [l_i, r_i],\ 1 \leq i \leq n)\}.$$

It holds that,

$$U_1 := U_1(KP) = \sum_{j=1}^{s-1} p_j + \left\lfloor \bar{c}\frac{p_s}{w_s} \right\rfloor. \tag{2.4}$$

A second bound U_2 was introduced Martello and Toth in [147]. It imposes the integrality of the critical item s. Either item s belongs to the optimal solution (leading to a value U^1) or not (leading to a value U^0):

$$U^0 = \sum_{j=1}^{s-1} p_j + \left\lfloor \bar{c}\frac{p_{s+1}}{w_{s+1}} \right\rfloor. \tag{2.5}$$

$$U^1 = \sum_{j=1}^{s-1} p_j + \left\lfloor p_s - (w_s - \bar{c})\frac{p_{s-1}}{w_{s-1}} \right\rfloor. \tag{2.6}$$

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 let s denote 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

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

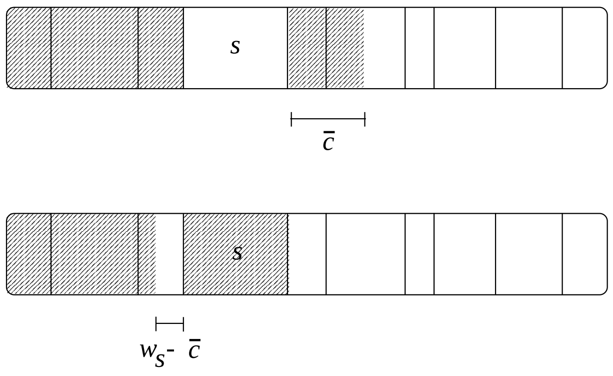


Fig. 2.12: U_3 requires the integrality of item s. The figures show $U_1(\text{KP}[X_s = 0])$, and $U_1(\text{KP}[X_s = 1])$.

It holds that,

$$U_2 := U_2(KP) = \max(U^0, U^1) \leq U_1. \tag{2.7}$$

Instead of estimating the loss caused by the integrality of item s using 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$[X_s = 1]$ [68, 114, 212]. 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.

The Figures 2.11 and 2.12 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 [10] developed an algorithm for the calculation of s using linear time without any preprocessing. However, 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. On top of that, 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 re-computed every time new dual values of the master problem lead to a change of the objective in the successive CKPs.

2.5.2.2 Reduction Techniques for Knapsack Problems

A first reduction algorithm for KPs based on upper bound U_1 has been proposed by Ingargiola and Korsh [122]. 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 filtering can be achieved in the same asymptotic running time. Martello and Toth [148] 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 [53] 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 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 having been developed more than a decade ago, the methods DHR and MTR are still vital ingredients in state-of-the-art solvers for the pure KP and the QKP [167, 168, 170].

The algorithm we present in the following cannot improve 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 require the same asymptotic running time in $\Theta(n \log n)$.

The situation changes, however, if a reduction method is called many times for similar knapsack instances, as it is the case when applying a tree search: in every search node, we try to prune the search or at least to tighten the problem formulation by applying domain filtering. When using unary branching constraints, the subsequent instances only differ with respect to the sets of variables that have already been fixed. As we will see, such a situation allows to hide parts of the work in a preprocessing step that takes time $\Theta(n \log n)$. Provided with the information gathered in that preprocessing, every call to the reduction routine requires linear time only.

2.5.3 Cost-based Filtering for Knapsack Constraints

2.5.3.1 A Fast Propagation Algorithm based on Bound U_1 and U_2

Now, we show how the running time of IKR and MTR can be reduced 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 $\mathrm{KP}[X_j = b]$ denote $\mathbb{B} \times \cdots \times \{b\} \times \cdots \times \mathbb{B}$, $b \in \{0,1\}$, and let $s(M_1 \times \cdots \times M_n) = \min_j \{\sum_{i \leq j \mid M_i = \mathbb{B}} w_i > C - \sum_{i \mid M_i = \{1\}} w_i\}$ denote the critical item of $\mathrm{KP}[X_j = b]$. The key idea of the routine is to calculate the bounds of the reduced problems $U(\mathrm{KP}[X_j = b])$ in an order of increasing weight of the items j. Thereby, we obtain a sequence of critical items that is monotonically increasing. Thus, the critical item and the upper bound for the j-th 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 $s(\mathrm{KP}[X_{j+1} = b])$ at $s(\mathrm{KP}[X_j = b])$.

The time consuming step in reduction algorithms using bound U_1, U_2 is to determine the critical items $s(\mathrm{KP}[X_i = b])\ \forall\ 1 \le i \le n$, and $b \in \{0,1\}$. Once these values are known, the calculation of the upper bounds and the reduction itself only require 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.) [7]

Although calculating $s(\mathrm{KP}[X_i = b])$ for each single $i \in \{1,\dots,n\}$, $b \in \{0,1\}$ generally takes linear time, the calculation of *all* these values also only requires time $\Theta(n)$ once we know an ordering $\sigma = (\sigma_1,\dots,\sigma_n)$ of the items according to their weight, i.e. $w_{\sigma_i} \le w_{\sigma_j}$ iff $i \le j$. 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 $s = s(\mathrm{KP})$, we know that $U(\mathrm{KP}[X_i = 1]) = U(\mathrm{KP})\ \forall\ i < s$, and $U(\mathrm{KP}[X_i = 0]) = U(\mathrm{KP})\ \forall\ i > s$. Thus, we only need to calculate the arrays $S^1 := [s(\mathrm{KP}[X_i = 1]) \mid i \ge s]$, and $S^0 := [s(\mathrm{KP}[X_i = 0]) \mid i \le s]$. 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. That way, we can be sure that $s(\mathrm{KP}[X_i = 0])$ increases monotonically with growing $i \in \{1,\dots,s-1\}$. Thus, we can start the search for the next critical item at the position of the last one.

The following bookkeeping 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 $j \ge s$ 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 being included fractionally.

The first group of items causes at most n € costs as the critical items are monotonically increasing: every item is being passed at most once. It remains to calculate the effort for all items that are being included fractionally. Obviously, there are at most as many fractionally included items as critical items. 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 $s(\mathrm{KP}[X_s = 0])$ can be performed in time that is linear in the number of items as well. Another possibility to calculate this value is to insert item s at the position corresponding to $\bar{c}$ in the weight ordering of items and to calculate $s(\mathrm{KP}[X_s = 0])$ just like the critical items for the exclusion of the other items.

[7]Note that, by omitting the fractional parts, it is also possible to calculate lower bounds for the pure KP. For the general CKP, the necessary feasibility checking with respect to additional constraints makes the generation of lower bounds more complicated. Thus, more elaborate and problem dependent primal heuristics have to developed here. In any case, reduction should only take place, after all lower bounds have been calculated [148].

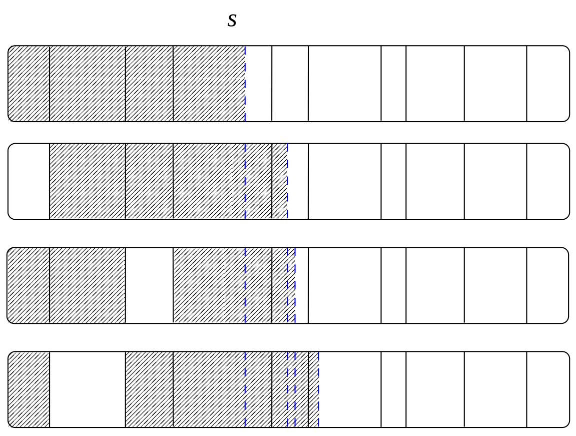


Fig. 2.13: The figure illustrates the process of the reduction algorithm presented for KP$[X_i = 0]$. The weight ordering in which the items are tested ensures that the critical item moves monotonically to the right.

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

Theorem 2.13 *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 the preprocessing is dominated by the accumulated time needed in the choice points. In that case, Theorem 2.13 implies

Corollary 2.5 *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 search tree with $\Omega(\log n)$ search nodes, the method presented here is asymptotically optimal and superior to the algorithms proposed before.

2.5.3.2 More Effective Cost-based Filtering using Bound U_3

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

$U_3(KP)$ is obtained by calculating bound U_1 on KP$[X_s = 0]$, and KP$[X_s = 1]$. When we want to use that bound for cost-based domain filtering, we need to compute s_i^b, $b \in \{0,1\}$, the critical items of those restricted KPs if, additionally, $X_i = b$: Let $1 \leq i \leq n$, $b \in \{0,1\}$. Then $s_i^0 := s(\mathrm{KP}[X_i = b, X_{s(\mathrm{KP}[X_i=b])} = 0])$, and $s_i^1 := s(\mathrm{KP}[X_i = b, X_{s(\mathrm{KP}[X_i=b])} = 1])$.

To compute these values efficiently, first we determine the values $s(\mathrm{KP}[X_i = b])$ using the algorithm in Section 2.5.3.1. Then we apply a binary search to determine s_i^0 and s_i^1 for all $1 \leq i \leq n$. This leads to a running time of $\Theta(n \log n)$. A similar idea has been introduced in [148].

Corollary 2.6 *With the previous procedure, relaxed U_3-consistency for a knapsack constraint can be obtained in time $O(n \log n)$ per choice point.*

For real-life instances, using a binary search to determine the critical item of $\mathrm{KP}[X_s = b_2, X_i = b_1]$ 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 filtering algorithm that is asymptotically as efficient as the one presented in Section 2.5.3.1 (that runs in amortized linear time), but that 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 $1 \leq i \leq n$, $b \in \{0,1\}$, $s := s(KP)$, $s_i^0 := s(\mathrm{KP}[X_i = b, X_s = 0])$, and $s_i^1 := s(\mathrm{KP}[X_i = b, X_s = 1])$. In contrast to the sequence of critical items that is computed for U_3, the second variable X_s that is being fixed remains the same for all s_i^0, and s_i^1. Again, by using the algorithm in Section 2.5.3.1, we determine $U_2(\mathrm{KP}[X_s = 0, X_i = b]) \; \forall \; 1 \leq i \leq n$, and then $U_2(\mathrm{KP}[X_s = 1, X_i = b]) \; \forall \; 1 \leq i \leq n$. For any given $1 \leq i \leq n$, we check whether $\max\{U_2(\mathrm{KP}[X_s = 0, X_i = b]), U_2(\mathrm{KP}[X_s = 1, X_i = b])\} \leq B$. 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 $s(\mathrm{KP}[X_i = 0]) = s$ and items $i > s$ with $s(\mathrm{KP}[X_i = 1]) = s$, however, domain filtering is just 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.

2.5.4 Experiments

After having analyzed the new algorithms theoretically, now we compare them numerically with 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 [121].

2.5.4.1 Test Environment

To show the potential of the new propagation algorithms, and to avoid cross-talking with other constraints, we decided to base the experiments on pure Knapsack Problems only. That way, we get a clear view on the performance of each filtering algorithm without disturbing interferences that can evoke easily when using more complex settings that incorporate additional constraints. For an example of a combination of the algorithms presented here and a shortest path constraint, we refer the reader to Chapter 6. Likewise, we omit specially tailored tree search or branching strategies for pure KPs. Instead, we used the default settings of the underlying CP library.

A word of caution 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 competitive to the best KP solvers (see Section 2.5.1.1). Our focus here is clearly on Constrained Knapsack Problems.

A weak propagation algorithm, if started from scratch, will obviously have to visit more choice points to find an optimal or near-optimal solution of the problem than a good 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 takes to prove optimality.

The generator code of David Pisinger [167] 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 2.5.2.1 are not suited for 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 [146, 150]. In the application area that we focus on (see Section 2.5.1), however, it is justified to assume that the evolving KPs are more likely to fall into one of the classes we used for our tests.

2.5.4.2 The Opponents

The algorithms referred to as linU_1 and linU_2 are based on the amortized linear time reduction method described in Section 2.5.3.1, and use bounds U_1 and U_2, respectively. Methods DHR, and MTR have been described in Section 2.5.2.2. We implemented all algorithms in the same CP environment. Table 2.2 summarizes the major characteristics for the candidates used in the experiments. All methods need $O(n)$ memory for the propagation stack and for the different orderings used. Within a choice point, only $O(1)$ additional memory is required.

Notice that, in our experiments, we do not evaluate the filtering algorithm based on a mixture of bound U_2 and U_3 that was sketched in Section 2.5.3.2. The propagation algorithm based on this

Name	**see**	**Bound**	**pre-proc. time**	**time per node**
DHR	Sect. 2.5.2.2,	$D/H-bound$	–	$\Theta(n)$
MTR	Sect. 2.5.2.2,	U_2	$\Theta(n\log n)$	$\Theta(n\log n)$
linU_1	Sect. 2.5.3.1	U_1	$\Theta(n\log n)$	$\Theta(n)$
linU_2	Sect. 2.5.3.1	U_2	$\Theta(n\log n)$	$\Theta(n)$

Tab. 2.2: Characteristics of the four algorithms used in the experiments.

mixed bound visits only slightly fewer choice points than linU_2, but requires more computation time. Recall from Section 2.5.3 that the work that has to be done to perform domain filtering using bound U_2 is almost the same as using bound U_1. When using the mixed bound, however, the workload is twice as much 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 of choice points that can be achieved by using tighter bounds. Within the test environment that we have chosen for our experiments, a slight reduction of choice points does not justify a much higher effort undertaken in every choice point. Therefore, the filtering algorithm based on the mixed bound is of interest only in the context of more complex CKPs incorporating additional and possibly hard side constraints that would make even small reductions of choice points more favorable. However, in the KP setting that we consider here, to avoid cross-talking with additional constraints and to evaluate the pure performance of the different propagation algorithms, the algorithm developed in Section 2.5.3.2 is not competitive.

2.5.4.3 Numerical 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 2.3, that approach cannot compete at all with the other propagation methods. Both the number of choice points and the CPU time grow exponentially when the problem size increases. A dash means that the average calculation for a test instance takes more than two hours. For both classes, only small problems with not more than 40 items can be solved within that time limit. The poor performance of the pure CP approach shows 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.

Small Instances Tables 2.4 and 2.5 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 that the method visits. Table 2.6 shows a comparison of the different methods regarding the time per choice point for uncorrelated and weakly correlated data.

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	—	—

Tab. 2.3: 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

Tab. 2.4: 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.

Size	**DHR**		**linU_1**		**linU_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

Tab. 2.5: 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.

Size	**Type**	**DHR**	**linU_1**	**linU_2**	**MTR**
n		**time/cp**	**time/cp**	**time/cp**	**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

Tab. 2.6: Uncorrelated and weakly correlated data instances. We give the average time per choice point in milliseconds 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

Tab. 2.7: 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.

The Dembo/Hammer-based filtering algorithm needs to visit the largest amount of choice points among the four propagation algorithms tested. This matches the expected behavior 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 numerical results reflect the expected trade-off between an effective filtering and the time needed to achieve a higher level of consistency. In the presence of additional constraints (causing a longer time spent per choice point that is needed for constraint propagation), it is likely that a smaller number of choice points will result in a faster overall computation. Algorithm linU_1 uses fewer choice points than DHR, but is not as effective as the U_2-based algorithms, MTR and linU_2. For the larger instances of this test set, these two only visit between 50% and 65.6% of the choice points needed by DHR.

For weakly correlated data, linU_2 only visits at most 69.7% of the choice points of the DHR routine. Moreover, linU_2 slightly outperforms DHR with respect to the total running time. Notice that the time per choice point spent by linU_2 for weakly correlated instances is smaller than that for uncorrelated data. The reason for this is that the preprocessing time for initializing the more complex data structures for linU_2 and for sorting the items according to weight and efficiency is spread over a much higher amount of choice points.

Large Instances To get a clearer insight into the characteristics of the different algorithms, we performed some tests on larger instances. Going up to 10 000 items, the disadvantages of the poor bounds used by linU_1 and especially DHR become obvious. Due to a much bigger amount of choice points that have to be visited, the total running times exceed those of linU_2 and MTR (see Table 2.8).

Still, on average, the algorithms based on MTR and linU_2 need about the same running time. We assume that, for smaller test instances, the binary search performed by MTR is faster because it causes less overhead than linU_2. As the problem size increases, however, the difference

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

Tab. 2.8: 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.

	uncorrelated			**weakly correlated**		
Size		**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

Tab. 2.9: 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.

in efficiency becomes more noticeable, and linU_2 slightly outperforms MTR (see Tables 2.7 and 2.9).

A 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 2.8, the total running time for these problem sizes is much higher. Hence, the preprocessing time can be neglected in practice.

2.5.5 Cost-based Filtering for Knapsack Related Problems

Before summarizing our results on the filtering algorithms of knapsack constraints, we would like to discuss their applicability to two special variants of the Knapsack Problem that have been introduced in the literature.

Multidimensional Knapsack Problems The Multidimensional Knapsack Problem consists in 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* Knapsack Problems sharing one objective:

$$\begin{array}{lll} \max & \sum_j p_j X_j & \\ s.t. & \sum_j w_{i,j} X_j \leq C_i, & i = 1, \ldots, m \\ & X_j \in \{0,1\} & \end{array} \tag{2.8}$$

Thus, for each of the capacity constraints, we can define an optimization constraint and perform cost-based filtering using the propagation algorithms we 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 Chapter 3, we develop a generic method for linking filtering algorithms of linear optimization constraints, the *CP-based Lagrangian relaxation*. When applied to Multidimensional Knapsack Problems, problem reduction is based on the filtering routines of the individual knapsack constraints incorporating the other constraints in a Lagrangian objective. We will see that this approach is clearly favorable compared to the loose connection of optimization constraints that interact via domain reduction only.

Note, however, that the asymptotic complexity improvements that we introduced are lost when applying the knapsack filtering algorithm in the context of CP-based Lagrangian relaxation, because for each Lagrangian sub-problem, the objective changes. Thus, the efficiency ordering has to be re-computed which then dominates the algorithmic complexity. It is worth noting that this problem does not occur when the filtering algorithms presented here are applied to column generation sub-problems (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 re-computed only when a new sub-problem is set up.

Bounded Knapsack Problems Bounded Knapsack Problems generalize the 0-1 KP by defining individual bounds on the solution vector:

$$\begin{array}{lll} \max & \sum_j p_j X_j & \\ s.t. & \sum_j w_j X_j & \leq \; C \\ & X_j \in \{0,1,2,\ldots,u_j\} & \end{array} \tag{2.9}$$

The discussion in Section 2.5.1 on the Constrained Cutting Stock Problem has shown an application of Bounded Knapsack Problems. Obviously, (2.9) can be transformed into a CKP by replacing each original variable X_j by u_j new variables $X'_{j,k} \in \{0,1\}, k = 1,\ldots,u_j$. (Note 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. That approach, however, artificially enlarges the number of variables and ignores the additional structure of (2.9) completely.

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 the reader to [149, pp. 84ff.]. Using these extended bounds, efficient propagation for the Bounded Knapsack Problem is then easily achieved by the algorithms proposed in Sections 2.5.3.1 and 2.5.3.2.

2.5.6 Summary

Based on relaxation bounds for KP, we introduced a reduction algorithm that runs in amortized time $\Theta(n)$ for $\Omega(\log n)$ calls. The algorithm can be used efficiently as a propagation routine when solving a combinatorial optimization problem that contains one or more knapsack constraints.

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 are the preprocessing times for initialization and sorting. Also, if more time per choice point is spent by other routines – that, for instance, propagate additional constraints of a CKP or calculate more expensive bounds on the objective – the more important is an effective filtering behavior that justifies a higher effort spent per choice point.

Experiments show that the algorithms presented are as effective as another method based on a reduction technique previously proposed by Martello and Toth for KP. The theoretical analysis and numerical comparison show that the new filtering algorithm is asymptotically more efficient.

Chapter 3

Cost-based Filtering and Problem Decomposition

In the previous chapter, we developed a tool box of efficient cost-based filtering algorithms for a whole variety of important optimization constraints. None of these filtering algorithms is actually useful when the problem corresponding to the constraint has to be solved: For the Shortest Path Problem, the Weighted Stable Set Problem on interval graphs, and the Weighted Bipartite Matching Problem there exist efficient polynomial time algorithms. Therefore, there is no need to apply a tree search and domain filtering to solve these problems. As a matter of fact, the contrary is the case: the filtering algorithms that we developed are based on the efficient algorithms available to solve the corresponding optimization problems. The only NP-hard optimization problem that we considered was the Knapsack Problem. However, even for this problem, it is not clear how the state of the art algorithms for its solution could benefit from the filtering algorithms that we developed.

On the other hand, real-life problems often consist in a combination of various constraints. Frequently, the resulting problem is NP-hard, and the special composition even of well-known optimization problems has not been studied before. Of course, for every such combination at hand, the complexity of the resulting problem could be studied; questions regarding the approximability of the problem may be answered; and algorithms for the predominant constraints may successfully be adapted to efficiently handle the additional constraints. In general, a sound theoretical work will usually establish an understanding of the special augmented constraint structure, and this approach will most likely yield the most efficient algorithms. Therefore, for composed problems that are of great practical relevance or that occur very frequently, this way of constructing an efficient algorithm for a specific problem is favorable and necessary.

In industrial practice, however, the efficiency of the resulting algorithm is not the only criterion. Of course, faster algorithms providing solutions of very good or even provably high quality are clearly favorable. On the other hand, there is an obvious need to develop stable software solutions quickly, and rapid prototyping is of great importance: For example when a company wants to occupy a new market more quickly than its competitors. Or, when the problems that have to be solved are varying, which may be caused by flexible environments in which the types of constraints to be obeyed are changing frequently.

In constraint programming, a problem is represented as a set of constraints on variables with finite domains. The standard algorithmic approach is to apply a tree search where, in every choice point, constraints are *propagated*, i.e., they are used to shrink the domains of the corresponding variables, if possible. This process of constraint propagation is repeated until a stable state is reached where no values can be removed from variable domains anymore. Then a new branching decision is made and the search continues.

This way, constraints interact only via the domains of their variables, which makes the approach extremely flexible with respect to the addition or removal of constraints. Moreover, and in contrast to linear programming, the types of constraints that can be used are not really predefined. All that is needed is a domain filtering or at least a checking algorithm for every constraint that is used in the problem model. These algorithms can be tailored for a given constraint. Standard CP solvers like ILOG SOLVER [121] even offer the possibility to compose constraints out of a set of basic logic and algorithmic constraints, which facilitates the software development process and gives less room for mistakes in the implementation.

In this setting, when given a discrete optimization problem, we may be able to identify substructures that match one of the optimization constraints considered in the previous chapter. Then we can simply plug in the constraint and use its corresponding filtering algorithm. Problem tightening with respect to cost considerations that used to be highly problem-dependent can be handed over to standard libraries that take this task over. This results in a faster and safer software development.

There is a price to pay, however. The way how constraint programming decomposes a problem is very weak, because only one constraint is considered at a time. This allows local inconsistencies to be resolved very efficiently, but on the other hand the approach lacks a global view on a problem, which is particularly bad with respect to the computation of meaningful bounds on the objective.

In the following, we show how to improve upon this situation by making use of two standard decomposition methods in operations research: column generation and Lagrangian relaxation. The results of Section 3.1 were published in [128, 129], and parts of the Sections 3.2, 3.3 were published in [188, 189, 190].

3.1 CP-based Column Generation

Given a natural number $n \in \mathbb{N}$ and a finite set[1] $X \subseteq \mathbb{Q}^n$, we consider the following discrete optimization problem that consists of two constraint families $\mathcal{A}$: $Ax \leq b$, and $\mathcal{B}$: $Bx \leq e, x \in X$: [2]

$$\begin{array}{ll} \text{Minimize} & LP_P = c^T x \\ \text{subject to} & Ax \leq b \\ & Bx \leq e \\ & x \in X. \end{array}$$

The convex hull of solutions to $\mathcal{B}$ defines a compact polytope in $\mathbb{Q}^n$. Let $D = (d^1, \ldots, d^L)$ denote the matrix that consists of one column for each corner of this polytope. Then, each solution of the system $\mathcal{B}$ can be written as convex-combination of the columns of D, i.e., for all $x \in X$ with $Bx \leq e$ there exist $\lambda_1, \ldots, \lambda_L \geq 0$ such that $\sum_i \lambda_i = 1$ and $x = D\lambda$. Therefore, LP_P can be rewritten as

$$\begin{array}{ll} \text{Minimize} & LP_C = c^T D\lambda \\ \text{subject to} & AD\lambda \leq b \\ & \sum_{i \leq L} \lambda_i = 1 \\ & \lambda \geq 0 \\ & D\lambda \in X. \end{array}$$

We achieve a linear continuous relaxation by omitting the discrete constraint $D\lambda \in X$. The advantage of the above re-formulation is, that there is no cross-talking between the constraints in $\mathcal{A}$ and $\mathcal{B}$ anymore. However, in general the matrix AD will contain far too many columns to allow an explicit representation. Fortunately, such a representation is not needed to solve the corresponding LP, because the simplex algorithm considers only one column at a time. Therefore, columns can simply be generated when needed. This idea gave yield to the concept of column generation, and it is one of the most frequently used techniques in the linear programming practice.

The origins of column generation date back to the works of Dantzig and Wolfe [51] and Gilmore and Gomory [96]. The latter paper applies column generation to the classical Cutting Stock Problem where the sub-problem is a Knapsack Problem. More recent applications include specially structured integer programs such as the Generalized Assignment Problem, Time Constrained Vehicle Routing, Crew Pairing, Crew Assignment and related problems. We refer the reader to [56] for a survey.

The procedure works as follows: We start with a sub-matrix $\bar{D} = (d^1, \ldots, d^k)$ and solve the reduced system

$$\begin{array}{ll} \text{Minimize} & LP_R(\bar{D}) = c^T \bar{D}\lambda \\ \text{subject to} & A\bar{D}\lambda \leq b \\ & \sum_{i \leq k} \lambda_i = 1 \\ & \lambda \geq 0 \end{array}$$

[1]A typical example is $X = \{0,1\}^n$.

[2]Here and in the following we identify the name of an LP (here LP_P) and its optimal objective value.

that is called the *master problem*. Denote the dual of the convex combination constraint by $\pi \in \mathbb{Q}$, and let μ denote the vector of duals of the constraints $A\bar{D}\lambda \leq b$. We want to use the dual data to generate a new column that has the potential to reduce the costs in the master problem. In the simplex algorithm, those columns are determined with the help of reduced costs, that must be negative. Thus, we consider the *sub-problem*

$$\begin{array}{ll} \text{Minimize} & LP_S(\mu) = (c^T - \mu^T A)x \\ \text{subject to} & Bx \leq e \\ & x \in X. \end{array}$$

If $LP_S(\mu) < \pi$, we add the solution $d^{k+1} := x$ to the master matrix $\bar{D}$ and start over with the next iteration by re-optimizing the increased master problem $LP_R(d^1, \ldots, d^{k+1})$. Otherwise, the process stops, and we achieve a valid lower bound on LP_P. Since the solution computed will in general not fulfill $D\lambda \in X$, the remaining gap between upper and lower bound has to be closed in a branch & price approach. We refer the reader to Barnhart et al. [13] for further information on this topic.

If a discrete optimization problem can be decomposed in the way we just described, the sub-problem may be viewed as a constraint satisfaction problem where the set X is defined by a set of additional constraints. A typical example of such a sub-problem is the Constrained Knapsack Problem that evolves e.g. when solving the Constrained Cutting Stock Problem with the help of column generation. Another important class of sub-problems are Constrained Shortest Path Problems that evolve in many contexts that range from route guidance [123] and duty scheduling in public transit [25] up to the scheduling of switching engines [142]. The crew scheduling application that we consider in Chapter 5 is another example where the sub-problem exhibits the structure of a Constrained Shortest Path Problem.

For real-life applications, the additional constraints defining X can exhibit very complicated structures, such as gliding time window constraints in the Airline Crew Assignment for example. Also, the additional constraints may vary from case to case. A constraint propagation approach can easily cope with that situation.

In that context, the advantage of the problem decomposition consists in the fact that the constrained sub-problem does not contain the restrictions of $\mathcal{A}$ anymore. Standard CP modeling would also separate the families $\mathcal{A}$ and $\mathcal{B}$. However, when considering $\mathcal{B}$, the constraints in $\mathcal{A}$ are simply ignored, which can have a severely bad impact on the bounds used for cost-based filtering. Using the above decomposition, we also consider the constraint family $\mathcal{B}$ only, but in combination with changing objectives that reflect the constraints in $\mathcal{A}$. Therefore, we achieve a global view on the problem and tighter bounds that are used for a much more effective domain filtering.

Of course, our hope is to find a decomposition such that we can identify a predominant optimization constraint in the sub-problem that can be used for an efficient cost-based filtering in the column generation process. Provided with such a decomposition, the algorithms developed in the previous chapter can help to solve these problems efficiently.

3.2 CP-based Lagrangian Relaxation

Given a natural number n and vectors $l,u \in \mathbb{N}^n$, we consider an integer linear optimization problem (IP) consisting of the two constraint families $\mathcal{A}$: $Ax \leq b$, $x_i \in \{l_i,\ldots,u_i\}$, and $\mathcal{B}$: $Bx \leq d$, $x_i \in \{l_i,\ldots,u_i\}$:

$$\begin{array}{ll} \text{Minimize} & L = c^T x \\ \text{subject to} & Ax \leq b \\ & Bx \leq d \\ & x_i \in \{l_i,\ldots,u_i\}. \end{array}$$

A common way to achieve a lower bound $\bar{L}$ on such a problem is to drop the integrality constraints $x_i \in \{l_i,\ldots,u_i\}$ and to replace them by $l_i \leq x_i \leq u_i$ instead. We get

$$\begin{array}{ll} \text{Minimize} & \bar{L} = c^T x \\ \text{subject to} & Ax \leq b \\ & Bx \leq d \\ & l \leq x \leq u. \end{array}$$

Now, to achieve a state of relaxed $\bar{L}$-consistency we could of course solve a series of LPs $\bar{L}[x_i = v]$ where we set some variable x_i, $1 \leq i \leq n$, to some value $v \in \{l_i,\ldots,u_i\}$. Then, given an upper bound B, we can eliminate v from the domain of x_i if $\bar{L}[x_i = v] \geq B$. Note that, due to $\bar{L}[x_i \leq v] \geq \bar{L}[x_i \leq w]$ for all $w \geq v$ (the lower bound constraints follow analogously), this procedure will not split the domains of the variables x. That is, after the filtering the domains of the variables x_i can again be represented as $x_i \in \{\hat{l}_i,\ldots,\hat{u}_i\}$ for some $\hat{l}_i \geq l_i$ and $\hat{u}_i \leq u_i$, $1 \leq i \leq n$.

The problem with the previous probing procedure is that it requires to re-optimize a dual feasible LP many times, and this is usually unattractive with respect to the required computation time. Therefore, it has been suggested to estimate the loss in performance by carrying out exactly one dual re-optimization step. This method is known as *reduced-cost filtering*. It is computationally cheap, but since it only indirectly exploits the structure of the problem it has a tendency to be rather ineffective.

To improve the inherent trade-off between computational effort and effectivity, we try to decompose the problem. Assume that efficient filtering algorithms Prop($\mathcal{A}$) and Prop($\mathcal{B}$) exist that achieve a state of relaxed consistency for the constraint families $\mathcal{A}$ and $\mathcal{B}$, respectively. The obvious approach to solve problem L exactly is to apply a branch-and-bound algorithm using linear relaxation bounds for pruning and the existing filtering algorithms Prop($\mathcal{A}$) and Prop($\mathcal{B}$) to tighten the problem formulation in every choice point.

However, even though Prop($\mathcal{A}$) and Prop($\mathcal{B}$) may be effective for the substructures they have been designed for, their application for the combined problem is usually not. This is because 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 looking at either one constraint family only.

Lagrangian relaxation allows us to bring together the advantages of a tight global bound and the existing filtering algorithms that exploit the special structure of their respective constraint families. The idea of Lagrangian relaxation was first presented in [59] for Resource Allocation Problems. Held and Karp used it for the TSP [107, 108], and it has been applied in many different areas since then. For a general introduction we refer the reader to [1]. The method that we present in the following is somewhat related to that in [80] where Focacci et al. introduce a method to strengthen cost-based filtering by using Lagrangian multipliers to incorporate additional cuts to tighten the bound used for propagation.

For our abstract composed problem, we introduce a vector of Lagrange multipliers $\lambda \leq 0$ and define the Lagrangian sub-problem

$$\begin{array}{ll} \text{Minimize} & L_{\mathcal{B}}(\lambda) = c^T x - \lambda^T (Ax - b) \\ \text{subject to} & Bx \leq d \\ & x_i \in \{l_i, \ldots, u_i\}. \end{array}$$

For every choice of $\lambda \leq 0$, $L_{\mathcal{B}}(\lambda)$ is a lower bound on L. Then the Lagrange multiplier problem or *Lagrangian dual* consists in finding the maximum lower bound that can be achieved:

$$\begin{array}{ll} \text{Maximize} & G = L_{\mathcal{B}}(\lambda) \\ \text{subject to} & \lambda \leq 0. \end{array}$$

Lemma 3.1 *Given $1 \leq j \leq n$, a value $v \in \{l_j, \ldots, u_j\}$, let $L_{\mathcal{B}}(\lambda)[x_j = v]$ denote the IP that evolves when adding the constraint $x_j = v$ to $L_{\mathcal{B}}(\lambda)$. Furthermore, let $B \in \mathbb{Q}$ denote an upper bound on the objective of L such that $\overline{L}[x_j = v] \geq B$. Finally, denote the continuous relaxation of $L_{\mathcal{B}}(\lambda)$ by $\overline{L}_{\mathcal{B}}(\lambda)$. Then there exists a vector $\lambda \leq 0$ such that $L_{\mathcal{B}}(\lambda)[x_j = v] \geq \overline{L}_{\mathcal{B}}(\lambda)[x_j = v] \geq B$.*

Proof: Let $\lambda \leq 0$ denote a vector of optimal dual values of the constraint family $\mathcal{A}$ in $\overline{L}[x_j = v]$. The theory of Lagrangian relaxation shows that the vector λ defines optimal Lagrange multipliers for $\overline{L}_{\mathcal{B}}(\lambda)[x_j = v]$. Therefore, $L_{\mathcal{B}}(\lambda)[x_j = v] \geq \overline{L}_{\mathcal{B}}(\lambda)[x_j = v] \geq \overline{L}[x_j = v] \geq B$. ■

To put the result into words: Lemma 3.1 shows that for every variable x_j and value $v \in \{l_j \ldots, u_j\}$ that can be filtered with respect to the relaxation $\overline{L}$, there exists a vector of Lagrange multipliers that allows to filter this value with respect to the constraint family $\mathcal{B}$ only. Of course, due to symmetry, the same result holds when we relax $\mathcal{B}$ and keep the constraints in $\mathcal{A}$ as hard constraints only.

This observation motivates the following procedure: We compute G with the help of an iterative algorithm for the maximization of a piece-wise linear, concave function. Standard algorithms used in the literature are subgradient algorithms or bundle methods [1]. For every selection of multipliers $\lambda \leq 0$, $L_{\mathcal{B}}(\lambda)$ is a valid lower bound on L. Thus, we can apply Prop($\mathcal{B}$) on the constraint family $\mathcal{B}$ *every time* when we solve the Lagrangian sub-problem $L_{\mathcal{B}}(\lambda)$. Of course, our hope is that, while solving the Lagrangian dual, we traverse through most relevant selections of Lagrange multipliers, which will result in a filtering that almost achieves a state of relaxed $\overline{L}$-consistency.

If we still find that this filtering procedure does not sufficiently reduce the variables domains, we can do even more. Consider the other possible decomposition

$$\begin{array}{ll} \text{Minimize} & L_{\mathcal{A}}(\pi) = c^T x - \pi^T (Bx - d) \\ \text{subject to} & Ax \leq b \\ & x_i \in \{l_i, \ldots, u_i\}. \end{array}$$

Given a current selection of Lagrangian multipliers $\lambda \leq 0$, denote the optimal dual values of $Bx \leq d$ in the continuous relaxation of $L_{\mathcal{B}}(\lambda)$ by $\pi_\lambda \leq 0$.

Lemma 3.2 *Denote the continuous relaxation of $L_{\mathcal{A}}(\pi_\lambda)$ by $\overline{L}_{\mathcal{A}}(\pi_\lambda)$. Then, $\overline{L}_{\mathcal{A}}(\pi_\lambda) \geq \overline{L}_{\mathcal{B}}(\lambda)$.*

Proof: Denote the dual of $\overline{L}_{\mathcal{A}}(\pi_\lambda)$ by $D_{\mathcal{A}}(\pi_\lambda)$. By assumption, the vector π_λ is dual optimal for $\overline{L}_{\mathcal{B}}(\lambda)$. Let $\mu_\lambda \leq 0$ and $\nu_\lambda \geq 0$ denote the optimal duals for the constraints $x \leq u$ and $x \geq l$ in $\overline{L}_{\mathcal{B}}(\lambda)$, respectively. Then, due to strong LP duality, it holds that

$$\overline{L}_{\mathcal{B}}(\lambda) = d^T \pi_\lambda + \mu_\lambda^T u - \nu_\lambda^T l + \lambda^T b \quad \text{(optimality)},$$

and

$$c - \lambda^T A - \pi_\lambda^T B - \mu_\lambda + \nu_\lambda \geq 0 \quad \text{(feasibility)}.$$

Therefore, λ, μ_λ and ν_λ are feasible solutions to $D_{\mathcal{A}}(\pi_\lambda)$ with the objective value $\overline{L}_{\mathcal{B}}(\lambda)$. Thus, $\overline{L}_{\mathcal{A}}(\pi_\lambda) = D_{\mathcal{A}}(\pi_\lambda) \geq \overline{L}_{\mathcal{B}}(\lambda)$. ■

As a simple consequence, we get the following

Corollary 3.1 *If the Lagrangian relaxation $L_{\mathcal{B}}(\lambda)$ exhibits the integrality property, it holds that: $L_{\mathcal{A}}(\pi_\lambda) \geq L_{\mathcal{B}}(\lambda)$.*

Lemma 3.2 and Corollary 3.1 show that the duals π_λ of $\overline{L}_{\mathcal{B}}(\lambda)$ are a good candidate to achieve an improved lower bound $L_{\mathcal{A}}(\pi_\lambda)$ on L. This observation motivates the idea to improve the effectiveness of the filtering algorithm by applying Prop($\mathcal{A}$) to $L_{\mathcal{A}}(\pi_\lambda)$ in every or at least some iterations of the algorithm that maximizes the Lagrangian dual.

We put the ideas together. Two linear optimization constraint families $\mathcal{A}$ and $\mathcal{B}$ for which efficient filtering algorithms Prop($\mathcal{A}$) and Prop($\mathcal{B}$) are known can be combined effectively: we compute Lagrangian multipliers for $\mathcal{A}$ and use Prop($\mathcal{B}$) for filtering in each Lagrangian sub-problem $L_{\mathcal{B}}(\lambda)$. Then, in selected Lagrangian iterations, we hand back optimal dual information π_λ of $\overline{L}_{\mathcal{B}}(\lambda)$ to propagate $\mathcal{A}$, i.e. we apply Prop($\mathcal{A}$) on $L_{\mathcal{A}}(\pi_\lambda)$.

3.3 Remarks and Generalizations

3.3.1 Solving the Lagrangian Dual and Impotence

When using CP-based Lagrangian relaxation, after having shrunk the domain of the variables, the immediate re-application of the filtering algorithm may yield a further reduction of the domains. This effect is caused by the algorithms — such as subgradient algorithms, bundle methods or the volume algorithm [11] — used for the maximization of the Lagrangian dual, that will in general proceed differently when the domains of the variables are changed. As a result, different Lagrangian multipliers and sub-problems are investigated, which also gives yield to a different filtering behavior. As a consequence, the filtering procedure as described is not idempotent [6].

Moreover, it is not clear whether domain reduction should actually take place during the optimization of the Lagrangian dual. We are save if we just mark those values that can be deleted from variable domains and postpone the actual reduction until the Lagrangian dual is solved. On the other hand, it may be also favorable to incorporate the new knowledge as early as possible. It is subject to further research to investigate how e.g. a subgradient search can cope with changing problems, and whether convergence can still be proven in such a scenario. A practical application of this procedure will be evaluated in Chapter 7.

3.3.2 Redundant Constraint Generation

Since the filtering behavior of the reduction algorithm based on Lagrangian relaxation relies on the sub-problems investigated during the optimization of the Lagrangian dual, we cannot be sure that our cost-based filtering algorithm exhibits a property that we call *continuity*:

Let B denote an upper bound on the minimization problem L, let $\mathcal{C}$ denote the current choice point and $L_{\mathcal{C}}[x=v]$ the best bound achieved regarding the removal of v from the domain of x in $\mathcal{C}$. Now assume that we have $\delta := B - L_{\mathcal{C}}[x=v] > 0$ for some variable x and v in the domain of x. Assume further that a primal heuristic finds a new upper bound $\overline{B} \leq B - \delta$ next. We call a cost-based filtering algorithm *continuous*, if it is guaranteed that in every child node $\mathcal{D}$ of the current choice point $\mathcal{C}$ it is detected that v can be removed from the domain of x.

When using Lagrangian decomposition, this is not the case. Let $\lambda \leq 0$ denote Lagrangian multipliers such that $L_{\mathcal{B}}(\lambda)[x=v] = L_{\mathcal{C}}[x=v]$. Then we cannot be sure that, when performing problem reduction in $\mathcal{D}$, the algorithm optimizing the Lagrangian dual will investigate the Lagrangian multipliers λ. Thus, it may very well be the case that $L_{\mathcal{B}}(\lambda)[x=v] > L_{\mathcal{D}}[x=v]$ and $\overline{B} > L_{\mathcal{D}}[x=v]$.

To overcome this problem, we suggest to store, for each variable-value assignment, the value $L_C[x=v]$ of the largest lower bound achieved so far. This procedure may be viewed as a generation of redundant local constraints of the form: $L > L_C[x=v]$ or $x \neq v$.

3.3.3 Linking more than Two Optimization Constraints

The procedure sketched can easily be generalized if the linking of more than two constraints is desired. All we need to do is to select the substructure that determines the Lagrangian sub-problem, i.e., the one that is used to guide the algorithm for the solution of the Lagrangian dual. In selected iterations, we apply the 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 sub-problem and the Lagrange multipliers for the remaining substructures.

3.3.4 Linear Relaxations and Cuts

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

Of course, the 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 we have to find Lagrange multipliers (or dual values) for.

3.3.5 Binary IPs

Interestingly, as a special case we achieve a propagation algorithm for binary IPs. 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:

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

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:

$$\begin{array}{lrcl} \text{Minimize} & b^T\pi + 1^T\mu & & \\ \text{subject to} & A^T\pi + \mu & \geq & p \\ & \pi, \mu & \geq & 0 \end{array}$$

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

$$\begin{array}{lrcl} \text{Maximize} & (p^T - {}^i\pi^T\,{}^iA)x + {}^i\pi^T\,{}^ib & & \\ \text{subject to} & a_i x & \leq & b_i \\ & x & \in & \{0,1\}^n \end{array}$$

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 one of the knapsack filtering algorithms described in Section 2.5) after we have found optimal dual values of the continuous relaxation.

3.3.6 Column Generation vs. Lagrangian Relaxation

Finally, we compare the two reduction methods developed in the previous two sections. Column generation and Lagrangian relaxation are both absolutely identical with respect to the structure of the sub-problems they investigate. The only difference consists in the way how the sub-problems are achieved. In column generation, the penalties are determined by the duals of a linear program, the master problem. Whereas in Lagrangian relaxation the penalties are updated with respect to subgradients or variants of them. In practice, an average master iteration in column generation is far more costly than in the Lagrangian relaxation setting, but therefore much fewer iterations are necessary to obtain a satisfactory solution value.

The filtering methods we described take this important difference into account: In column generation, we suggest to consider a CP-based tree search to solve the sub-problem, and to use optimization constraints like the ones developed in the previous chapter to ensure the generation of columns with negative reduced costs. The application of a tree search is affordable, because the re-optimization of the master problem is rather costly itself, and because the total number of master iterations is usually low.

On the other hand, we suggest to use Lagrangian relaxation *within* a tree search to obtain lower bounds on the objective. Then, in every Lagrange iteration, we can use optimization constraints for cost-based filtering that determine substructures of the optimization problem at hand.

Chapter 4

Symmetry Breaking

In the previous chapters, we have studied the interplay between the objective functions and the constraints of discrete optimization problems. Now we want to focus on a special aspect of the constraint structure of many constraint satisfaction or optimization problems: Symmetry.

Symmetries can give rise to severe problems for exact and heuristic algorithms as equivalent search regions are unnecessarily being explored more than once. Generally, there are two ways of handling symmetries. The first one is to model the problem in such a way that none or at least less symmetries remain. This may also imply the adding of constraints which will only be satisfied by one assignment in each equivalence class. The major disadvantage of this approach is that it requires the user to have a certain level of experience, and sometimes it is even not possible to remove symmetries from a problem formulation as they are inherent to the given problem. The second way is to break symmetries while searching for a solution. This can be done by adding new constraints on backtracking. Those constraints can be used for domain filtering or, if the detection of symmetries appears to be rather expensive, only for pruning.

The standard approach for breaking symmetries is to model a given discrete optimization or constraint satisfaction problem in some clever and often non-intuitive way. These re-formulations are usually highly problem specific and not generic. In recent years, symmetry breaking was studied more systematically. In [181], Rothberg presents ways to remove symmetries from mixed integer problems by using cuts. Sherali and J.C. Smith discuss the effectiveness of adding constraints to a basic model in a number of case studies [201]. In [93], Gent and B. Smith develop a generic approach called *Symmetry Breaking During Search (SBDS)*. In every choice point, SBDS may extend the model dynamically by adding symmetry breaking constraints. For the Social Golfer Problem, this approach has been shown to be efficient in combination with refined problem formulations which are used to remove some symmetry already in the model [202]. As the number of symmetries in the given problem is enormous, the approach presented is not able to detect all of them and thus also gives non-unique solutions. In [155], Meseguer and Torras introduce a symmetry avoiding approach that works by adapting the search strategy.

We introduce a method that also detects symmetries within the search procedure. Every time the search algorithm generates a new choice point, we check if it is equivalent to or dominated by a node that has been expanded earlier. If so, the current choice point can be pruned. If not, it is processed normally. By checking whether a value assignment to a variable yields a symmetric search node, we can also use symmetries to shrink the domains of variables. However, that propagation can be very costly, and therefore it is not suited in all cases. As the method is based on the detection of dominance relations between sub-trees, we call it *Symmetry Breaking via Dominance Detection (SBDD)*.

The method that we present in the following was also developed independently by Focacci and Milano [81] who presented their work at the same conference. In a later work, the idea of dominance detection between choice points was extended to achieve a method for the heuristic pruning of search nodes when solving discrete optimization problems [82].

The work presented in this chapter was published in [63]. It is structured as follows: In Section 4.1, we formally introduce the SBDD approach. In Sections 4.2, 4.3, and 4.4, it is applied to three different examples from combinatorial optimization and combinatorial design. Numerical results are given that illustrate the effectiveness of the approach.

4.1 Symmetry Breaking by Dominance Detection

The goal of breaking symmetries is to avoid the exploration of a search subspace $\triangle$ that can be mapped into a previously considered part $\square$ via a symmetry function. For if $\square$ does not contain any solution, then neither does $\triangle$. Otherwise, all solutions in $\triangle$ are symmetric to those already computed in $\square$. Thus, symmetries can be used to prune the search tree, and also to remove values from variable domains that would yield the search to a symmetric part of the search space. Before we outline the concept formally, we introduce some helpful definitions first.

Definition 4.1 *Let $\mathcal{X} = \{x_1 \dots x_n\}$ denote the set of variables of the model to solve, and let $D(x)$ denote the domain of a variable $x \in \mathcal{X}$. The tuple $P^c = (D^c(x_1), \dots, D^c(x_n))$ denotes the current state in choice point c. We refer to the representation P^c as a* pattern.

Definition 4.2 *Let $P^c = (D^c(x_1), \dots, D^c(x_n))$, $P^{c'} = (D^{c'}(x_1), \dots, D^{c'}(x_n))$ denote two patterns.*

- *We say that $P^{c'}$* includes *P^c and write $P^c \subseteq P^{c'}$, iff $\forall\, x \in \mathcal{X} : D^c(x) \subseteq D^{c'}(x)$.*
- *We set $\mathcal{MD}^c := D^c(x_1) \times \cdots \times D^c(x_n)$.*
- *Given a* symmetry mapping function *$\varphi : \mathcal{MD}^c \to \mathcal{MD}^c$, we say that $P^{c'}$* dominates *P^c (under the symmetry φ), iff $\varphi(P^c) \subseteq P^{c'}$. Then, we write $P^c \sqsubseteq P^{c'}$.*

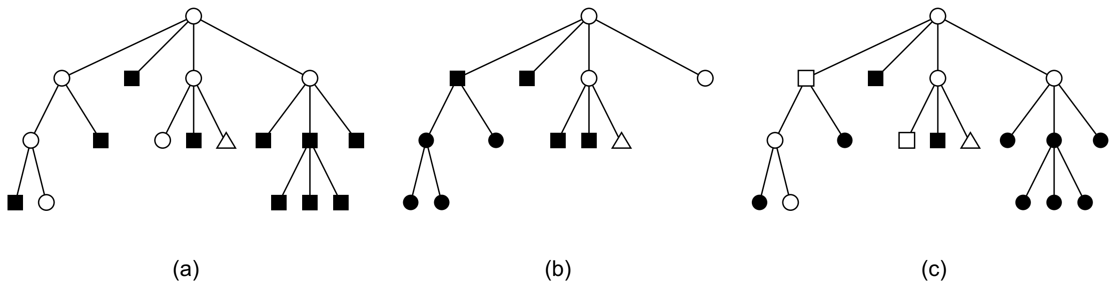


Fig. 4.1: The concept of SBDD.

Due to the monotonicity of filtering algorithms [6], we have the following

Property 4.1 *Given two choice points c and c', where c' is a successor of c in the search tree. Then, it holds that: $P^{c'} \subseteq P^c$.*

The approach that we suggest for pruning symmetric parts of the search space is very simple and straightforward, but to the best of our knowledge apparently it has not been considered before. The method is based on the following ingredients:

- A database $\mathcal{T}$ that stores information on the search space already explored.
- A problem specific function $\Phi : (P^{\triangle}, P^{\square}) \longrightarrow \{false, true\}$ that yields $true$ iff the pattern $P^{\triangle}$ is dominated by $P^{\square}$ under some symmetry function φ.
- If symmetries shall also be used for propagation, a similar function is needed that, for all variables x, removes all values b from the domain of x for which $\Phi(P^{\triangle}[x = b], P^{\square}) = true$.

In every choice point, we check whether the current pattern $P^{\triangle}$ is dominated by some previously considered pattern in $\mathcal{T}$. If so, the current node is pruned. Otherwise, we can use the function Φ for propagation. Thus, we perform Symmetry Breaking via Dominance Detection (SBDD). Figure 4.1 visualizes the general procedure. White nodes are still active, black nodes have been fully expanded already. Boxes represent patterns in $\mathcal{T}$, circles are patterns not or no longer contained in $\mathcal{T}$. Finally, $\triangle$ marks the current node. Originally, a pattern $\triangle$ must be checked against all fully expanded nodes (see Figure 4.1(a)).

Obviously, it is problematic if we are to store all expanded nodes in $\mathcal{T}$. In the next section, we describe how to handle $\mathcal{T}$ efficiently for depth first search (DFS). Later we will generalize the result to arbitrary search strategies.

4.1.1 Efficient Realization in a Depth First Search

The key for an efficient realization of the general SBDD concept as described above is the observation that, within a DFS, we do not need to keep the information of all previously expanded nodes in the search tree. Instead, we can merge sibling entries in $\mathcal{T}$ on backtracking, thus summarizing and compressing the information gathered.

Lemma 4.1 *Let c be a choice point with state $P^c = (D^c(x_1), \ldots, D^c(x_n))$, and denote the states of the children $c_1, \ldots, c_l$ of c by $P^{c_k} = (D^{c_k}(x_1), \ldots, D^{c_k}(x_n))\ \forall\ 1 \le k \le l$. Finally, let $P^{c'}$ denote the state in choice point c' with $P^{c'} \sqsubseteq P^{c_k}$ for some $1 \le k \le l$. Then, it holds that $P^{c'} \sqsubseteq P^c$.*

Proof: Denote the symmetry function by φ with $\varphi(P^{c'}) \subseteq P^{c_k}$. Then, with property 4.1, we have that $\varphi(P^{c'}) \subseteq P^{c_k} \subseteq P^c$. Thus, $P^{c'} \sqsubseteq P^c$. ■

Using Lemma 4.1, SBDD in combination with DFS can be realized efficiently: We start with $\mathcal{T} = \emptyset$ and process each choice point as follows:

---

1. Check the pattern P^c of the current choice point c against all patterns in $\mathcal{T}$. If there exists a pattern $P \in \mathcal{T}$ with $\Phi(P^c, P)$ then fail. (Alternatively encapsulate this function in a constraint and use it also for domain filtering.)
2. Process the current choice point.
3. On backtracking: If there are more sibling nodes to be expanded, then add the current pattern to $\mathcal{T}$, else delete all patterns of the sibling nodes from $\mathcal{T}$.

When using DFS, the current pattern needs only be compared with patterns left-adjacent to the path from the root to $\triangle$ (see Figure 4.1(b)). Notice that step 2 refers to the normal processing of a choice point that also takes place when no additional symmetry breaking framework is utilized, including the choice of a branching constraint and the exploration of the children.

The efficiency of the approach depends on two parameters: the time needed to evaluate the function Φ, and on the number of such evaluations needed. Using the previous procedure, the number of patterns in $\mathcal{T}$ is at most as large as the depth of the search tree times the cardinality of the largest domain.

4.1.2 Arbitrary Search Strategies

With respect to the importance of the size of $\mathcal{T}$, at first it seems to be impractical to combine SBDD with search strategies other than DFS, because the number of previously expanded nodes, and thus the size of $\mathcal{T}$, may be enormous. Another possibility is that the symmetry breaking method becomes ineffective, because many nodes are closed late, which is the case for breadth first search, for instance.

Nevertheless, with a slight modification, it is possible to cope with general search strategies. Let c denote the current choice point, and P^c the corresponding pattern. The idea now is to check whether a symmetry function maps P^c to a pattern of a choice point c' that would have been processed before c if DFS would have been applied on a static ordering of the branching constraints (see Figure 4.1(c)). If so, c is rejected, otherwise we proceed normally. That way, we prune the tree because we detect that the work has either been carried out already or because we decide to do it later. Notice that the current path in the search tree contains all information necessary to identify the patterns that are relevant for checking. The assumption of a static branching constraint ordering defines a virtual ordering of all choice points. The approach rejects the current choice point iff a dominating pattern exists left of it in a virtual DFS tree, i.e. iff the current choice point has a later virtual DFS closing time stamp. As an exhaustive search will eventually consider the leftmost nodes as well, we can be sure not to miss a solution.

Note that the search strategy is slightly affected by this procedure, because the exploration of choice points can be postponed by the symmetry breaking algorithm. This side-effect is clearly not desirable. However, one might expect that a reasonable search strategy rates symmetric parts of the search tree as equally important. In that case, the expanding of the current choice point is only postponed formally, but in fact is carried out next in a symmetric version.

4.1.3 A Different Representation of Choice Points

In Definition 4.1, we have defined a pattern with respect to the current state of the domains. We could also have used the current set of constraints including the branching decisions taken to identify a choice point. When defining symmetry detection functions on pairs of sets of constraints, we can again detect symmetries between choice points. Note that Property 4.1 and Lemma 4.1 are still valid in this setting. Therefore, the idea of SBDD can also be realized efficiently when using a constraint representation of a choice point.

Such a representation may be favorable with respect to non-unary branching constraints, and also with respect to the efficiency of the evaluation of the symmetry detection function [174]. However, functions based on this representation tend to be less intuitive. Therefore, in the examples that we consider in the following, we will use the definition of a pattern as in Definition 4.1 and in combination with unary branching constraints only.

After having outlined the general approach, in the following sections we apply it to three different applications in the field of combinatorial optimization and constraint satisfaction.

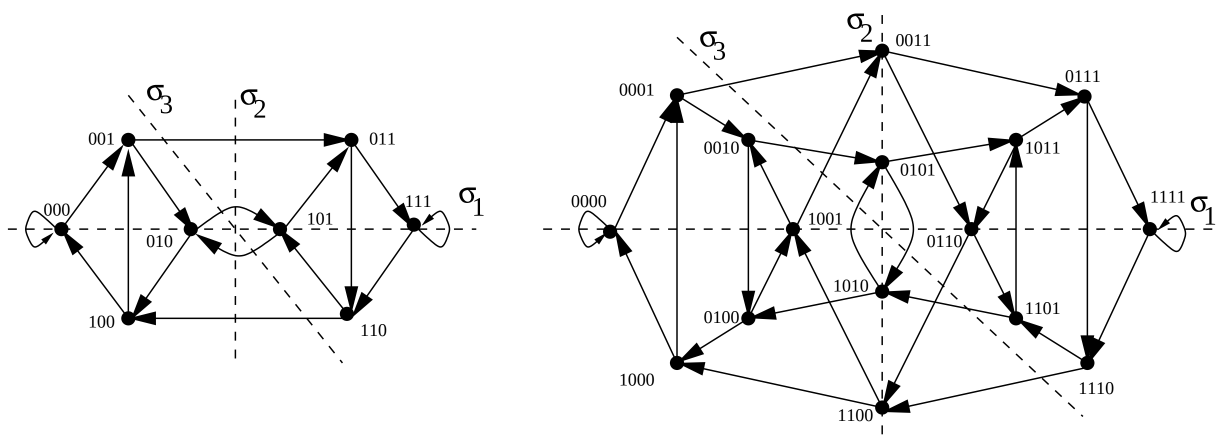


Fig. 4.2: DeBruijn networks of dimension 3 (left) and 4 (right). A node is marked by the binary string corresponding to its number. The dashed lines mark the symmetries of the DeBruijn network.

4.2 DeBruijn Graph Bisection

The first application of the method described in Section 4.1 that we present is the Graph Bisection Problem. Given an undirected graph $G = (V, E)$, the Graph Bisection Problem asks for a set $S \subset V$ such that the cardinalities of S and S^C differ at most by one, and the number of edges between both sets is minimal. This optimal number is often referred to as the *bisection width* of the graph. Graph Bisection is known to be NP-hard, exact solutions can only be computed for small graphs, typically $|V| < 200$. Interestingly, Graph Bisection alone already induces a symmetry as the sets S and S^C can be exchanged.

An obvious symmetry breaking strategy in this case is the initial assignment of a node to the set S. However, if the graph G to be partitioned is itself symmetric, such an assignment does not break the resulting combined symmetries.

In parallel computing, connection networks are typically nicely structured and their symmetries are known. Graphs of the hypercube family have been studied intensively (see [19, 140]). One popular network is the so-called *DeBruijn network* which is defined as follows:

Definition 4.3 *The* DeBruijn Network of dimension k *is a directed graph* $DB(k) = (V_k, E_k)$ *with* $V_k = \{0, \dots, 2^k - 1\}$. *The edge set can be described best by associating the nodes with their corresponding binary representation, i.e.* $V_k = \{(b_0 \dots b_{k-1}) \in \{0,1\}^k\}$. *Then,*

$$E_k = \left\{ (b\alpha, \alpha b),\ (b\alpha, \alpha\overline{b}) \,|\, \alpha \in \{0,1\}^{k-1},\ b \in \{0,1\} \right\}$$

where $\overline{b}$ *denotes inverting bit b, i.e.* $\overline{b} = 1 - b$.

In the following, for the Graph Bisection Problem, we will interpret any directed arc of DB(k) as an undirected edge. Then DB(k) contains 2^k nodes, each having degree 4, and 2^{k+1} edges. Furthermore, DB(k) contains 3 symmetries described by the following automorphisms:

$$\begin{aligned}
\sigma_1 &: V \to V,\ (b_0, b_1, \ldots, b_{k-1}) \mapsto (b_{k-1}, b_{k-2}, \ldots, b_0)\\
\sigma_2 &: V \to V,\ (b_0, b_1, \ldots, b_{k-1}) \mapsto (\overline{b_0}, \overline{b_1}, \ldots, \overline{b_{k-1}})\\
\sigma_3 &: V \to V,\ (b_0, b_1, \ldots, b_{k-1}) \mapsto (\overline{b_{k-1}}, \overline{b_{k-2}}, \ldots, \overline{b_0})
\end{aligned}$$

Symmetries σ_1, σ_2 and σ_3 are visualized in Figure 4.2, where DB(3) and DB(4) are shown.

4.2.1 Bisection Width of the DeBruijn Graph

It can be shown that the bisection width of DB(k) is in $\Theta(\frac{2^k}{k})$, but there are only few results known for specific dimensions. In [70], an optimal bisection width of 30 for DB(7) has been computed. At the time that paper was written, the algorithm based on LP bounds ran for about two weeks. To our knowledge, no exact bisection widths for bigger DeBruijn graphs were known at that time.

In [197], Sensen improved the well-known bound based on clique embeddings by introducing variable multicommodity flows. Using interior point methods for the resulting linear programs, he was able to prove an exact bisection width of 54 for DB(8). SBDD was used to prevent the consideration of symmetric parts of the search space. We refer the reader to [197] for details on the overall approach. Here, we concentrate on the breaking of symmetries. We use this example to show an easy application of SBDD rather than to underline its efficiency. For comparisons with SBDS, we refer the reader to Sections 4.3 and 4.4.

4.2.2 Symmetry Breaking for the Bisection of DeBruijn Graphs

When bisectioning DeBruijn graphs, seven symmetries have to be encoded in Φ. They stem from the three automorphisms of the graph itself, the exchange of S and S^C, and the combination of these symmetries.

For the Graph Bisection Problem, a pattern is implemented as an n-tuple $p \in \{0, 1, *\}^n$. $p_i = 0$ ($p_i = 1$) means that node $i \in S$ ($i \in S^C$). $p_i = *$ means that node i has not been assigned yet. The symmetry functions $\varphi_1, \ldots, \varphi_7$ permute the nodes according to σ_1, σ_2 or σ_3 and/or invert the entries. A pattern $P^{\triangle}$ is dominated by $P^{\square}$ iff there is a symmetry function φ_k, $1 \le k \le 7$ such that, for all $0 \le i < n$, it holds that $\varphi_k(P^{\square})_i = *$ or $P_i^{\triangle} = \varphi_k(P^{\square})_i$.

It is also possible to use pattern information for propagation. Assume that there is a symmetry function φ_k and an index j, $0 \le j < n$, such that $\varphi_k(P^{\square})_i = *$ or $P_i^{\triangle} = \varphi_k(P^{\square})_i\ \forall 1 \le i < n, i \ne j$ and $p_j^{\triangle} = *$. Let $\varphi_k(p^{\square})_j = 0$ (or $\varphi_k(p^{\square})_j = 1$). Then we can enforce that node j is in S^C (or S, respectively).

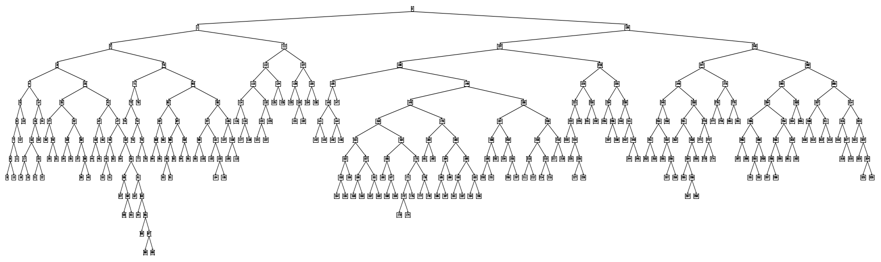


Fig. 4.3: The search tree when bisectioning DB(8) without breaking any symmetries.

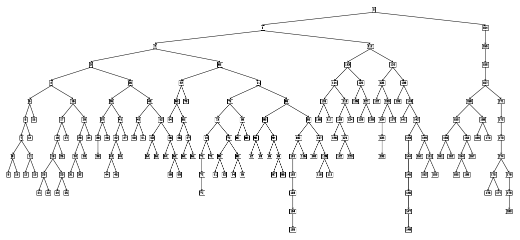


Fig. 4.4: The search tree for the bisection of DB(8) when breaking all possible symmetries. Chains of choice points with only one successor result from symmetry-based domain fi ltering.

Figures 4.3 and 4.4 show the different branching trees resulting from a computation of DB(8) with and without breaking symmetries. Huge parts of the solution space are cut off by lower bound information. Thus, many symmetric sub-trees are pruned early, thereby diminishing the effect of symmetry breaking. However, since the effort per choice point in this approach is very high due to expensive bound computations ($\approx$ 14 minutes per choice point), even small reductions of the tree size improve the overall performance significantly. Thus, for the computation of the bisection width of DB(8), the breaking of symmetries was able to reduce the running time by roughly 2 days, whereby the remaining overall computation time then took 37.5 hours.

In Chapter 9, we consider the Graph Bisection Problem in more detail and develop an approximation scheme for the efficient computation of Sensen's lower bound. The approximation of the bound is fast, but less accurate, which results in far larger search trees. In this environment, when bisectioning DeBruijn graphs, the reduction of choice points is even more visible.

4.3 The Social Golfer Problem

We also applied SBDD to find solutions for the *Social Golfer Problem*. We study that problem in detail in Chapter 8. Therefore, here we introduce it only very briefly. The original question was posed as follows (Problem 10 in CSPLib [47]):

32 golfers want to play in 8 groups of 4 each week, in such way that any two golfers play in the same group at most once. How many weeks can they do this for?

The problem can be generalized by parameterizing it to w weeks and g groups of s players each, written as g-s-w from now on[1]. In case of $(s-1)w = gs-1$, we achieve a specification where every player must play with every other exactly once. This problem is also known as the *Schoolgirl Problem* (see Chapter 8).

4.3.1 Symmetries in the Social Golfer Problem

Obviously, there is a lot of symmetry in the problem. First, players can be placed at any position within a group (φ_P), groups can be exchanged within their week (φ_G), and also the weeks can be ordered arbitrarily (φ_W). Furthermore, the players can be permuted (φ_X).

Following the idea that symmetry detection should also work well in combination with simple models, we have chosen a straightforward one that can be implemented with little effort using the ILOG SOLVER [121] environment. The groups are modeled as sets of players with the cardinality of each set fixed to s. Each week contains g such sets, and the full pattern covers w weeks. To shrink the search space, we fix all players in the first week in increasing order. Additionally, we insert the first s players into the first s groups for all weeks thereafter. Finally, the first group of the second week is filled with the smallest players possible. All these assignments can be made without increasing the complexity of the model nor losing unique solutions.

4.3.2 Symmetry Breaking for the Social Golfer Problem

By using set variables for each group, the model does not contain symmetry φ_P anymore. To detect the domination of patterns with respect to the other symmetries, we describe three symmetry detection functions Φ_G, $\Phi_{W,G}$ and $\Phi_{W,G,X}$, that are used during the search. Function $\Phi_{W,G}$ includes checks performed by Φ_G, and $\Phi_{W,G,X}$ includes those done by $\Phi_{W,G}$.

Φ_G Given two week indices $1 \leq i, j \leq w$, Φ_G is used to check if a week i of pattern $P^{\Box}$ dominates week j of pattern $P^{\triangle}$ with respect to symmetry φ_G. This is done by checking whether all players of week i of pattern $P^{\Box}$ can be mapped to week j of pattern $P^{\triangle}$. In the example shown in Figure 4.5, week 3 of pattern $P^{\Box}$ cannot be mapped to week 2 of pattern $P^{\triangle}$, because players 2 and 3 are in the same group in pattern $P^{\Box}$, but are in different groups in pattern $P^{\triangle}$. Week 1 of pattern $P^{\Box}$ also cannot be mapped to week 2 of pattern $P^{\triangle}$, because player 8 has no matching partner. However, week 2 of pattern $P^{\Box}$ can be mapped to week 2 of pattern $P^{\triangle}$.

[1]In the original problem, it is clear that the golfers cannot play for more than 10 weeks. On the other hand, a solution for 5 weeks can be found easily without backtracking by always choosing the first possible player for a group in each week. Meanwhile, a 9 week solution has been found, but it remains unclear whether there exists a 10 week solution or not.

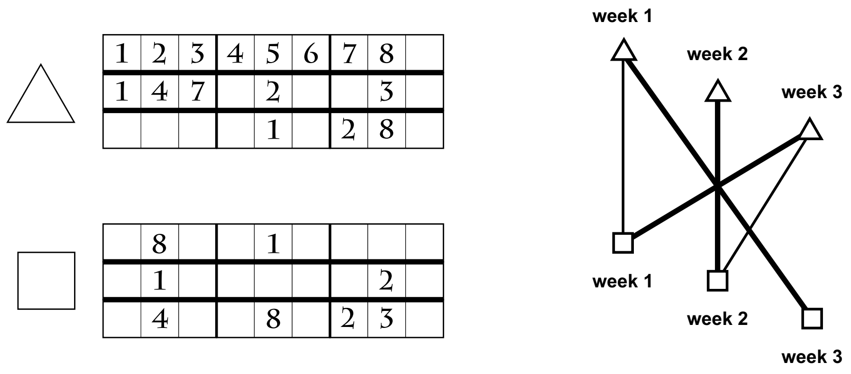


Fig. 4.5: The left hand side shows two patterns $P^{\triangle}$ and $P^{\square}$. Each pattern consists of three weeks (horizontal) of three groups of three players. Unfi xed variables are left empty. On the right hand side, the corresponding bipartite graph is shown, containing a node for each week of both patterns. Since a matching of cardinality 3 exists (bold edges), $P^{\triangle}$ is dominated by $P^{\square}$.

$\Phi_{W,G}$ To break symmetries φ_W and φ_G, function $\Phi_{W,G}$ constructs a bipartite graph G containing a node for each week of $P^{\square}$ and $P^{\triangle}$. An edge is inserted, iff a week of $P^{\square}$ dominates a week of $P^{\triangle}$, which is determined using φ_G. If G contains a matching of cardinality w, i.e., a perfect matching, $P^{\square}$ dominates $P^{\triangle}$. Again, Figure 4.5 shows an example.

$\Phi_{W,G,X}$ Incorporating also the last symmetry φ_X results in a huge computational effort, as $\Phi_{W,G}$ has to be applied for $(g \cdot s)!$ different permutations. To reduce the cost of this check, we use the fact that the first week of a pattern is always complete due to the fixed entries. Since it has to be matched to some other week, "only" $w \cdot (s!)^g \cdot g!$ possibilities are left. However, the test remains expensive. Therefore, we tried some variations reducing the frequency when $\Phi_{W,G,X}$ is applied. A parameter q can be set to restrict full symmetry checks to every q-th level of the search tree. Optionally, it can be limited to be performed on full patterns, i.e. leaves, only, which is the default.

4.3.3 Numerical Results

The model described has been implemented in ILOG SOLVER 5.0 [121] and run for different configurations on a Sun Enterprise 450 (400 MHz UltraSparc-II). Tables 4.1 and 4.2 show the results of the experiments. Apart from the time (in seconds) needed to find the first solution (t_1) and the time to find all solutions (t_{all}), the number of calls to the symmetry detection functions $\Phi_{W,G}$ and $\Phi_{W,G,X}$ is given. In the *sym*-section, $\Phi_{W,G}$ is applied to check for symmetries φ_W and φ_G in each node of the search tree. Since symmetries φ_X are not detected, there are many non-unique solutions found. In the *nosym*-section, $\Phi_{W,G}$ is also applied in every node of the

problem	*solutions*	t_1	t_{all}	$\Phi_{W,G}$	$\Phi_{W,G,X}$	*symmetries*	*cp*	*fails*
				sym				
4-3-2	48	0.00	0.03	226	0	0	195	148
4-3-3	2688	0.02	6.09	99454	0	0	28299	25612
4-3-4	1968	0.05	26.70	382120	0	2808	94845	92878
4-3-5	0	0.00	36.34	412456	0	3120	100389	200390
				nosym				
4-3-2	1	0.00	0.04	226	47	47	195	194
4-3-3	4	0.01	10.00	99454	2687	2684	28299	28296
4-3-4	3	0.04	29.18	382120	1967	4773	94845	94843
4-3-5	0	0.00	36.28	412456	0	3120	100389	200390

Tab. 4.1: Results of the golfer 4-3-X problem.

search tree, and additionally $\Phi_{W,G,X}$ is applied in leaves preventing symmetric solutions from being written out. The tables continue with the number of detected symmetries (*symmetries*), the number of choice points (*cp*), and the number of fails.

Since invoking the symmetry detection function $\Phi_{W,G,X}$ is computationally very expensive, applying it in every search node does not improve the overall run-time, although the number of choice points is reduced. Clearly, there is a trade-off between the reduction of choice points and the effort spent for the detection of symmetries. We have tested a scheme that applies $\Phi_{W,G,X}$ not only in leaves but also performs additional checks for all symmetries in every node in the q-th level of the search tree. Table 4.3 shows that invoking $\Phi_{W,G,X}$ too often rather increases the overall run-time, but applying it too rarely (e.g., only in leaves) is not the best choice either. For the 4-4-4 instance, an invocation in about every 8-th level has shown to be the best. Similar observations have been made for other instances as well. Table 4.4 shows the improved running times for the 4-4-X instance.

4.3.3.1 SBDS versus SBDD

In [202], an SBDS approach is developed for the Social Golfer Problem. To break symmetries, SBDS inserts additional constraints to the model during the search, and hands them over to the solver. Due to the large amount of symmetries in the Social Golfer Problem, the approach presented is not able to add all constraints necessary to break all symmetries.

Therefore, different models for the Social Golfer Problem are discussed. In combination with more complex models that break several symmetries themselves, SBDS performs well and is able to reduce the number of choice points significantly. However, the approach presented in [202] is not able to only compute unique solutions. Moreover, the general approach for the

problem	*solutions*	t_1	t_{all}	$\Phi_{W,G}$	$\Phi_{W,G,X}$	*symmetries*	*cp*	*fails*
				sym				
4-4-2	216	0.00	0.09	735	0	0	555	340
4-4-3	5184	0.01	8.71	74175	0	0	43755	38572
4-4-4	1296	0.01	20.53	140595	0	1296	82635	81340
4-4-5	432	0.01	25.90	132531	0	2160	75723	75292
4-4-6	0	0.00	30.76	114027	0	0	72267	72268
				nosym				
4-4-2	1	0.01	0.17	735	215	215	555	555
4-4-3	2	0.01	136.31	74175	5183	5182	43755	43754
4-4-4	1	0.01	22.09	140595	1295	2591	82635	82634
4-4-5	1	0.02	26.51	132531	431	2591	75723	75723
4-4-6	0	0.00	30.71	114027	0	0	72267	72268

Tab. 4.2: Results of the golfer 4-4-X instance.

level of $\Phi_{W,G,X}$	*solutions*	t_1	t_{all}	$\Phi_{W,G}$	$\Phi_{W,G,X}$	*symmetries*	*cp*	*fails*
				nosym				
1	1	0.01	698.51	0	26	18	82	82
2	1	0.02	271.35	29	27	24	123	123
4	1	0.02	101.26	156	79	79	339	339
8	1	0.01	14.51	5292	1296	1296	4730	4730
leaves	1	0.01	22.09	140595	1295	2591	82635	82634

Tab. 4.3: Results of the golfer 4-4-4 instance performing additional checks for symmetry φ_X in search nodes of every q-th depth.

problem	*solutions*	t_1	t_{all}	$\Phi_{W,G}$	$\Phi_{W,G,X}$	*symmetries*	*cp*	*fails*
				nosym, level of $\Phi_{W,G,X} = 8$				
4-4-2	1	0.00	0.17	735	215	215	555	555
4-4-3	2	0.01	134.10	5283	1298	1297	6492	2891
4-4-4	1	0.01	14.51	5292	1296	1296	4730	4730
4-4-5	1	0.02	15.68	5291	1295	1296	4722	4722
4-4-6	0	0.00	17.16	5290	1295	1295	4714	4715

Tab. 4.4: Improved results of the golfer 4-4-X performing additional checks for symmetry φ_X in search tree nodes of every 8-th depth.

Social Golfer Problem is not able to tackle larger instances like the golfers 5-3-7 efficiently. Only in combination with a model designed for the specific case of the Schoolgirl Problem, a solution is found.

Using SBDD for the Social Golfer Problem, it is possible to find unique solutions only. Additionally, it also works in combination with very simple models. Obviously, the performance of the approach that we presented for the Social Golfer Problem can be further improved by using more sophisticated problem formulations (see Chapter 8). However, here we wanted to demonstrate that SBDD can also be used efficiently by inexperienced users and in combination with simple models. We believe that the symmetry breaking method that we developed is so easy to use because all it requires is the definition of the pattern structure and of the function that checks whether a pattern dominates another or not. Thus, the user can think of symmetries algorithmically rather than in terms of constraints.

4.4 The *n*-Queens Problem

Finally, we consider the classical *n*-Queens Problem. It consists in placing n queens on a $n \times n$ chessboard such that no two queens can capture each other. That is, no two queens are allowed to be placed on the same row, the same column, or the same diagonal.

Nowadays constraint programming approaches are able to find *one* solution for 1 000-Queens in a few seconds. Asking for *all non-symmetric* solutions of *n*-Queens requires more effort. In the following, we describe the SBDS approach of Gent and B. Smith [93] on the *n*-Queens Problem and compare it with SBDD.

4.4.1 Symmetry Breaking for the *n*-Queens Problem

It is easy to see that the *n*-Queens Problem incorporates seven symmetries, namely reflections in the horizontal and vertical axis, reflections in the main diagonals, and rotations through $90°, 180°, 270°$.

We use the following standard model for *n*-Queens:

- Each row $i = 0, \ldots, n-1$ is represented by an integer variable x_i. Assigning $x_i = j$ corresponds to placing a queen in row i and column j.
- Additional integer variables y_i and w_i, $i = 0, \ldots, n-1$, are used to check the diagonals of the chessboard. We post the constraints $y_i = x_i + i$, $w_i = x_i - i$.
- The domains are $x \in \{0, \ldots, n-1\}$, $y \in \{0, \ldots, 2n\}$, $w \in \{-n, \ldots, n\}$.
- AllDiff constraints on x, y, and w ensure that no two queens can capture each other.

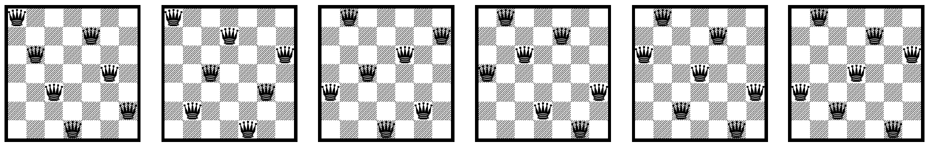


Fig. 4.6: Six out of 40 solutions of 7-queens are unique.

4.4.1.1 SBDS

In [93], SBDS is introduced first and tested on a variety of problems. The approach is general and compatible with different search strategies. A user of the concept only needs to provide symmetry functions mapping a single assignment to its symmetric version.

In a choice point where we set $x = v$ on the left and $x \neq v$ on the right branch, SBDS adds all constraints that are necessary to prevent the solver from exploring a sub-tree symmetric to an already investigated one. By keeping track of all previously broken symmetries, only necessary constraints are posted, thus keeping the overhead small.

4.4.1.2 SBDD

For the n-Queens Problem, a pattern p is an n-tuple where p_i is the column number in which the queen covering row i is placed, or, in case the position of the queen in row i has not been set yet, $p_i = *$. E.g., the pattern corresponding to the first chessboard in Figure 4.6 is $p = (0,4,1,5,2,6,3)$.

4.4.2 Numerical Results

In contrast to the algorithm we developed for the Social Golfer Problem, here we also use symmetry for domain filtering. A constraint is posted to the model that keeps track of the current situation in the search. As propagation turned out to be rather expensive, we limited the number of calls to the propagation routine to one.

We also implemented a version of SBDS and tested it on the model described above. Both codes were running on the same Sun Enterprise as the program for the Social Golfer Problem in Section 4.3.

Table 4.5 compares the number of solutions, the number of fails, and the computation time for calculating *all* solutions (*sym*), calculating only *unique* solutions via SBDS, and *unique* solutions using SBDD. We omit the number of solutions for *SBDD* as it is identical to *SBDS*. The results given for SBDS are similar to those given in [93]. Only the number of fails slightly differs, which we believe to be caused by small variations in the implementation and the different CP engines used (ILOG SOLVER 4.3 vs. ILOG SOLVER 5.0).

	sym			*SBDS*			*SBDD*	
n	*solutions*	*fails*	*time*	*solutions*	*fails*	*time*	*fails*	*time*
4	2	4	0.01	1	3	0.00	6	0.00
5	10	4	0.00	2	4	0.00	13	0.00
6	4	35	0.01	1	11	0.02	31	0.01
7	40	69	0.02	6	19	0.01	56	0.02
8	92	289	0.04	12	63	0.01	130	0.03
9	352	1111	0.16	46	216	0.04	397	0.08
10	724	5072	0.57	92	851	0.13	1464	0.29
11	2680	22124	2.49	341	3808	0.53	5991	1.26
12	14200	103956	11.88	1787	17673	2.52	27731	6.27
13	73712	531401	61.56	9233	89534	12.55	140348	33.11
14	365596	2932626	337.00	45752	483214	69.62	746530	189.07
15	2279184	16920396	1946.07	285053	2784876	403.16	4391877	1213.36
16	14772512	105445065	12154.60	1846955	17277508	2608.51	27153758	7463.62

Tab. 4.5: Solving *n*-Queens without breaking symmetries (*sym*), with breaking symmetries via *SBDS*, and by avoiding them via *SBDD*. Computing times are given in seconds.

Obviously, SBDD does not perform as well as SBDS on the *n*-Queens Problem. The reason for this is that the number of symmetries is fairly small. The difference between SBDS and SBDD can be viewed as follows: SBDS iterates through the symmetries of a problem and adds symmetry breaking constraints if necessary. SBDD on the other hand iterates through the choice points expanded earlier. The latter approach is clearly favorable if the number of symmetries is very high (like for the Social Golfer Problem, for example). However, when the number of symmetries is very limited (as it the case for the *n*-Queens Problem), it is much more efficient to add some few additional symmetry breaking constraints on backtracking.

4.5 Summary

We have suggested an approach for breaking symmetries that is based on the detection of dominance relations between choice points. The method is generally applicable and works in combination with all exhaustive search strategies while it may overrule strategies other than DFS. Moreover, it removes symmetric parts of the search tree efficiently in combination with any model. Thus, it can also be easily used by inexperienced users on straightforward models that do not break symmetries themselves.

The ease of use mainly results from the fact that it is only necessary to define the pattern structure and a function that checks if one pattern dominates another. This algorithmic approach allows somewhat more flexibility than a model that breaks symmetries itself, as has been demonstrated for the Social Golfer Problem when adapting the frequency of constraint propagation for certain symmetries.

The method has shown to be easily applicable without causing a big implementation overhead on three very different applications from combinatorial optimization and constraint satisfaction. Moreover, it worked efficiently even in combination with easy models and also on highly symmetric problems such as the Social Golfer Problem.

As a disadvantage, the use of patterns is less efficient on problems that contain only very few symmetries such as the *n*-Queens Problem. There, the dynamic adding of constraints in an SBDS fashion is clearly favorable.

PART II

—Applications —

In Part I of this thesis, we have introduced general purpose methods for pruning and filtering with respect to cost considerations and symmetry. In the following Part II, we consider some specific combinatorial optimization and constraint satisfaction problems. The applications that we study are used to provide a practical evaluation of the previously developed methods.

In particular, we consider the Airline Crew Assignment Problem in Chapter 5. The approach presented is based on the concept of CP-based column generation in combination with shortest path constraints.

In Chapter 6, we study the Automatic Recording Problem, that evolves in the context of modern multimedia applications. An algorithmic approach is presented that links knapsack constraints and weighted stable set constraints on interval graphs following the idea of CP-based Lagrangian relaxation.

The Capacitated Network Design Problem is tackled in Chapter 7. Lower bounds can be computed by decomposing the problem. We review previously developed reduction techniques and use CP-based Lagrangian relaxation to link them together. Moreover, a new technique is presented that adds locally valid cuts based on Lagrangian relaxation to the problem.

A new approach for the Social Golfer Problem is developed in Chapter 8. Using SBDD for symmetry breaking and the new idea of heuristic constraint propagation, we are able to solve problems that were previously out of reach for solvers based on constraint programming.

Finally, in Chapter 9, we develop a solver for the Graph Bisection Problem. The core of the algorithm is a lower bounding procedure that approximates maximum multicommodity flows.

Chapter 5

Airline Crew Assignment

The Airline Crew Assignment Problem (CAP) consists in assigning lines of work to a set of crew members such that a set of activities is partitioned and the costs for that assignment are minimized. Especially for European airline companies, complex constraints defining the feasibility of a line of work have to be respected. We present two different algorithms to tackle the large-scale optimization problem of Airline Crew Assignment. The first is an application of CP-based column generation that we introduced in Section 3.1. The approach incorporates shortest path sub-problems and uses algorithms from Section 2.2. The second approach performs a CP-based heuristic tree search. We show how both algorithms can be linked to overcome their inherent weaknesses by integrating methods from constraint programming and operations research. Numerical results show the superiority of the hybrid algorithm in comparison to CP-based tree search and column generation alone.

Scheduling flying crews of airline companies is a hard combinatorial problem, given the complexity of the constraints that have to be satisfied and the huge search space that has to be explored. The problem is often tackled by breaking it down into the Crew Pairing and the Crew Assignment (or Rostering) Problem. In the crew pairing part, basic activities such as *flight legs* (flights without stopover) are grouped into *pairings*. The latter ones are lines of work for one or more days starting and ending at a home base. Then, in the crew assignment phase, these pairings are assigned to crew members.

Although easier in practice than the original problem, both sub-problems are still hard to solve. Obviously, the Airline Crew Assignments Problem that we consider here is NP-hard, which is easy to see by reduction to the Set Partitioning Problem [88]. Generally, operations research (OR) and constraint programming (CP) techniques are available to solve the CAP, since it has drawn the interest of both scientific communities for many years until today. Most industrial software is based on OR techniques. However, especially for European airlines, there are strict rules enforced by legislation, unions, etc. that define the feasibility of schedules. Thus, since a huge amount of computational effort is put into the generation of infeasible lines of work, common OR-based generate and test approaches are not efficient enough. We show how constraint programming can be incorporated to overcome typical weaknesses of OR approaches. For a recent overview on optimization problems and solution techniques in the airline industry, we refer the reader to [182, 218].

During the last decade, some work was done on the Crew Assignment Problem. Column generation methods have proven to be quite successful [52, 87, 183]. For solving the Railway Crew Rostering Problem, which is similar, but not identical to the Airline Crew Assignment Problem, Caprara et al. developed both an OR- and a CP-based approach [30, 32]. For the latter, a lower bound from the OR field was used to improve the efficiency.

By construction, OR 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 advanced search techniques and reduction algorithms based on concepts like arc-consistency. 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.

We present two different approaches to tackle the Airline Crew Assignment Problem: a CP-based heuristic tree search approach (HTS) [205], and one following the CP-based column generation framework (CGA) (see Section 3.1). We show how these two approaches can be combined to overcome their inherent limitations.

The work presented in this chapter was published in [62, 194, 195]. It is structured as follows: In Section 5.1, we formally define the Airline Crew Assignment Problem. In Section 5.2, we discuss two autonomous approaches to solve the CAP. We give the characteristics of two real-world airline test cases and present detailed ways of how the two approaches developed can be combined to form an efficient hybrid algorithm in Section 5.3. Finally, in Section 5.4, numerical results show the superiority of the hybrid algorithm compared to the individual approaches.

5.1 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 person. 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 5.1 *Given $k, m, n \in \mathbb{N}$, we denote the* set of crew members *by $C := \{1, \dots, m\}$ and the* set of pairings *by $T := \{1, \dots, n\}$. Furthermore, denote the set of all subsets of a set S by 2^S.*

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 $H := \{h_1, \dots, h_k \mid h_i : R \to B\ \forall 1 \leq i \leq k\}$. 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* (Airline) Crew Assignment Problem (CAP) *is to minimize $\sum_{1 \leq i \leq m} f((c_i, t_i))$, whereby $(c_i, t_i) \in L(H)\ \forall 1 \leq i \leq m$ such that:*
   (a) $\{c_1, \dots, c_m\} = C$
   (b) $\bigcup_{1 \leq i \leq m} t_i = T \quad \text{whereby} \quad t_i \cap t_j \neq \emptyset \Rightarrow i = j \quad \forall 1 \leq i, j \leq m.$

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 for real-life applications. Nevertheless, both methods that we present for solving the previous problem allow to treat linear multiple crew member rules as well.

5.2 Two Approaches for the Crew Assignment Problem

In this section, we introduce two approaches for the CAP that we want to combine later. As a major objective, we aim at developing a generic tool that is able to treat different rules and regulations that typically arise in airline companies. Particularly for European airlines, these rules are very complex and often non-linear. It was therefore decided to model the rules and regulations as a constraint program. Hence, both approaches and the resulting integrated approach are based on a CP core. For further details on this core, we refer the reader to [163].

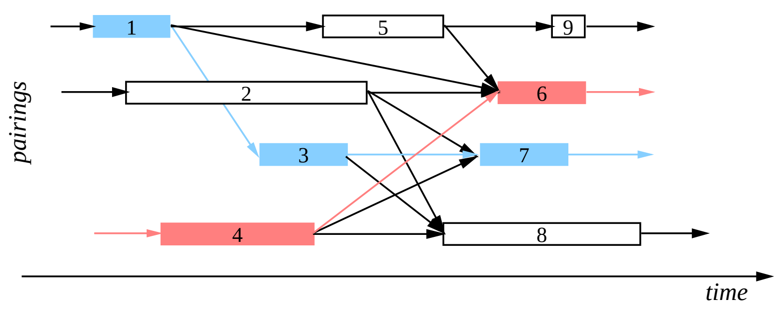


Fig. 5.1: Constructing a legal reduced-cost optimal roster is equivalent to finding a *constrained shortest path* in a weighted DAG.

5.2.1 CP-based Column Generation Approach

The definition of the CAP as stated above allows to decompose it naturally into the *sub-problem* of generating legal rosters and the set partitioning (SPP) *master problem*. Therefore, we can apply the idea of CP-based column generation that was introduced in Section 3.1. The master problem is an integer program (IP) that ensures restrictions (5a) and (5b) in Definition 5.1:

$$\min \sum_{i=1,\dots,k} f((c_{\varphi(i)}, t_i))\, x_i$$

$$\text{s.t.} \sum_{\substack{i=1,\dots,k \\ \text{and } \varphi(i)=j}} x_i = 1 \qquad j = 1,\dots,m \tag{5.1}$$

$$\sum_{\substack{i=1,\dots,k \\ \text{and } s \text{ belongs to } t_i}} x_i = 1 \qquad s = 1,\dots,n \tag{5.2}$$

$$x_i \in \{0,1\}$$

whereby $\varphi : \{1,\dots,k\} \to \{1,\dots,m\}$ maps a column number 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.

The sub-problem consists in finding rosters respecting all rules and improving the objective. From linear programming (LP) duality theory, it is known that columns with negative reduced costs are candidates for such an improvement. Notice that duality theory is only valid for the LP-relaxation of the original IP. Thus, pure column generation must be viewed as a heuristic only. To prove optimality of the IP model, column generation has to be extended to a branch and price approach [13].

We first generate a bunch of individual lines of work and then try to combine them to partition the entire work. When solving the LP-relaxation of the master problem, we get dual information

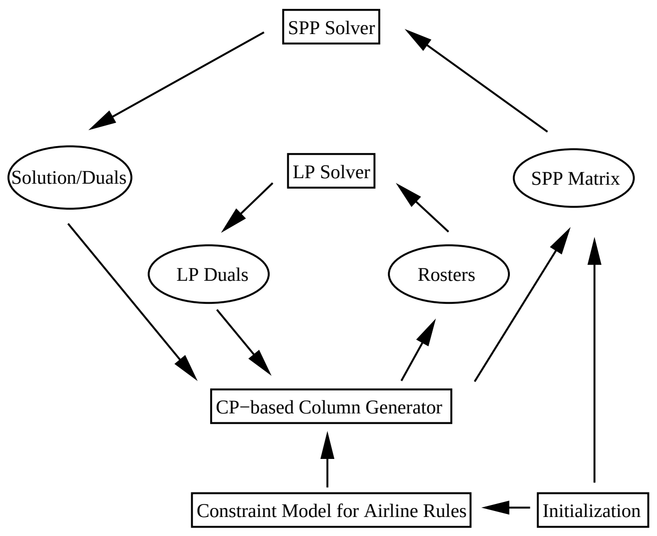


Fig. 5.2: The entire approach: The inner loop generates columns using dual information, the outer loop solves the master problem.

that allows to search for potentially improving columns. That is, in the sub-problem we try to generate new rosters that have negative reduced costs. Those rosters are added to the master problem, which is solved again, and so on until no more rosters with negative reduced costs can be computed or until a certain iteration limit is reached.

Selecting an optimal set of non-overlapping activities respecting the rule set can be interpreted as the problem of finding a constrained shortest path in a weighted directed acyclic graph (DAG) G (see Figure 5.1). However, due to complex and possibly non-linear single crew member rules, generating legal rosters with negative reduced costs can be very difficult, and it is doubtful whether the shortest path substructure is really dominant in the constraint satisfaction problem that arises. Moreover, rule sets vary from airline to airline and have no common structure that could easily be exploited to design a generic efficient constrained shortest path algorithm that can cope with any rule set. Therefore, we apply a CP search to generate legal rosters. As we are only searching for individual lines of work with associated negative reduced costs, we add an optimization constraint that is used for problem reduction. That is, instead of searching for constrained shortest paths, we rather introduce a shortest path constraint.

The entire approach is sketched in Figure 5.2. In the initialization phase we start be setting up the desired airline rules and regulations and generate an initial SPP matrix that may consist of dummy columns only (see Section 5.2.1.2). In the outer loop of *master iterations*, we solve the current SPP integer program and get a first current solution and new dual values of the LP-relaxation. The column generator then defines the next sub-problem to be solved by picking a crew member. A specified number of rosters is generated in the inner loop: we add

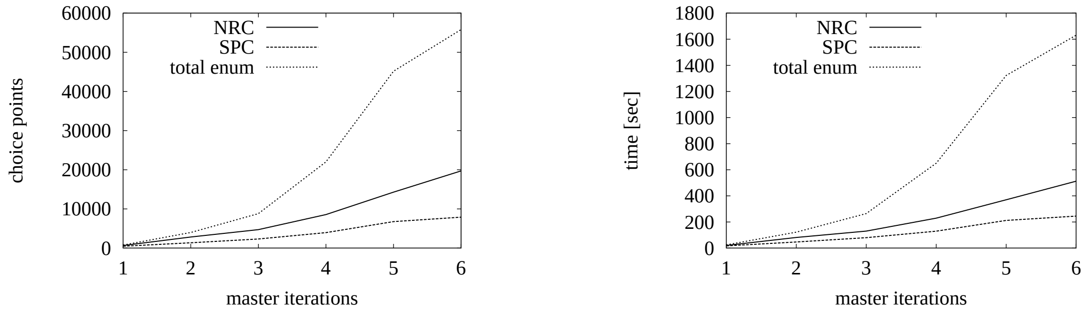


Fig. 5.3: Number of choice points versus master iterations (left), and running time versus master iterations (right) for SPC, NRC, and total enumeration. The tests were run with a data instance of type 10-00-20 that was solved to optimality.

the corresponding columns to the (continuous) master problem, solve it by the means of linear programming, and obtain new dual values. After rosters have been generated for all crew members, we achieve an enhanced SPP matrix and the next master iteration begins. This process is either interrupted after a given time limit has been exceeded or when no more rosters have been generated that yield an improvement of the LP-relaxation in any of the sub-problems.

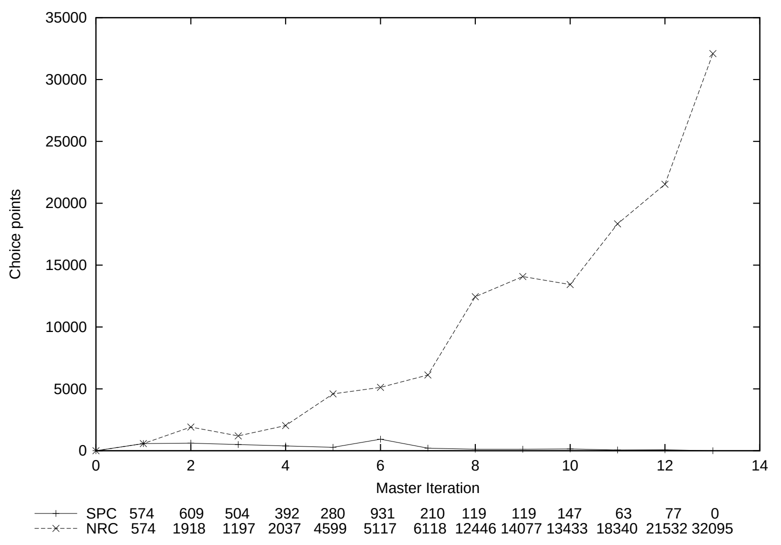
In the following, we investigate some properties of the sketched procedure in more detail and show that CP-based column generation is able to solve non-trivial Crew Assignment Problems. In particular, we demonstrate the effect obtained by the propagation of the path constraint.

The problem instances that we consider here 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 pre-assigned activities, and the number of activities to be assigned. For example, an instance of type 67-165-280 consists of 67 crew members, 165 pre-assignments, and 280 tasks. The experiments were run on a SUN UltraSparc-IV with 296 MHz CPU and 1024 MB main memory. For the constraint model of the CAP, the ROSTER LIBRARY [163] based on ILOG SOLVER 4.4 [120] was used. The LPs and IPs were solved with ILOG PLANNER 3.3 [119].

5.2.1.1 Shortest Path Filtering

In the experiment for Figure 5.3, we compare three models for the generation of legal rosters with negative reduced costs. The first performs total enumeration, the second (NRC) uses a simple arithmetic constraint to ensure the generation of columns with negative reduced costs, and the third (SPC) uses a DAG shortest path constraint (see Section 2.2.2) for this purpose. The



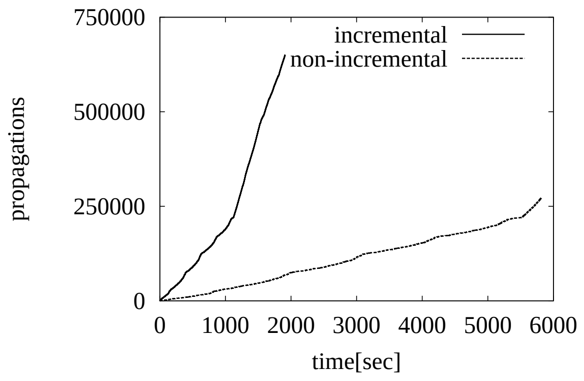
—+—	SPC	574	609	504	392	280	931	210	119	119	147	63	77	0
---×---	NRC	574	1918	1197	2037	4599	5117	6118	12446	14077	13433	18340	21532	32095

Fig. 5.4: Number of choice points versus master iterations using SPC, NRC with a data set of type 7-0-30.

left picture shows the reduction of choice points when using cost-based filtering techniques for problem reduction. In the sixth master iteration, SPC uses less than half the number of choice points than NRC. This gain is not consumed by a significant increase in computation time per choice point: As shown in the right 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 a shortest path constraint when compared with a simple arithmetic constraint in more detail, we run a test on a small instance where, in each master iteration, the number of choice points is noted. Figure 5.4 shows that SPC uses much fewer choice points than NRC. Furthermore, in the last master iteration, the shortest path constraint is able to prove optimality for the continuous relaxation of the master problem very quickly by showing that no more columns with negative reduced costs exist. 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 of choice points and the reduction in time is the use of the incremental version as mentioned in Section 2.2.2.3. In Figure 5.5, we compare a non-incremental version of the shortest path constraint with an incremental one. For a fixed time of 10 000 seconds for the entire optimization, the faster incremental version needs only 2 000 seconds for propagation, whereas, in the non-incremental version, almost 60% of total calculation time is consumed by 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 the solution quality.



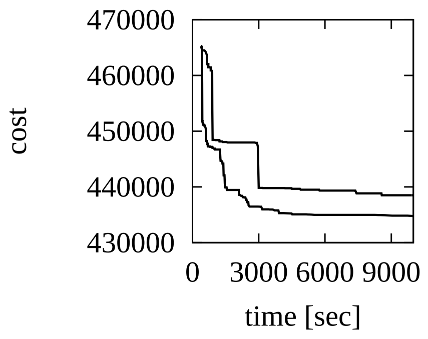


Fig. 5.5: The left picture shows time versus the number of calls of the propagation routine using the incremental and the non-incremental implementation of the shortest path constraint. Both versions were stopped after 10 000 seconds total CPU time. The experiment was run with a data instance of type 10-00-70. The right picture shows a comparison of NRC (upper curve) and SPC (lower curve) in a time versus quality diagram on a data instance of type 67-165-280.

The right picture in Figure 5.5 shows a time versus quality comparison of NRC and SPC. After a first big drop in the objective, NRC dives deeply into huge search trees that only consist of rosters with non-negative reduced costs. SPC can prune those search trees much earlier and therefore continuously reduces the objective without stalling.

We have shown that CP-based column generation works reasonably for the CAP. However, we observe two major drawbacks. We will see later in Section 5.3, how these problems can be overcome by combining the column generation approach (CGA) with a direct CP approach.

5.2.1.2 Set Partitioning — Set Covering

The major obstacle for CGA is the set partitioning (SPP) structure of the master problem. Finding a feasible solution to the SPP is NP-hard already [88]. Moreover, the dual information gained from equation constraints is more difficult to exploit than that of cover or packing constraints. Therefore, we would actually 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 suggest to relax (5.2) to $\sum_i x_i \geq 1$. Then, however, to compute a legal schedule, we have to decide which crew member finally gets an over-covered pairing assigned.

5.2.1.3 Feasible Solutions for Set Partitioning

To obtain a formulation that guarantees that we can always find a feasible solution, we add two types of *dummy columns*: The first type of column covers exactly crew member i, the second exactly one activity j, for all $i = 1, \dots, m$ and $j = 1, \dots, n$. That is, we allow empty rosters and unassigned activities. 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 is infeasible.

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

5.2.2 Heuristic Tree Search Approach

The other algorithm developed to tackle the CAP is the heuristic tree search approach (HTS) based on constraint programming. In that HTS, each complete feasible solution of the CAP is constructed by solving the corresponding constraint satisfaction problem [205]. 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 for this paring. 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 pre-assigned activities, or due to regulation violations because of the crew member's history, etc. The search tree of the problem is created by iterating over pairings in a heuristic dynamical order and assigning each pairing to a crew member.

As usual, every branch in the search tree corresponds to assigning a pairing to a crew member. Every non-leaf node corresponds to a partial assignment, identified by the path from the root to the node. Leaf nodes correspond to infeasible partial assignments or complete and legal schedules, i.e. (not necessarily optimal) feasible solutions of the problem. Each allocation of a crew member to a pairing activates the constraint propagation mechanism. Total enumeration is tried to be avoided by removing values which are inconsistent with the posted constraints from variables' domains. For example, the assignment of a pairing to a crew member causes the removal of this crew member from all pairings' domains that overlap in time with the one that has just been assigned. When a node is proven to be a dead-end, which means that one or more pairings cannot be carried out be any crew member in the given partial assignment, 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 ones are in violation of the airline rules. A solution to a con-

[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 one single crew member.

straint 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 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. Regarding the way the search tree is traversed, a variety of search methods were developed and tested.

5.2.2.1 Tree Traversal

A variety of search methods for traversing the search tree exists in the literature. The oldest, most popular and, by far, most widely used search method is Depth First Search (DFS). The main drawback of DFS is that, even for instances of moderate size, it only explores a very small portion of the search tree at the lower left.[2] DFS was implemented and tested for the CAP, and, as no surprise, it was found that it does not perform very well, because the first decisions taken are never reconsidered.

Innovation in the field came from the notion of *discrepancy*. At a given node, a heuristic function suggests which branch the search should follow, as the one that is assumed is most likely to contain solutions (or solutions of good quality in the case of optimization). Always following the heuristic's advice defines a unique path that is said to contain no discrepancies. Following the heuristic's advice except for one case defines paths of discrepancy 1, for two cases discrepancy 2, and so on.

Limited Discrepancy Search (LDS) [106] is an iterative search method. In the *i*-th iteration, it explores all paths with *i* discrepancies. In the original LDS method, paths with discrepancies higher up the tree are explored before the ones where the discrepancies occur further to the bottom. The intuitive justification for that approach is that a heuristic is more likely to fail higher up the tree, where information is limited. We implemented a variant of LDS. Our variant searches paths with discrepancies lower down the tree before the ones with "higher" discrepancies. The 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 is much bigger than a modern computer could explore in a reasonable amount of time. We refer to this variant as *modified Exact Discrepancy Search* (mEDS).

Depth-Bounded Discrepancy Search (DDS) [213] is also an iterative method. In the *i*-th iteration, it explores all paths where discrepancies occur before depth *i*. In contrast to LDS, a path with many discrepancies high in the tree is explored before a path with very few discrepancies low in the tree. This is also justified by the assumption that heuristics tend to fail with a higher

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

probability on top of the tree.

Finally, we also implemented Large Neighborhood Search (LNS) that was introduced in [200] and incorporates local search techniques within the CP framework. The idea is to restrict the search within a fragment of the search space. In this way, local improvements can be made that would remain unnoticed by most incomplete search methods. A reduced search space for a problem with a set of 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_1 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 sub-space has been exhausted or because any other termination criterion is met), another sub-space is selected and the process is repeated. The advantage of LNS is that local improvements are discovered easily, and the objective value may improve quickly. The disadvantage is that the search space cannot be viewed globally. Thus, it is likely that important improvements are missed. A reasonable 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.

5.3 Integration

We present two ways of integrating both methods each one motivated by one of the two following problem cases:

5.3.1 The Airline Test Cases

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

The objective of company B is to achieve a fair distribution of activities over all crew members, whereas in A 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 A, 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 A are hard constraints, meaning that if they are violated, the solution is considered infeasible. In B, 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 caused by the fact that company B does not insist on a partitioning of the work, i.e., restriction (5b) is relaxed to $\bigcup_{1\leq i\leq m} t_i \subseteq T$. Obviously, this difference requires that our column generation approach is able to incorporate two different types of master problems.

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 over-covered (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.

5.3.2 Transforming a Set Covering into a Set Partitioning Solution

The first method is applied on case A. 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 and the number of pairings that a single crew member is able to service (for example 959 daily pairings and 65 crew members on a typical monthly instance). 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 can much more easily be combined to SCP solutions. Then the conversion of SCP to SPP solutions is performed by the HTS approach, which can resolve local conflicts efficiently by using sophisticated propagation algorithms. An outline of the procedure is shown in Algorithm 1. 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 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 that forms search sub-spaces by dividing the planning horizon into time windows.

Algorithm 1 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
```

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 over-covered pairings. An 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 extensible to a feasible solution.

There are other scheduling problems, such as the Vehicle Routing Problem With Time Windows [185] for example, for which an set covering solution can be repaired easily by removing entries for over-covered rows from all but one of the corresponding columns. However, in our case, this procedure is likely to fail 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 sub-roster 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 over-covered 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.

Algorithm 2 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$(S, v)$
```

The function PARTITION partitions V in exactly this manner. Assignments of variables in V_3 are known to be extensible to a full solution, since one has already been found. Thus, since there is no information which suggests the contrary, they are realized as soon as possible in each iteration (Lines 5 and 6 in Algorithm 1). Assignments in set V_2 may be considered as almost certain. However, in Line 7 of Algorithm 1, they are realized in a way that allows to reconsider them in case there exists no feasible solution that extends the assignments of variables in V_2 and V_3. Finally, CGA does not provide meaningful information for variables in V_1. Therefore, 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 2. There, the variables that are not fixed yet are given in the set 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 1, 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 will most probably lead to an area that contains 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. Two important details are worth to note:

1. The variable selection heuristic is consulted *every time* when 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. That way, *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 motivated by 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 up in the search tree, early decisions are reconsidered first. In our case, though, the assignments for variables in V_2 are realized in the beginning, and we want to stick to them. Therefore, we prefer to use mEDS in this phase, too.

We further note that the function HTSOPTIMIZE in Line 1 of Algorithm 1 may or may not use LNS. Whether LNS can help to improve the efficiency is problem dependent. Our experiments show that, as a stand-alone method, it is not preferable because it is likely to get stuck in a local optimum soon. However, we found that it can be useful to apply LNS after having found the first solution with the help of a global tree search method. Locality is not a major problem when using LNS in combination with CP-based column generation: the latter carries the major burden of optimization, whereas the CP-approach is used to resolve minor local inconsistencies, hopefully without loosing much of the relaxed set covering solution quality. We show the effects of using LNS in our experimental results.

We also use LNS in Line 8 of Algorithm 1 to overcome a problem that might arise when fixing variables in V_3. Recall that they are determined by 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. We give the search more freedom by allowing that these assignments may be reconsidered and use LNS on $V_1 \cup V_2 \cup V_3$ instead of only $V_1 \cup V_2$. To be more precise, in our experiments, we used LNS with mEDS as the sub-tree search method.

5.3.3 Generating Combinable Columns and Exploiting Dual Values

We propose a second integration strategy, that is applied on company B. In this case, the convergence of the CGA approach towards an optimal solution is assisted by HTS first by constructing a set of initial columns that are combinable to complete partitioning solutions in a start-up phase, and second by constructing columns with negative reduced costs during the main optimization phase. These columns are guaranteed to be extensible to a feasible solution, since they are extracted from one. A top level sketch of this method is shown in Algorithm 3. $\mathcal{C}$ is a set of rosters, $\mathcal{A}$ is an assignment, and *duals* are the dual values corresponding to this assignment (obtained by the CGA). The function HTSPOSTNRC transfers the dual values to HTS that is forced to search for columns with negative reduced costs.

5.3.3.1 Start-up 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-

Algorithm 3 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
```

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 be combined legally 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.

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 can be combined feasibly 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 3 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 account is the diversity of the columns that are 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. The idea is realized in the slightly modified value selection heuristic DiverseSVal, which is shown in Algorithm 4. It works exactly as the value selection heuristic that is normally used, but it also records the assignments made and limits the number of times a crew member can be assigned to a pairing. This heuristic, for example in combination with depth-bounded discrepancy search [213], guarantees that columns will be adequately different from each other to make the CGA method even more efficient.

Algorithm 4 Modified value selection heuristic for the second method

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

Especially for large data sets, we find that 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 in locally bounded areas of the search space.

5.3.3.2 Main Optimization Loop

As shown in Line 3 of Algorithm 3, 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 crew members 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 ensures 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 show that schedules that contain 30% columns with associated negative reduced costs can be achieved for our test set. Of course, this does not imply that 70% of the columns produced are useless. Instead, those columns guarantee that all newly generated columns can be extended to a feasible solution. Thus, all columns that are 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 continuous objective.

Line 5 of Algorithm 3 performs an LNS search with few deviations regarding the solution provided by CGA. The pairing with the maximum dual is assigned to the crew with the maximum dual as long as this crew member's reduced costs are not guaranteed to be negative already. Again, our search method of choice is mEDS.

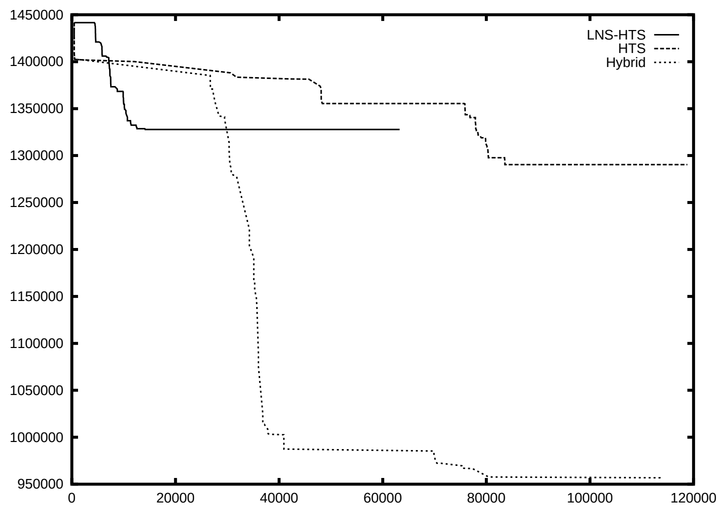


Fig. 5.6: Data set with 65 crew members and 959 pairings.

5.4 Numerical Results

To demonstrate the superiority of combined approaches integrating CP and OR techniques, we applied the hybrid algorithms as presented to real-world Crew Assignment Problems (see Section 5.3.1). 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 SOLVER [120] and ILOG CPLEX [116]. The first integration strategy was applied on two monthly data sets from company A. Experiments for this case were performed on a 640 MB, 296 MHz SUN UltraSparc-II, with a time limit of 120 000 seconds.[3]

The efficiency of our algorithm improves the production system which company A used at the time when this work was done. Figure 5.6 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 stated as 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 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 is 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 1 in order to show that the hybrid has the best performance regardless of the start-

[3]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.

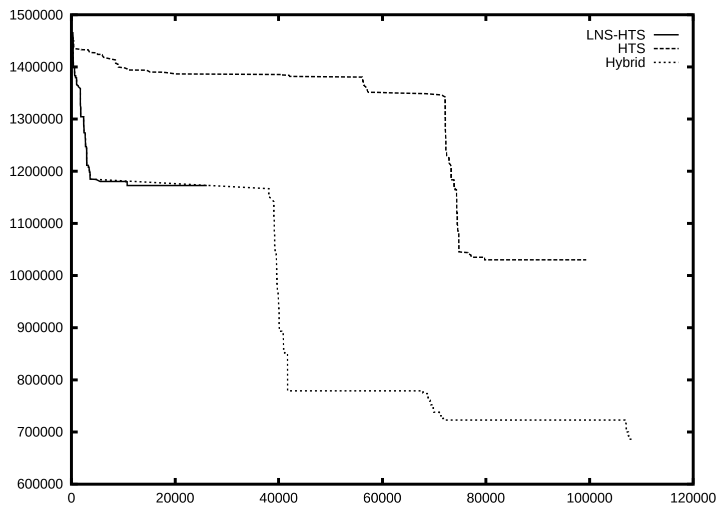


Fig. 5.7: Data set with 50 crew members and 766 pairings.

up phase. That is the reason why “LNS-HTS” outperforms “hybrid” in the beginning. Of course, we repeat that a reasonable choice for the start-up phase of Algorithm 1 would be a strategy more like “LNS-HTS”. This strategy is used in the experiments of Figure 5.7, which shows the performance of the same methods on another monthly data set of company A containing 766 pairings and 50 crew members.

The following set of experiments is carried out in order to investigate the second way of integration. Experiments for this case 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. Figures 5.8 and 5.9 show the costs versus time plot for CGA, HTS and the second, so-called, *consolidated approach* for data sets with 7 crew members and 129 pairings, and 30 crew members and 279 pairings, respectively.

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 with the help of CGA.

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 start-up 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.

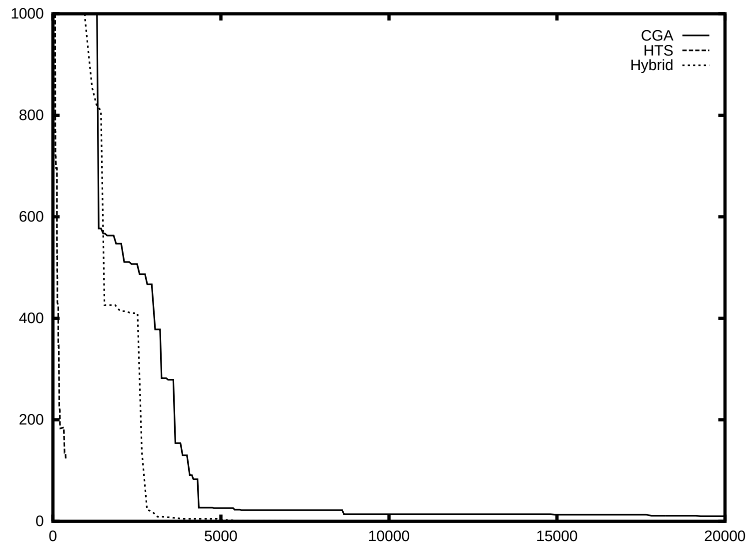


Fig. 5.8: Data set with 7 crew members and 129 pairings.

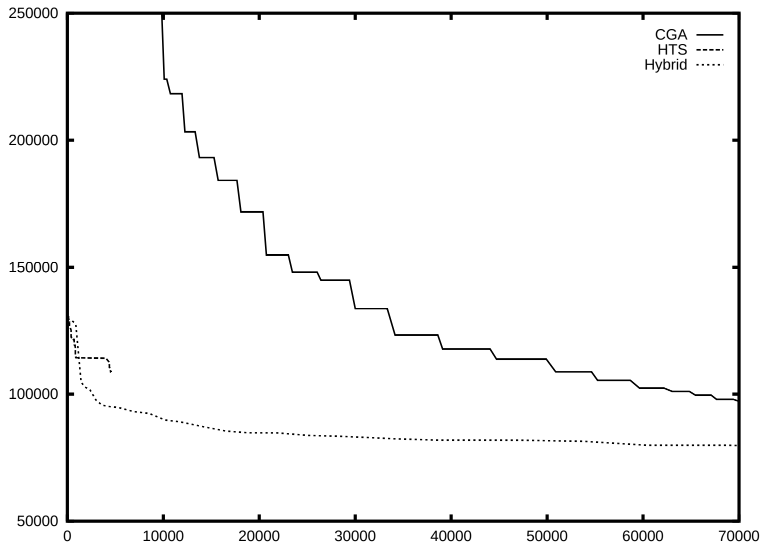


Fig. 5.9: Data set with 30 crew members and 279 pairings.

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. Moreover, 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.

The numerical 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 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:

- 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 B, 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 if it spends too much time on this phase of the process. This is not given in airline case A, 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. Its tuning heavily relies on inherent problem properties, which might not be known a priori.

5.5 Summary

For the CAP, we have shown how the concept of CP-based column generation that we presented in Section 3.1 works in practice. The sub-problem of roster generation can be viewed as a

Constrained Shortest Path Problem. We applied the filtering algorithms that were developed in Section 2.2 and gave a real-world empirical evaluation that showed the positive influence of cost-based filtering, especially when using an efficient, incremental implementation.

Although the column generation approach works reasonably, we found that it suffers from two major drawbacks: dummy costs and in-combinable rosters. Therefore, we presented a direct CP approach and merged the two together. We showed how methods from CP and OR can help each other to overcome their fundamental weak points.

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 that we proposed 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.

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.

Chapter 6

Automatic Recording

In Chapter 3, we have seen that the invocation of an optimization constraint is likely to become inefficient when it only represents a partial view on the entire problem. This causes that the bounds used for domain filtering are not accurate anymore, which then leaves the propagation algorithm ineffective. We have shown how problem decomposition can help to overcome this problem by linking optimization constraints via the objective rather than by the common interplay via variable domains only.

Implicitly we assumed that many real-world problems can actually be decomposed naturally into two or more basic substructures. In this chapter, we introduce the Automatic Recording Problem (ARP) [139] that is an example for such a composed problem. The ARP can be viewed as a combination of a Knapsack Problem (see Section 2.5) and a Maximum Weighted Stable Set Problem (see Section 2.3) on an interval graph. For this example, we show the benefits of linking a knapsack and a weighted stable set constraint via CP-based Lagrangian relaxation.

The work presented in this chapter was published in [188, 189, 190]. It is structured as follows: In Section 6.1, we formally introduce the ARP. Then, in Section 6.2, the concept of CP-based Lagrangian relaxation is applied to the problem. Finally, in Section 6.3, we give numerical results by evaluating the practical performance of different combined filtering algorithms for the ARP.

6.1 The Automatic Recording Problem

The technology of digital television offers new possibilities for individualized services that cannot be provided by current analog broadcasts. Additional information like classification of content, or starting and ending times can be submitted within the digital broadcast stream. With this information at hand, new services can be provided that make use of individual profiles and maximize customer satisfaction.

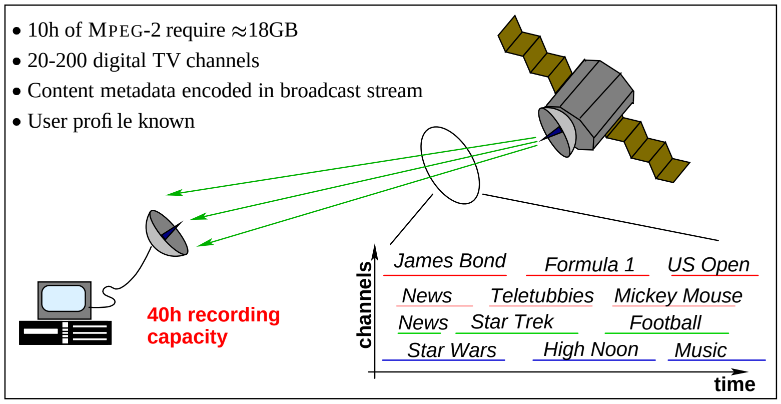


Fig. 6.1: The automatic recording scenario.

One service which is available already today [9, 208] is an "intelligent" digital video recorder that is aware of its user's preferences and records automatically. The 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 such that the user's satisfaction is maximized. As the number of channels may be enormous (more than 100 digital channels are possible), a service that automatically provides an individual selection is highly appreciated and subject of current research activities (for example within projects like *UP-TV* [210] funded by the European Union or the *TV-Anytime Forum*).

In this context, two restrictions have to be met. First, the storage capacity is limited (10 hours of MPEG-2 video needs about 18 GB). Second, only one program can be recorded at a time (see Figure 6.1).

More formally, we define the problem as follows:

Defi nition 6.1 *Let $n \in \mathbb{N}$, $V = \{1,\dots,n\}$ the set of programs, $start(i) < end(i)\ \forall\, i \in V$ the corresponding starting and ending times, $w = (w_i)_{1\le i\le n} \in \mathbb{Q}_+^n$ the storage requirements, $K \in \mathbb{Q}_+$ the storage capacity, and $p = (p_i)_{1\le i\le n} \in \mathbb{N}^n$ the profit vector.*

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 disc size, i.e. $\sum_{i\in X} w_i \le K$.

(b) At most one program is allowed to 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. $p_X \ge p_Y\ \forall\, Y \subseteq V$, Y respecting (a) and (b).

6.1.1 On the Complexity of the Automatic Recording Problem

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. Let $p_{max} := \max\{p_i \mid 1 \leq i \leq n\}$. We develop a pseudo-polynomial algorithm running in time $\Theta(n^2 p_{max})$ that will be used later to derive a fully polynomial time approximation scheme (FPTAS) for the ARP.

6.1.1.1 A Dynamic Programming Algorithm

The algorithm we develop in the following is similar to the teaching-book dynamic programming algorithm for Knapsack Problems. Setting $\overline{\mathbb{Q}} := \mathbb{Q} \cup \{\infty\}$ and $\psi := np_{max} + 1$, we compute a matrix $M = (m_{kl}) \in \overline{\mathbb{Q}}^{n \times \psi}$, $0 \leq k < \psi$, $1 \leq l \leq n$. In m_{kl}, we store the minimum knapsack capacity that is needed to achieve a profit greater or equal k when using items lower or equal l only ($m_{kl} = \infty$ iff $\sum_{1 \leq i \leq l} p_i < k$).

We assume that V is ordered with respect to increasing ending times, i.e., $1 \leq i < j \leq n$ implies $e_i \leq e_j$. Furthermore, let $last_j \in V \cup \{-1\}$ denote the last non-overlapping node lower than j, i.e.,

$$e_{last_j} < s_j \qquad \text{and} \qquad e_i \geq s_j \quad \forall\, last_j < i \leq j.$$

We set $last_j := -1$ iff no such node exists, i.e., iff $e_0 \geq s_j$. To simplify the notation, let us assume that $m_{k,-1} = \infty$ for all $0 < k < \psi$, and $m_{k,-1} = 0$ for all $k \leq 0$. Then,

$$m_{kl} = \min\{m_{k,l-1}, m_{k-p_l, last_l} + w_l\}.$$

The previous recursion equation yields a dynamic programming algorithm: First, we sort the items with respect to their ending times and determine $last_i$ for all $1 \leq i \leq n$. Both can be done in time $\Theta(n \log n)$. Then we build up the matrix row by row. Finally, we compute $\max\{k \mid m_{k,n} \leq K\}$. The total running time of this procedure and the memory needed are obviously in $\Theta(n^2 p_{max})$.

6.1.1.2 A Fully Polynomial Time Approximation Scheme

As for Knapsack Problems, we can use the dynamic programming algorithm to derive an FPTAS by scaling the profit vector. Given $\varepsilon > 0$, we set $S := \varepsilon p_{max}/n$, and $\overline{p_i} := \lfloor p_i/S \rfloor$. Then, $\overline{p_{max}} = \lfloor n/\varepsilon \rfloor$. Thus, the running time of our dynamic programming algorithm applied with the scaled profit vector is in $\Theta(n^3/\varepsilon)$.

Now let us study the error that we make by using $\overline{p}$ instead of p. Let $x \in \{0,1\}^n$ denote an optimal solution with respect to p, and $\overline{x} \in \{0,1\}^n$ an optimal solution with respect to $\overline{p}$. Then,

$$\begin{array}{rcccccc} p^T\overline{x} & \geq & S\lfloor p^T/S \rfloor \overline{x} & = & S\overline{p}^T\overline{x} & \geq & S\overline{p}^T x \\ & \geq & S((p^T x)/S - n) & = & p^T x - Sn. & & \end{array} \tag{6.1}$$

Therefore,

$$|p^T x - p^T \bar{x}|/p^T x \quad \leq \quad Sn/p_{max} \quad = \quad \varepsilon,$$

i.e., the relative error is at most ε. Thus, we have found an FPTAS for the ARP.

6.1.2 A Mathematical Programming Formulation

Since the problem of finding and proving optimal solutions is of interest in its own right, and also since the FPTAS we developed requires far too much memory to be applicable in practice, we focus on exact approaches for solving the ARP. Using mathematical programming, the problem can be stated as an integer linear program (IP):

$$\begin{array}{lrcll}
\text{Maximize} & IP_1 = p^T x & & & \\
\text{subject to} & x_i + x_j & \leq & 1 & \forall\, 1 \leq i < j \leq n,\ I_i \cap I_j \neq \emptyset \\
 & \sum_{1 \leq i \leq n} w_i x_i & \leq & K & \\
 & x & \in & \{0,1\}^n &
\end{array}$$

The objective function maximizes the user satisfaction. Constraints of the form $x_i + x_j \leq 1$ ensure that for overlapping intervals I_i, I_j, at most one program i or j can be selected. Memory restrictions are enforced by the last row. The formulation can be tightened when replacing the non-overlapping constraints by maximal clique constraints (see Section 2.3):

Denote the set of maximal conflict cliques by $M := \{C_1, \ldots, C_m\} \subseteq 2^V$. Then restrictions of the form $\sum_{i \in C_p} x_i \leq 1 \; \forall\, 1 \leq p \leq m$ 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 $\sum_{i \in C_p} x_i \leq 1 \; \forall\, 1 \leq p \leq m$. Thus, IP_1 is equivalent to

$$\begin{array}{lrcll}
\text{Maximize} & IP_2 = p^T x & & & \\
\text{subject to} & \sum_{i \in C_p} x_i & \leq & 1 & \forall\, 1 \leq p \leq m \\
 & \sum_{1 \leq i \leq n} w_i x_i & \leq & K & \\
 & x & \in & \{0,1\}^n &
\end{array}$$

Though being NP-complete on general graphs, finding maximal cliques on interval graphs that naturally model the non-overlapping constraints on the programs is simple. It can be performed in time $\Theta(n \log n)$ [99], and hence, IP_2 can be obtained in polynomial time.

6.1.3 Solving the Resulting Integer Linear Program

Although methods exist that do not split the search space – like cutting plane algorithms, for example – to solve a (mixed) integer linear program, branch-and-bound approaches have proven to be efficient, widely applicable and thus are most commonly used. In every choice point, a bound based on some (often continuous) relaxation is being computed. If that bound is worse than the objective value B of the incumbent solution, then backtracking occurs. A successful application of the branch-and-bound paradigm relies heavily on tight bounds that can be computed quickly. Problem reduction can help to improve the performance of a branch-and-bound search if the filtering algorithm is both effective and efficient. Effective means that it must have an impact, i.e., it has to be able to filter many values, whereas the efficiency measures how quickly the routine works.

The effectiveness of a filtering algorithm mainly depends on the quality of 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 on the solution quality that can be reached. Thus, it can be used for pruning purposes in a branch-and-bound approach. However, it is not straightforward to see how this bound could be used for filtering purposes effectively, that is, other than by probing via full re-optimization, which is inefficient. On the other hand, domain reductions with respect to reduced-cost information can be done quickly, but is not very effective. In the following, we will show how CP-based Lagrangian relaxation can help here.

6.2 CP-based Lagrangian Relaxation for the ARP

Using the refined model IP_2, the ARP can be viewed as a combination of two simpler optimization constraints: a knapsack constraint, and a maximum weighted stable set constraint on an interval graph. For the knapsack constraint, a filtering algorithm was developed in Section 2.5 that runs in time $\Theta(n \log n)$. Likewise, in Section 2.3, we developed a filtering algorithm for the maximum weighted stable set substructure (WSSP) of the ARP. The algorithm runs in time $\Theta(n \log n)$ or in amortized linear time for $\Omega(\log n)$ incremental propagation calls.

Provided with the two filtering algorithms, we are able to perform domain reduction for the two natural substructures of the ARP. According to the abstract description in Section 3.2, we will now tie the two filtering algorithms together:

As the filtering algorithm for the WSSP allows us to incorporate changing objectives at a low computational cost, we decide to relax the capacity constraint. We introduce a non-negative Lagrange multiplier $\lambda \geq 0$ and define the Lagrangian sub-problem

$$\begin{array}{lrcll} \text{Maximize} & \multicolumn{3}{l}{L(\lambda) = \sum_{1 \leq i \leq n} (p_i - \lambda w_i) x_i + \lambda K} & \\ \text{subject to} & \sum_{i \in C_p} x_i & \leq & 1 & \forall\, 1 \leq p \leq m \\ & x & \in & \{0,1\}^n & \end{array}$$

The Lagrange multiplier problem then is to minimize $L(\lambda)$, such that $\lambda \geq 0$. For every $\lambda \geq 0$, $L(\lambda)$ is a valid upper bound on the objective. Therefore, we can apply cost-based filtering for the weighted stable set constraint on interval graphs every time we solve the Lagrangian sub-problem. For given Lagrangian multipliers λ, we use dual information $\pi = \pi(\lambda) \in \mathbb{Q}^m$ from the corresponding stable set sub-problem to perform variable fixing with respect to the knapsack substructure next. Note that the algorithm developed in Section 2.3 provides us with those values at essentially no additional cost. By Lagrange relaxing the maximal clique constraints with multipliers $\pi \geq 0$, we obtain a Knapsack Problem. Let $\mu_i := \sum_{j \,:\, i \in C_j} \pi_j \; \forall\, 1 \leq i \leq n$ and $\overline{\pi} := \sum_{1 \leq j \leq m} \pi_j$. Then the problem is to

$$\begin{array}{lrcl} \text{Maximize} & \sum_{1 \leq i \leq n} (p_i + \mu_i) x_i - \overline{\pi} & & \\ \text{subject to} & \sum_{1 \leq i \leq n} w_i x_i & \leq & K \\ & x & \in & \{0,1\}^n \end{array}$$

Relaxations of this problem again yield a valid upper bound, and we can propagate the knapsack optimization constraint on the modified objective.

6.2.1 Implementation Details

We have so far left out some implementation details concerning the choice of the branching variable and the computation of optimal Lagrangian multipliers λ^*. In this section, we will give an insight in the implementation the tests are performed with.

We use four different approaches for our experiments: the first is a pure branch-and-bound algorithm without any problem tightening (referred to as *P-0*). The second uses the filtering algorithms for Knapsack and Maximum Weighted Stable Set Problems on the original objective (*P-1*). The third and the fourth approach (*P-2* and *P-3*) realize the idea of linking filtering algorithms for linear optimization constraints via Lagrangian relaxation. *P-2* calls for domain reduction with respect to both substructures just once after the Lagrangian dual has been solved, whereas *P-3* also propagates the maximum weighted stable set constraint during the search for optimal Lagrange multipliers.

6.2.1.1 Continuous Bound Computation

For pruning, the computation of a linear bound on the objective is needed. *P-2* and *P-3* obviously use the objective value corresponding to $L_{\mathcal{B}}(\lambda^*)$ for this purpose. As the computation via Lagrangian relaxation with stable set sub-problems turned out to be very efficient, we use that algorithm for all four approaches.

6.2.1.2 Computation of λ^*

To determine λ^*, we use a method to maximize one-dimensional concave functions based on the golden section. We obtain a sequence of λ^k, $k \in \mathbb{N}$. Let $e_{max} := \max\{p_i/w_i \mid 1 \leq i \leq n\}$. 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 Lagrange multiplier λ^*. Each iteration costs amortized linear time for a total of at least $\Omega(\log n)$ iterations in all search nodes. Finally, in every choice point we add $O(n \log n)$ for the succeeding knapsack filtering algorithm. Therefore, the integrated filtering algorithm for the tight global Lagrangian relaxation bound runs in time $O(n \log e_{max} + n \log n)$.

Notice that the Lagrangian sub-problem is totally unimodular, i.e., it exhibits the integrality property. Thus, the Lagrangian relaxation bound has the same value as the bound that is determined by a linear continuous relaxation.

6.2.1.3 Branching Variable Selection

Using the shortest path interpretation of the weighted stable set constraint on interval graphs (see Section 2.3.3), all algorithms choose the first node on the shortest path[1] with maximal efficiency p_i/w_i as branching variable.

6.3 Numerical Results

All experiments are 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 algorithms are built on top of ILOG SOLVER 5.0 [121].

6.3.1 Test Instance Generation

The experiments are conducted on several sets of randomly generated test instances. To achieve scenarios which we believe to be of relevance for the real-life 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 last. For each new program a *class* is being chosen randomly. That class then determines the interval from which the length is chosen randomly. We consider either 3, 5, or 7 different classes. The lengths of programs in the classes vary from 5±2 minutes to 150±50 minutes. The disc 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, it remains to choose the associated profits of programs. For the experiments, we use 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.

[1]According to the optimal reduced-costs objective $\sum_{1\leq i\leq n}(p_i - \lambda^* w_i)x_i + \lambda^* K$.

- 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 that program has a non-empty intersection with.
- For the *weakly correlated (TWC)* instances, that 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 that we believe to hold for real-life instances. In the CU instances for example, programs of the same class cause similar attractions. On the other hand, the TC and TWC instances cause many conflicts regarding the choice of programs that are being broadcasted at the same time. The assumption that programs overlapping in time cause similar attractions is justified by the fact that TV channels are planning their broadcasts according to the behavior of target groups. To a large extent, these target groups are determined by and vary with the time of the day. However, the different strategies that we consider are only intuitively justified. The feasibility of our approach for real-life 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 giving the parameters the generator is started with. According to the previous description those parameters are: The time horizon in minutes, the number of channels, the number of different classes [3, 5, or 7], and the objective type [CU, TC, TWC, or SSS].

6.3.2 Experimental Evaluation

In the following, we present our numerical results. The experiments consist of 50 random instances per test set. For each instance, the approaches *P-0 – P-3* are run to find and prove an optimal solution. We give running times and the number of choice points needed for an exhaustive search. All approaches find a first solution rather early in the search. Therefore, the main work consists in the proof of optimality rather than in the construction of the solution. We conclude that the branching variable selection that we use efficiently supports finding near-optimal solutions in a non-exhaustive search.

Table 6.1 shows the performance (time and choice points) of all four 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 approaches, the TC instances are much harder than CU and TWC, which are comparably easy 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.

We further observe that a higher degree of integration between the two optimization constraints yields partially drastic reductions in the number of choice points of up to a factor of

test set	*P-0*		*P-1*		*P-2*		*P-3*	
5 12h 20 ch	*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

Tab. 6.1: The table compares the different approaches on three different test sets with 5 classes, 12 hours, 20 channels and different objectives. The time (in seconds) and the number of choice points are averages for 50 randomly generated instances for each objective. The average number of programs per instance are between 607.6 and 612.6.

about 70 on the difficult time correlated instances. Regarding the computation time, there is a trade-off between the reduction of choice points and the time spent per choice point (TpCP). The TC and TWC instances show that *P-2* can outperform *P-3* because of the shorter TpCP that is needed for that degree of integration. When comparing *P-1* and *P-3* on the CU instances, the reduction of choice points is not big enough to justify the longer TpCP needed, and *P-1* is the approach that takes the least computation time.

Generally, a bigger reduction of choice points is more likely to pay off when the absolute TpCP needed is rather high. Particularly, this holds for applications where additional constraints are propagated on top of the objective constraint itself. For the ARP, the optimization constraint is the only active constraint. Therefore, to justify the worse TpCP caused by the more complicated propagation algorithm, a substantial reduction of the number of choice points must be achieved. The *P-2* and *P-3* approaches obtain a sufficient reduction of choice points on the more difficult TC test sets, and also for larger test instances:

Table 6.2 shows the performances of all approaches on test instances that are generated using 5 different program classes with different time horizons and numbers of channels. The objective is 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 linking approaches *P-2* and *P-3* outperform *P-1* roughly by a factor of 4 regarding the number of choice points and almost a factor of 2 with respect to the computation time needed. The minima, maxima and standard deviations prove that the average numbers we present are not biased by very few outliers, but represent meaningful values for the evaluation of the algorithms performances.

In Table 6.3, we compare the different approaches on a variety of very different test instances that are generated using different parameters and objective functions. Again, relevant and partially substantial reductions in the number of choice points can be obtained by CP-based Lagrangian relaxation realized in *P-2* and *P-3*.

test set		*P-0*		*P-1*		*P-2*		*P-3*	
5 CU		*time*	*nodes*	*time*	*nodes*	*time*	*nodes*	*time*	*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

Tab. 6.2: The table shows a comparison of the performance of the different approaches on 5 test sets with 5 classes and objective CU for various time horizons (in hours) and channel numbers (ch). Italic numbers give the average time (in seconds) and the average number of nodes of 50 randomly generated instances in each test set (avg). 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), respectively.

	P-0		*P-1*		*P-2*		*P-3*	
test set	*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

Tab. 6.3: The table illustrates the performance of the different approaches on very different benchmark classes. Each test set contains 50 randomly generated problem instances. There is an average of 1956.7 programs in the 120h/20ch test set, 1782.6 programs in test set 72h/20ch, and 1423.3 programs in test set 24h/50ch.

test set			*P-0*		*P-1*		*P-2*		*P-3*	
5 SSS			*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

Tab. 6.4: The table shows the performance of the different approaches on subset sum data sets ranging from 12 hours and 20 channels up to 72 hours and 50 channels. The average number of programs in the 50 randomly generated instances per test set is given as parameter p.

So far, we have left out comparisons regarding the choice of the objective according to SSS. Table 6.4 shows the results obtained for a collection of very different test sets generated with SSS. Two facts stand out: first, a comparison with Table 6.1 shows that the SSS instances are much easier to solve than for other choices of the objective. Second, *P-1* achieves only a slight reduction of choice points compared to *P-0* that cannot be improved by *P-2* and *P-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. Due to the equal efficiency p_i/w_i of all programs, the knapsack optimization constraint has great difficulties to include or exclude programs. Therefore, the knapsack constraint is not effective, and the burden of domain reduction lies on the WSSP optimization constraint only. In total, using the optimization constraint for pruning purposes only is most time efficient here.

test set	*avg. no*	*P-0*		*P-1*		*P-2*		*P-3*	
3 CU	*of programs*	*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

Tab. 6.5: The table compares the performance of the different algorithms on benchmark sets with 3 classes and objective CU, each containing 50 randomly generated problem instances with roughly 1000 programs on average.

Finally, we investigate the impact of the number of channels. Table 6.5 shows a comparison of three different test sets that are generated using 3 different program classes and CU objectives. All instances have a similar size and roughly contain 1000 programs. We observe that the instances become more difficult to solve for all approaches when the number of channels is decreasing. This surprising result may be caused by the fact that a smaller number of channels increases the relative importance of the knapsack optimization constraint that is inherently more difficult than the WSSP constraint. However, a sound answer to that question can only be given by further investigation. It remains to note that we observe a converse behavior for TC data sets: the instances become more difficult the more channels are involved which is obviously caused by many temporally conflicting programs of similar value.

6.4 Summary and Future Work

We have shown that the Automatic Recording Problem can be viewed as a composition of a Knapsack Problem and a Weighted Stable Set Problem on an interval graph. By introducing an FPTAS, we showed that the problem, though being NP-hard, can be approximated with arbitrary approximation quality. Then, using an exact tree search approach, we showed the benefits of linking filtering algorithms via Lagrangian relaxation. The numerical results we obtained show a significant improvement achieved by CP-based Lagrangian relaxation with respect to the computation time and the number of choice points.

There are several natural extensions for the ARP. For example, a digital video recorder could have more than one recording unit which allows the recording of a limited number of channels simultaneously. In an IP context, this modification can be introduced easily. For the exact approach presented, a fast and efficient filtering algorithm for this type of relaxed non-overlapping constraint is subject to further research.

Chapter 7

Capacitated Network Design

In the last chapter, we presented the Automatic Recording Problem (ARP) as an example of a discrete optimization problem that can be naturally decomposed into two basic substructures. Now, we want to develop a branch & bound approach for the Capacitated Network Design Problem, that can also be viewed as a conglomerate of two simple constraint families defined by the mass-balance and the bundle constraints. We investigate both induced Lagrangian sub-problems and develop problem reduction algorithms for them. As we shall see, applying the strongest version of CP-based Lagrangian relaxation is inefficient for this problem. Therefore, we generalize the idea of cost-based domain filtering, that may be viewed as an adding of unary local constraints. An algorithm is presented that adds local cardinality cuts based on Lagrangian relaxation. Their usefulness is evaluated empirically by numerous tests.

The work presented in this chapter was published in [136, 193]. It is structured as follows: In Section 7.1, we introduce the Capacitated Network Design Problem (CNDP). To solve the problem, we use bounds, variable fixing algorithms and local cardinality cuts based on Lagrangian relaxation as described in Section 7.2. The entire branch & bound-approach is described in Section 7.3. Finally, in Section 7.4, we give numerical results.

7.1 The Capacitated Network Design Problem

The Capacitated Network Design Problem was first defined in [91] and is relevant for a wide area of applications, ranging from telecommunications to transportation problems [97, 144]. The problem consists in finding an optimal subset of arcs in a network $G = (V,E)$ such that we can transport a given demand $d^k \in \mathbb{Q}^{|V|}$ of goods $1 \leq k \leq K$ (so-called *commodities*) at optimal total cost. The latter consists of two components: the flow costs and the design costs. The flow cost is the sum of costs for the routing of each commodity, whereby, for each arc (i, j) and commodity k, a scalar $c_{ij}^k \geq 0$ determines the cost of routing one unit of commodity k via (i, j). The design costs are determined by the costs of installing the chosen arcs, whereby, for each arc (i, j), we are given a fixed arc-installation cost f_{ij}. Additionally, for each arc, there is a capacity u_{ij} given that limits the total amount of flow that can be routed via (i, j).

For all arcs $(i, j) \in E$ and commodities $1 \leq l \leq K$, let $b^l_{ij} = \min\{|d^l|, u_{ij}\}$. Using variables $x^l \in \mathbb{Q}^{|E|}_+$ for the flows and $y \in \{0,1\}^{|E|}$ for the design decisions, a mixed integer program formulation for the Capacitated Network Design Problem can be stated as follows:

$$\begin{array}{llcll}
\text{Minimize} & L_{CNDP} = \sum_l (c^l)^T x^l + f^T y & & & \\
\text{subject to} & Nx^l & = & d^l & & (1)\\
& \sum_l x^l_{ij} & \leq & u_{ij}y_{ij} & \forall\, (i,j) \in E & (2)\\
& x^l_{ij} & \leq & b^l_{ij}y_{ij} & \forall\, (i,j) \in E,\ 1 \leq l \leq K & (3)\\
& x & \geq & 0 & & (4)\\
& y & \in & \{0,1\}^{|E|} & & (5)
\end{array}$$

For ease of notation, we refer to the previous LP with L_{CNDP}, which is also used to denote the optimal objective value. The network flow constraints (also called *mass-balance constraints*) (1) are defined by the node-arc-incidence matrix $N = (n_{ia})_{i\in V, a\in E}$ and a demand vector $d^k \in \mathbb{Q}^{|V|}$ for all commodities k, whereby $n_{ia} = 1$ iff $a = (h, i)$, $n_{ia} = -1$ iff $a = (i, h)$, and $n_{ia} = 0$ otherwise, and $d^k_i > 0$ iff node $i \in V$ is a demand node and $d^k_i < 0$ iff node i is a supply node for commodity k. Without loss of generality, we may assume that there is exactly one demand node and one supply node for each commodity [112].

The total flow on an arc (i, j) is constrained by the capacity u_{ij} (so-called *capacity* or *bundle constraints* (2)). The set of *upper bound constraints* (3) is redundant to the problem formulation. It is introduced to strengthen the linear continuous relaxation of the mixed integer problem.

7.1.1 State of the Art

The CNDP is an NP-hard problem, which is easy to see by reduction to the Steiner Tree Problem. Since the latter is MAXSNP-hard [20], we cannot even hope for a fully polynomial approximation scheme (FPTAS) for the CNDP. Here, we focus on exact and heuristic solution approaches.

Regarding exact solution approaches, Crainic, Frangioni, and Gendron develop lower bounding procedures for the CNDP [44]. The main insights are the following: Tight approximations of the so-called *strong* LP-relaxation (see L_{CNDP} including the redundant constraints (3)) can be found much faster by Lagrangian relaxation than by optimizing the LP using standard LP-solvers. The authors investigate so-called *shortest path* and *knapsack relaxations* (see Section 7.2). When solving the Lagrangian dual, bundle methods converge faster than ordinary subgradient methods and are more robust. Motivated by this successful work, we evaluate several Lagrangian relaxations in the context of branch & bound.

In [112], Holmberg and Yuan present a method to compute exact or heuristic solutions for the CNDP. They use the Lagrangian *knapsack relaxation* in each node of the branch & bound tree to efficiently compute lower bounds. Special penalty tests were developed which correspond to variable fixing strategies presented in the following. An evaluation of the following components is given: subgradient search procedure for solving the Lagrangian dual, primal heuristic for finding feasible solutions, and interplay between branch & bound and the subgradient search. On top of that work, a heuristic is developed that is embedded in the tree search procedure. That heuristic is able to provide near-optimal solutions for large CNDP instances which are out of reach of exact methods like Lagrangian relaxation based branch & bound or branch-and-cut approaches (represented by the CPLEX implementation, for example).

In [22], Bienstock et al. describe two cutting-plane algorithms for a variant of the CNDP with multi-arcs (i.e., an arc can be inserted multiple times). One of them is based on the multi-commodity formulation of CNDP and uses cutset and three-partition inequalities. The other one adds total capacity, partition, and rounded metric inequalities. In a branch-and-cut framework, both variants provide sound results on a benchmark of realistic data. A substantial improvement of this procedure is achieved by Bienstock in [23]: a branch-and-cut algorithm based on ε-approximations of linear programs performs better on the same benchmark data.

Further branch-and-cut algorithms for the Multi-Arc CNDP are investigated in research papers by Günlük, and Atamtürk et al. [7, 102]. Several valid inequalities are considered, and it is found that branch-and-cut is substantially superior to branch & bound code on the investigated benchmark data. After adding the cuts, the integrality gap at the root node is reduced significantly and the total number of evaluated nodes diminishes. These results emphasize the importance of tight lower bounds for the CNDP.

Sridhar and Park report about an implementation of a Benders-and-cut algorithm for the CNDP [204]. The algorithm consists of three parts: a cutting plane algorithm for the computation of tight lower bounds, a heuristic to generate feasible solutions, and the Benders-and-cut algorithm itself. The computational results provided in the paper are based on a wide range of instances with varying traffic demand. The complexity of the problem instances depends heavily on the capacity provided in the network. CNDP instances with low transportation demand are easy to solve, whereas for problem instances with higher demand the authors suggest to use the Benders-and-cut algorithm. In cases when the traffic becomes extremely dense, the integrality gaps increase. To strengthen the LP-relaxation, flow inequalities are very effective.

A branch-and-price algorithm for the path-based formulation of the CNDP is compared with the traditional branch & bound for the link-based formulation by Clarke and Gong in [40]. The path-based formulation is computationally more efficient. Adding SOS-branching on each origin and destination node increases this advantage. The efficient solution of the pricing problem (which is a simple Shortest Path Problem) enables a faster solving of the LP-relaxations in the branch & bound tree. Computational results on instances with 6, 10, and 15 nodes are reported.

7.2 Lagrangian Relaxation Bounds

The CNDP can be viewed as a mixture of a continuous and a discrete optimization problem. The latter is obviously constituted by the design variables, whereas the first is a Min-Cost Multicommodity Flow Problem (MMCF) that evolves when the design variables are fixed. For the MMCF, besides linear programming solvers, especially cost-decomposition approaches based on Lagrangian relaxation have been applied successfully [85]. The bounds we will use for the CNDP will be based on those cost-decomposition approaches for the MMCF.

Regarding the MMCF and also for the CNDP, we are left with two promising choices of which hard constraints should be softened:

- the bundle constraints ("shortest path relaxation"), or
- the mass-balance constraints ("knapsack relaxation").

7.2.1 Shortest Path Relaxation

Assume, in L_{CNDP}, for each arc $(i,j) \in E$ we introduce a Lagrangian multiplier $\lambda_{ij} \leq 0$ and transfer the corresponding bundle constraint into the objective function. At the same time, we also relax the upper bound constraints $x^l_{ij} \leq b^l_{ij}$ using penalty costs $\nu^l_{ij} \leq 0$. Let $\otimes : \mathbb{Q}^n \times \mathbb{Q}^n \to \mathbb{Q}^n$ denote the component-wise product of two vectors, i.e. $z = x \otimes y$ iff $z_s = x_s y_s$ $\forall\, x,y \in \mathbb{Q}^n$ and $1 \leq s \leq n$. Then we get the following linear program:

$$\begin{array}{lrcll}
\text{Minimize} & \multicolumn{4}{l}{L_{SP}(\lambda,\nu) = \sum_l (c^l - \lambda - \nu^l)^T x^l + (f + \lambda \otimes u + \sum_l \nu^l \otimes b^l)^T y} \\
\text{subject to} & Nx^l & = & d^l & \forall\, 1 \leq l \leq K \\
 & x & \geq & 0 & \\
 & y & \in & \{0,1\}^{|E|} &
\end{array}$$

The theory of Lagrangian relaxation shows that, for every choice of the Lagrangian multipliers $\lambda, \nu \leq 0$, $L_{SP}(\lambda,\nu)$ is a lower bound on the CNDP. Notice that this value can be obtained easily, because there is no cross-talking between variables y and x and among the variables x^l anymore. Thus, we can compute $L_{SP}(\lambda,\nu)$ by solving K problems of the form

$$\begin{array}{lrcl}
\text{Minimize} & \multicolumn{3}{l}{L^l_{SP}(\lambda,\nu) = (c^l - \lambda - \nu^l)^T x^l} \\
\text{subject to} & Nx^l & = & d^l \\
 & x^l & \geq & 0
\end{array}$$

and by setting $y_{ij} = 1$ iff $f_{ij} + \lambda_{ij} u_{ij} + \sum_k \nu^l_{ij} b^l_{ij} < 0$, and $y_{ij} = 0$ otherwise. That is, we can solve the Lagrangian sub-problem mainly by computing K Single Source Shortest Path Problems with positive arc weights. Therefore, the shortest path sub-problem can be solved in time $O(K(m + n\log n))$.

7.2.2 Knapsack Relaxation

The other promising alternative is to relax the mass-balance constraints in L_{CNDP}. For the constraints to be relaxed, we introduce Lagrangian multipliers μ_i^l for all $1 \le l \le K$ and $i \in V$. We get the following linear program:

$$\begin{array}{lrcll}
\text{Minimize} & \multicolumn{4}{l}{L_{KP}(\mu) = \sum_l \sum_{ij} (c_{ij}^l + \mu_i^l - \mu_j^l)^T x_{ij}^l + f^T y + \mu^T d} \\
\text{subject to} & \sum_l x_{ij}^l & \le & u_{ij} y_{ij} & \forall\,(i,j) \in E \\
& x_{ij}^l & \le & b_{ij}^l y_{ij} & \forall\,(i,j) \in E \\
& x & \ge & 0 & \\
& y & \in & \{0,1\}^{|E|} &
\end{array}$$

Whereas the shortest path relaxation decomposes the Lagrangian sub-problem by the different commodities, here we achieve an arc-wise decomposition. To solve the previous LP, for each $(i,j) \in E$ we consider the following linear program, that is similar to the linear continuous relaxation of a Knapsack Problem:

$$\begin{array}{lrcll}
\text{Minimize} & \multicolumn{4}{l}{L_{KP}^{(i,j)}(\mu) = \sum_l \overline{c}_{ij}^l \overline{x}_{ij}^l} \\
\text{subject to} & \sum_l \overline{x}_{ij}^l & \le & u_{ij} & \\
& \overline{x}_{ij}^l & \le & b_{ij}^l & \forall\, 1 \le l \le K \\
& \overline{x} & \ge & 0 &
\end{array}$$

where $\overline{c}_{ij}^l = c_{ij}^l + \mu_i^l - \mu_j^l$. For each $(i,j) \in E$, we set $x_{ij}^l = \overline{x}_{ij}^l$ for all $1 \le l \le K$, and $y_{ij} = 1$, iff $f_{ij} + L_{KP}^{(i,j)}(\mu) < 0$. Otherwise, we set $x_{ij}^l = 0$ for all $1 \le l \le K$, and $y_{ij} = 0$. Obviously, this setting provides us with an optimal solution for $L_{KP}^{(i,j)}(\mu)$. Thus, the main effort is to solve the problems $L_{KP}^{(i,j)}(\mu)$. However, this is an easy task (compare with [147, 149]): first, we can eliminate all variables with positive cost coefficients, i.e., we set $\overline{x}_{ij}^l = 0$ for all $1 \le l \le K$ with $\overline{c}_{ij}^l \ge 0$. Next, we sort the $\overline{x}_{ij}^l$ according to increasing cost coefficients $\overline{c}_{ij}^l$, that is, from now on we may assume that $\overline{c}_{ij}^l < \overline{c}_{ij}^{l+1} < 0$ for all $1 \le l < s \le K$, whereby s is the number of negative objective coefficients. Let $k \in \mathbb{N}$ denote the *critical item* with $k = \min\{l \le s \mid \sum_{h \le l} b_{ij}^h > u_{ij}\} \cup \{s+1\}$. We obtain $L_{KP}^{(i,j)}(\mu)$ by setting $\overline{x}_{ij}^h = b_{ij}^h$ for all $h < k$, $\overline{x}_{ij}^h = 0$ for all $h > \min\{k,s\}$, and, in case of $k < s+1$, $\overline{x}_{ij}^k = u_{ij} - \sum_{h<k} b_{ij}^h$. Thus, the knapsack sub-problem can be solved in time $O(|E|(K \log K))$.

Note that both relaxations exhibit the integrality property. Thus, the bound we achieve in both settings equals the linear continuous relaxation bound of the CNDP [44].

7.2.3 Subgradient Optimization

For both relaxation types, the Lagrangian dual consists in maximizing the lower bound. That is, for the shortest path relaxation we have to maximize $L(\lambda,\nu)$ subject to $\lambda, \nu \le 0$. For the knapsack relaxation, our task is simply to maximize $L(\mu)$. A popular way of solving these concave, piecewise linear maximization problems over a convex region is to apply a subgradient search (see [1] for a general introduction). Several proposals regarding the specification of the general method have been made in the CNDP literature. Let s^t denote a subgradient in iteration t. We compute the new search direction by setting $d^t = (s^t + \alpha d^{t-1})/(1+\alpha)$ [112]. We also experimented with other approaches to solve the Lagrangian dual, such as the modified Camerini-Fratta-Maffioli rule [28] or the volume algorithm [11]. Without going into details here, we just note that none of these modifications yield visible improvements on the overall performance.

7.2.4 Variable Fixing

A big advantage of Lagrangian relaxation based bound computations is that they can be used for variable fixing in a very efficient way. In the presence of an optimal or at least high quality upper bound $B \in \mathbb{Q}$ for the CNDP, it is an easy task to check whether a variable y_{ij} can still be set to either of its bounds without worsening the lower bound too much. More formally, given the Lagrangian multipliers λ and ν in the current shortest path sub-problem, a value $l \in \{0,1\}$ and any arc $(i,j) \in E$, we can set

$$y_{ij} = l \quad \text{if } (2l-1)(f_{ij} + \lambda_{ij} u_{ij} + \sum_l \nu_{ij}^l b_{ij}^l) > B - L_{SP}(\lambda, \nu). \tag{7.1}$$

Analogously, given the Lagrangian multipliers μ in the current knapsack sub-problem, a value $l \in \{0,1\}$ and any arc $(i,j) \in E$, we can set

$$y_{ij} = l \quad \text{if } (2l-1)(f_{ij} + L_{KP}^{(i,j)}(\mu)) > B - L_{KP}(\mu). \tag{7.2}$$

Now, we have two variable fixing algorithms at hand for two different Lagrangian sub-problems. Therefore, we are able to apply the concept of CP-based Lagrangian relaxation (see Section 3.2). First, we can choose one of the two alternatives (for example the one for which the Lagrangian dual can be solved more quickly) and apply the corresponding variable fixing algorithm in every Lagrangian sub-problem. If we find that this procedure does not filter enough values, we can do even more: with the help of dual values gained in the solution process of the Lagrangian sub-problem, in every Lagrange iteration we can apply both variable fixing algorithms.

Let us assume we decide to use the shortest path relaxation. Given the current Lagrangian multipliers λ and ν, in every iteration we need to solve K Shortest Path Problems. Using the well-known Dijkstra algorithm for that purpose, we do not only get the optimal objective of each problem, we also get the shortest path distances $\overline{\mu}_i^l$ of each node i for free. It is known for a long time that those distances are optimal dual values in the corresponding LP. The idea of the linking method consists in using these duals as Lagrangian multipliers for the knapsack sub-problem next. That is, we consider $L_{KP}(\overline{\mu})$. That way, we can apply the variable fixing algorithms for both sub-problems. Note that, for optimal Lagrangian multipliers λ and ν, the shortest path distances $\overline{\mu}$ in combination with the multipliers are also optimal for the dual of L_{CNDP}.

When using the knapsack relaxation, the situation is slightly more complicated, because we need to provide dual values for the bundle as well as the upper bound constraints. Given the current Lagrangian multipliers μ, we solve $|E|$ knapsack sub-problems as described in 7.2.2. Again, when given any arc $(i,j) \in E$, we assume that $s \in \mathbb{N}$ denotes the number of negative cost coefficients in $L_{KP}^{(i,j)}(\mu)$, that the remaining variables $\overline{x}_{ij}^l$ are ordered with respect to increasing cost coefficients, and that $k \leq s+1$ is the critical item.

In case of $k < s+1$, we set $\overline{\lambda}_{ij} = \overline{c}_{ij}^k$, $\overline{\nu}_{ij}^l = \overline{c}_{ij}^l - \overline{c}_{ij}^k$ for all $l < k$ and $\overline{\nu}_{ij}^l = 0$ for all $l \geq k$. For $k = s+1$, we set $\overline{\lambda}_{ij} = 0$, $\overline{\nu}_{ij}^l = \overline{c}_{ij}^l$ for all $l < k$ and $\overline{\nu}_{ij}^l = 0$ for all $l \geq k$.

Theorem 7.1 *The vectors $\overline{\lambda}$ and $\overline{\nu}$ define optimal dual values for $L_{KP}(\mu)$.*

Proof: It is sufficient to show that the value $\overline{\lambda}_{ij}$ and the vector $\overline{\nu}_{ij} = (\overline{\nu}_{ij}^1, \ldots, \overline{\nu}_{ij}^K)$ give an optimal dual solution for $L_{KP}^{(i,j)}(\mu)$ for all $(i,j) \in E$. The dual of $L_{KP}^{(i,j)}(\mu)$ is the following linear program:

$$\begin{array}{lrcll} \text{Maximize} & \multicolumn{3}{l}{D_{KP}^{(i,j)}(\mu) = u_{ij}\lambda_{ij} + \sum_l b_{ij}^l \nu_{ij}^l} & \\ \text{subject to} & \lambda_{ij} + \nu_{ij}^l & \leq & \overline{c}_{ij}^l & \forall\, 1 \leq l \leq K \\ & \lambda, \nu & \leq & 0 & \end{array}$$

First, we show that our solution is dual feasible:

$\mathbf{k < s+1:}$ It holds that $\overline{\lambda}_{ij} = \overline{c}_{ij}^k \leq 0$, because $k \leq s$. Also, if $l < k$, then $\overline{\nu}_{ij}^l = \overline{c}_{ij}^l - \overline{c}_{ij}^k \leq 0$, because $\overline{c}_{ij}^l \leq \overline{c}_{ij}^k$. If $l \geq k$, then $\overline{\nu}_{ij}^l = 0$. Thus, all values are non-positive.

Furthermore, if $l < k$, then it holds that $\overline{\lambda_{ij}} + \overline{\nu}_{ij}^l = \overline{c}_{ij}^k + \overline{c}_{ij}^l - \overline{c}_{ij}^k = \overline{c}_{ij}^l$. For $l \geq k$, it holds that $\overline{\lambda_{ij}} + \overline{\nu}_{ij}^l = \overline{c}_{ij}^k \leq \overline{c}_{ij}^l$.

$\mathbf{k = s+1:}$ It holds that $\overline{\lambda}_{ij} = 0$ and $\overline{\nu}_{ij}^l = \overline{c}_{ij}^l \leq 0$ for all $l < k$, because this implies $l \leq s$. Furthermore, $\overline{\nu}_{ij}^l = 0$ for all $l \geq k$. Thus, all values are non-positive.

For $k = s+1$, we have that $\overline{\lambda}_{ij} + \overline{\nu}_{ij}^l = \overline{c}_{ij}^l$ for all $1 \leq l \leq K$.

To prove the optimality of the dual solution (and, at the same time, also for the primal solution we gave in Section 7.2.2), we show that the two objective values for $\overline{x}_{ij}$ in $L_{KP}^{(i,j)}(\mu)$ and $\overline{\lambda}_{ij}$, $\overline{\nu}_{ij}$ in $D_{KP}^{(i,j)}(\mu)$ match each other:

$$\begin{array}{llcl} \mathbf{k < s+1:} & \sum_l \overline{c}_{ij}^l \overline{x}_{ij}^l & = & \sum_{l<k} \overline{c}_{ij}^l b_{ij}^l + \overline{c}_{ij}^k (u_{ij} - \sum_{l<k} b_{ij}^l) = \\ & \sum_{l<k} b_{ij}^l (\overline{c}_{ij}^l - \overline{c}_{ij}^k) + u_{ij}\overline{c}_{ij}^k & = & \sum_l b_{ij}^l \nu_{ij}^l + u_{ij}\lambda_{ij}. \end{array}$$

$$\mathbf{k = s+1:} \quad \sum_l \overline{c}_{ij}^l \overline{x}_{ij}^l = \sum_{l \leq s} \overline{c}_{ij}^l b_{ij}^l = \sum_l b_{ij}^l \nu_{ij}^l + u_{ij}\lambda_{ij}. \qquad \blacksquare$$

To summarize: if we choose the shortest path relaxation, in every Lagrangian sub-problem we solve K shortest path sub-problems and achieve a lower bound on the CNDP. If that bound is worse than the best known or even optimal objective value B, we can prune the current choice point. Otherwise, we fix variables according to Implication 7.1. Then we set up $L_{KP}(\overline{\mu})$, where $\overline{\mu}$ are the shortest path distances, and fix variables according to Implication 7.2.

If, instead, we choose to use the knapsack relaxation, in every Lagrangian sub-problem we solve $|E|$ linear continuous Knapsack Problems and achieve a lower bound for the CNDP. Again, if that bound is worse than B, we can prune the current choice point. Otherwise, we fix variables according to Implication 7.2. Then we set up $L_{SP}(\overline{\lambda},\overline{\nu})$, where $\overline{\lambda}$ and $\overline{\nu}$ are the dual values of $L_{KP}(\mu)$ in Theorem 7.1, and fix variables according to Implication 7.1.

Of course, many variants of the algorithm sketched above can be thought of. For example, we may find that applying both variable fixing algorithms in every Lagrangian sub-problem is too time consuming and does not pay off. Then we can introduce frequency parameters that control the percentage of Lagrangian sub-problems for which either of the two variable fixing algorithms is applied. As an extreme choice, we may for example decide to apply variable fixing for the optimal Lagrangian multipliers only. As our experiments show, it is favorable to use the knapsack relaxation to solve the Lagrangian dual quickly. As one would expect, solving K Shortest Path Problems in every Lagrangian sub-problem in addition to the $|E|$ knapsack sub-problems is rather costly and slows down the solution process considerably. The following theorem helps to cope with this situation more efficiently.

Theorem 7.2 *Given Lagrangian multipliers μ in the knapsack relaxation, denote some optimal dual values for $L_{KP}(\mu)$ by $\overline{\lambda} \leq 0$ and $\overline{\nu} \leq 0$. Then,*

$$L_{SP}(\overline{\lambda},\overline{\nu}) \geq L_{KP}(\mu). \tag{7.3}$$

Proof: Let $\overline{\kappa} \leq 0$ denote optimal dual values for the upper bound constraints of y in $L_{KP}(\mu)$. The vectors μ and $\overline{\kappa}$ are dual feasible for $L_{SP}(\overline{\lambda},\overline{\nu})$, because $\overline{\lambda}$, $\overline{\nu}$ and $\overline{\kappa}$ are primal optimal for $L_{KP}(\mu)$, and thus

$$\begin{aligned} \mu_j^k - \mu_i^k &\leq c_{ij}^k - \lambda_{ij} - \nu_{ij}^k \qquad \forall\, (i,j) \in E,\ 1 \leq k \leq K, \quad \text{and} && (7.4)\\ -\overline{\kappa}_{ij} &\leq f_{ij} - u_{ij}\lambda_{ij} - \sum_k b_{ij}^k \nu_{ij}^k \qquad \forall\, (i,j) \in E && (7.5) \end{aligned}$$

Any dual feasible solution for a minimization problem determines a valid lower bound on that problem. Denote the dual of an optimization problem L by D. Then,

$$L_{SP}(\overline{\lambda},\overline{\nu}) = D_{SP}(\overline{\lambda},\overline{\nu}) \geq \mu^T d + 1^T \overline{\kappa} = D_{KP}(\mu) = L_{KP}(\mu). \tag{7.6}$$

■

We can use this theorem to relax Implication 7.1, which improves the running time of our variable fixing algorithm, but makes it also less effective. Unfortunately, as we shall see in Section 7.4, even in its strong version the shortest path variable fixing algorithm is already too ineffective, and therefore this idea cannot be used to improve the running time of a CNDP solver.

Another important choice is to decide when the effects of the variable fixing algorithms should be made visible to the subgradient algorithm. Obviously, when we fix variables during the process of finding optimal Lagrangian multipliers, we are actually changing the problem. Thus, with respect of the convergence of the iterative procedure, we have to reset. The most defensive strategy is to start from scratch (with the current Lagrangian multipliers as starting point) and to set the step-length coefficient to its initial setting. However, we may consider to allow only a little more flexibility by scaling the factor by a fixed percentage. A totally different approach would be not to incorporate the changes of the bounds of the variables during the optimization process, but to restart the whole procedure (if any variable fixing has taken place) after optimal Lagrangian multipliers have been found. We evaluate different parameter settings in the numeric section.

7.2.5 Lagrangian Cardinality Cuts

Since CP-based Lagrangian relaxation is partly inefficient for the CNDP, we generalize the idea of domain filtering. Variable fixing may be viewed as an addition of unary constraints that force a variable to take a specific value. Domain filtering that achieves a state of bound consistency may be viewed as a procedure that adds unary interval constraints. Even general domain filtering can be viewed as an addition of unary constraints that enforce a variable not to take some specific values. In general, all these additional constraints are only valid locally. Now, we do not want to restrict ourselves to unary constraints anymore. Instead, we consider the generation of additional local constraints with respect to cost considerations.

We will see that, in the presence of a near-optimal solution to the CNDP with associated objective value B, in each Lagrangian sub-problem we can infer restrictions on the number of arcs that need to be installed in any improving solution. Before we can state the idea more formally, to ease the notation, we introduce some identifiers. For the shortest path relaxation, we define

$$\begin{aligned} \hat{f}_{ij} &= f_{ij} + \lambda_{ij} u_{ij} + \sum_l \nu^l_{ij} b^l_{ij} \qquad \forall\,(i,j) \in E && (7.7)\\ \hat{B} &= B - \sum_l L^l_{SP}(\lambda, \nu) && (7.8) \end{aligned}$$

for Lagrangian multipliers $\lambda, \nu \leq 0$. Likewise, for the knapsack relaxation, we set

$$\begin{aligned} \hat{f}_{ij} &= f_{ij} + L^{(i,j)}_{KP}(\mu) \qquad \forall\,(i,j) \in E && (7.9)\\ \hat{B} &= B + \mu^T d && (7.10) \end{aligned}$$

for Lagrangian multipliers μ. Furthermore, denote the current Lagrangian sub-problem $L_{SP}(\lambda,\nu)$ or $L_{KP}(\mu)$ by L_R.

Theorem 7.3 *Denote an ordering of the arcs in E by $e_1,\dots,e_{|E|}$ such that $i<j$ implies $\hat{f}_{e_i}\le\hat{f}_{e_j}$. It holds: if $L_R<B$, then*

a) there exist values

$$\begin{aligned} F &= \min\{0\le u\le|E| \mid \sum_{h\le u}\hat{f}_{e_h}<\hat{B}\} \text{ and} && (7.11)\\ U &= \max\{0\le u\le|E| \mid \sum_{h\le u}\hat{f}_{e_h}<\hat{B}\}. && (7.12)\end{aligned}$$

b) In any improving solution (x,y), it holds that

$$F\le\sum_{(i,j)\in E}y_{ij}\le U. \tag{7.13}$$

Proof:

a) It is sufficient to show that there exists a value $0\le l\le|E|$ such that $\sum_{h\le l}\hat{f}_{e_h}<\hat{B}$. Let $l=\operatorname{argmin}_{0\le h\le|E|}\{\sum_{h\le l}\hat{f}_{e_h}\}$. It holds that:

$$\sum_{h\le l}\hat{f}_{e_h}=L_{SP}(\lambda,\nu)-\sum_{h\le K}L^h_{SP}(\lambda,\nu)<B-\sum_{h\le K}L^h_{SP}(\lambda,\nu)=\hat{B}. \tag{7.14}$$

Or, respectively,

$$\sum_{h\le l}\hat{f}_{e_h}=L_{KP}(\mu)+\mu^Td<B+\mu^Td=\hat{B}. \tag{7.15}$$

b) Let $L^>[U]:=L[\sum_{(i,j)\in E}y_{ij}>U]$ denote the optimization problem that evolves by adding the constraint $\sum_{(i,j)\in E}y_{ij}>U$ to a problem L. By the definition of U, we know that $L^>_R[U]>B$. We remind the reader that we identify the name of an optimization problem with its optimal value. Now, assume that there exists a feasible solution (x,y) for L_{CNDP} with $|E|\ge\sum_{(i,j)\in E}y_{ij}>U$. Then it holds that

$$c^Tx+f^Ty\ge L^>_{CNDP}[U]\ge L^>_R[U]>B. \tag{7.16}$$

The case $\sum_{(i,j)\in E}y_{ij}<F$ follows analogously. ■

Theorem 7.3 allows us to add cardinality cuts on the number of arcs to be installed without loosing improving solutions. We will evaluate the effect of local cardinality cuts on the solution process in Section 7.4.

7.3 A Branch & Bound Algorithm

After having described the bound computation and possible reduction strategies based on Lagrangian relaxation, now we sketch the decisions taken in the tree search.

Dominance Cut-Off Rule Apart from the lower bound exceeding the upper bound, the search in the current node can be pruned if the min-cost routing of all commodities only uses arcs that have already been decided to be installed. Thus, in every choice point we use a column generation approach to solve the Min-Cost Multicommodity Flow Problem on the subset of arcs with associated y_{ij} that have a current upper bound of 1. If that routing only uses arcs (i, j) with y_{ij} with lower bound 1, we can prune the search and backtrack.

Branching Variable Selection The previous discussion also induces a rule for the selection of the branching variable: it is clearly favorable to choose a variable for branching that is being used by the current optimal min-cost multicommodity flow. Of course, there may be more than just one such variable. Then we can choose the one with minimal or maximal reduced costs $|\hat{f}_{ij}|$ in the Lagrangian sub-problem with the best associated multipliers. The different choices will be evaluated in Section 7.4.

Tree Traversal A simple depth first search procedure is used to choose the next search node. This allows to find feasible solutions quickly and eases the reuse of Lagrangian multipliers.

Primal Heuristic To find reasonably good and near-optimal solutions quickly, in every search node we apply a Lagrangian heuristic that was suggested by Holmberg and Yuan. It works by computing multicommodity flow solutions on a subset of the arcs and de-assigning all arcs that carry no flow. For further details, we refer the reader to [112].

Variable Fixing Heuristic Since the Capacitated Network Design Problem is very hard to be solved exactly, we may decide to search for relatively good solutions quickly. The exact approach can be transformed into a heuristic for the problem by fixing variables more optimistically. Holmberg and Yuan [112] developed the so-called α-heuristic for this purpose: While solving the Lagrangian dual, we protocol how often a variable is set to zero or one. If one of the values is dominant with respect to a given parameter, the variable is simply set to this value.

7.4 Numerical Results

We report on our computational experience with the previously developed algorithms. The section is structured as follows: first, we introduce the benchmark data used in the experiments. Then we define the possible parameter settings that activate and deactivate different algorithmic components. Finally, we compare the variants when solving the CNDP from scratch, in the optimality proof, and when using the approach as a heuristic.

All tests were carried out on systems with AMD Athlon, 600 MHz processors, and 256 MB main memory. The code was compiled with the GNU g++ 2.95 compiler using optimization level O3.

7.4.1 Benchmark Data

Surprisingly, in spite of the theoretical interest the CNDP has drawn and the large number of research groups that have dealt with the problem, apparently there has been no benchmark set established on which researchers can compare algorithms that solve the CNDP exactly. Much work has been done with respect to the computation of lower bounds and the heuristic solution of the problem. Benchmarks used for this purpose (to be found in [44, 112], for example) are still too large to allow the computation of optimal solutions. For variations of the problem (such as the Multi-Arc CNDP, Network Loading, etc.), benchmark data exists, but it is not straightforward to see how it could be converted into meaningful instances for the pure CNDP as we consider it here.

Thus, we decided to base our comparison on a benchmark of 48 instances generated by a CNDP generator developed by Crainic et al. and described in [44]. The generator is used by different research groups after it was enhanced with a stable random number generator by A. Frangioni. We generated graphs with 12, 18, and 24 nodes with 50 to 440 arcs and 50 to 160 commodities. For the heuristic comparison we use the benchmark set from [44, 45].

7.4.2 Algorithm Variants Considered in the Experiments

The optimization system developed consists of several parts. We list the components that are compared and evaluated in the experiments: different Lagrangian relaxation algorithms based on the shortest path (SP) or the knapsack relaxation (KP), respectively; a branch & bound algorithm using bounds based on those relaxations, where the branching variable is chosen according to minimal (BR0) or maximal (BR1) absolute reduced-cost values $\hat{f}_{ij}$; two different filtering algorithms based on the shortest path relaxation (SF) and the knapsack relaxation (KF) that fix variables statically (STAT) after optimal Lagrangian multipliers have been computed, or dynamically (DYN) during the optimization of the Lagrangian dual; and finally, the cardinality interval tightening algorithm that adds Lagrangian cardinality cuts to the problem (CIT).

7.4.3 Evaluation

With the first experiments we performed we wanted to find out which type of Lagrangian relaxation was preferable. In accordance to the results reported in [112], we found that the knapsack relaxation is clearly superior both with respect to the number of subgradient iterations needed to solve the Lagrangian dual as well as the time needed to solve the Lagrangian sub-problems. Therefore, we start right away with an evaluation of the impact of Lagrangian cardinality cuts when solving the CNDP using the knapsack relaxation. Table 7.1 shows a comparison of lower bound routines using the Lagrangian knapsack relaxation with and without cardinality cuts. Table 7.2 shows a comparison of two different strategies for the selection of the branching variable. Comparing two variants, the tables give the average percentage of the second variant when compared to the first (that is always set to 100%) with respect to running times and the number of search nodes visited in the branch & bound trees. Moreover, we specify minima, maxima, and the variance of those percentages.

	BR0 $\rightarrow$ BR0-CIT	BR1 $\rightarrow$ BR1-CIT
time	**93.7%**	**25.3%**
min	4.72%	0.28%
max	353%	131.5%
variance	62.1%	11.5%
nodes	**38.6%**	**10.1%**
min	0.73%	0.02%
max	120.7%	78.4%
variance	14.5%	2.9%

Tab. 7.1: The impact of cardinality interval tightening when using the knapsack relaxation for pruning and problem reduction. Mean, minimum, maximum, and variance of running time and number of nodes in the branch-and-bound tree are given.

Clearly, choosing a branching variable with minimal reduced costs is favorable, no matter if cardinality cuts are introduced or not. This contradicts the recommendation given in [112]. Actually, this result is not very surprising. Intuitively, the variable with the minimal absolute reduced costs is least likely to be set by variable fixing. It is the variable we have the least knowledge about, and therefore it is a good choice to base a case distinction on it. In contrast, the variable with the largest absolute reduced costs is most likely to be set by variable fixing, and therefore it is no good idea to double the effort by using this variable for branching.

	BR0 $\rightarrow$ BR1	BR0-CIT $\rightarrow$ BR1-CIT
time	**1817.7%**	**235.1%**
min	68.03%	30.91%
max	7445.5%	1221.7%
variance	37944.6%	604.7%
nodes	**2750.4%**	**311.1%**
min	89.832%	15.636%
max	19415.3%	1427%
variance	172454%	1163.3%

Tab. 7.2: The impact of the branching variable selection when pruning and fi ltering is done with the help of the knapsack relaxation.

Regarding the introduction of Lagrangian cardinality cuts, Table 7.1 shows that they have a great impact on the number of search nodes that have to be investigated. Cardinality cuts are also favorable with respect to the total running time, but the gains are not as large as with respect to the size of the search tree. The trade-off is caused by the additional effort that is necessary to sort the arcs with respect to the current reduced costs $\hat{f}_{ij}$.

When looking at the data more precisely, we find that the primal heuristic works much better in the presence of cardinality cuts. The result of this positive effect is clear: high quality upper bounds are found much earlier in the search, pruning and variable fixing work much better, and the number of search nodes is greatly reduced, which explains the numbers in Table 7.1.

We conjecture that the primal heuristic works so well in the presence of cardinality cuts, because they provide a good estimate on the number of arcs that need to be installed in order to improve the current solution. Thus, the right amount of arcs is opened for the heuristic, and it is able to compute near-optimal solutions at a higher rate.

Next, we evaluate the use of CP-based Lagrangian relaxation for the CNDP. Table 7.3 shows a comparison of runs when using shortest path variable fixing in addition to the knapsack variable fixing algorithm. The results are very disappointing: not only is the integrated approach inferior with respect to the total running time. On top of that, the reduction of choice points is negligible, and therefore the additional effort taken is almost worthless.

	SOLVE: KF $\to$ KF-SF	OPT: KF $\to$ KF-SF
time	**148.6%**	**144.1%**
min	96.59%	51.87%
max	466%	271.3%
variance	46.1%	13.5%
nodes	**133.8%**	**94.9%**
min	71.42%	20%
max	677.1%	180.3%
variance	166.3%	7.5%

Tab. 7.3: The impact of additional shortest-path filtering when using the knapsack relaxation for pruning and problem reduction. Branching strategy BR0 and cardinality interval tightening are used.

Note that, when using the linking method, the number of search nodes sometimes even exceeds the value that is achieved when using knapsack variable fixing only. This is caused by differences when building up the search tree: the Lagrangian dual usually stops with different Lagrangian multipliers that have a severe impact on the variable selection. Moreover, the generation of primal solutions differs, which makes the comparison particularly difficult, because variable fixing is highly sensitive to the quality of upper bounds. Thus, to eliminate the last perturbation, we repeated the experiment on the algorithmic optimality proof. That is, we provide the algorithm with an optimal solution and let it prove its optimality only.

Table 7.3 shows the results that still reveal the poor performance of the additional application of shortest path variable fixing. The reason for this is that the shortest path variable fixing algorithm is much less effective than the one based on the knapsack relaxation: Recall from Section 7.2.4 that the shortest path relaxation value consists of two values, one for the shortest-path routings, and the other one for the design variables. However, the variable fixing algorithm only incorporates the latter costs and does not take into account that the removal of arcs may generally increase the routing costs as well. Therefore, using shortest path variable fixing as described in Section 7.2.4 is comparably ineffective and does not pay off.

We tried to improve the effectiveness of the algorithm by adding node-capacity constraints. If a node is a source for some commodities, its out-capacity must be large enough to push the corresponding supply into the network. Similarly, if a node is a sink node for some commodities, its in-capacity must be large enough to let the required demand in. In contrast to the knapsack relaxation, where the x- and y-variables are not independent, the shortest path relaxation allows to incorporate those constraints very easily. Moreover, we tried to improve the shortest path reduction algorithm further by computing a lower bound on the additional costs of routing a given commodity when a certain arc is not installed. The algorithm presented in Section 2.2.4.1 was implemented for this purpose. However, all these efforts did not result in a filtering algorithm that was effective enough to be worth applying. Therefore, we cannot recommend to use the shortest path relaxation, neither for the computation of Lagrangian relaxation bounds, nor for problem reduction. Note that this recommendation stands in contradiction to the one given in [44].

In Figure 7.1, we compare the strategy to fix variables during the optimization of the Lagrangian dual with the fixing of variables after optimal multipliers have been found only. We see that filtering "on the fly" is slightly favorable. Our experience also shows that the subgradient method is very robust with respect to problem reduction, and we can afford not to reset the step length after variables have been fixed in the Lagrangian sub-problem without loosing convergence in practice.

	DYN $\rightarrow$ STAT
time	**115.9%**
min	36.95%
max	229.2%
variance	14.1%
nodes	**123.2%**
min	29.72%
max	233.8%
variance	20.8%

Fig. 7.1: Optimality proofs: comparison of two strategies when using reduction based on knapsack relaxation: on-the-fly fixing vs. fixing after Lagrange.

Next, in Table 7.4, we compare the performance of the algorithm we developed and the standard solver ILOG CPLEX 7.5 [118] when solving the CNDP and when proving the optimality of a given solution. Clearly, using LP-bounds improved by several kinds of cuts that CPLEX adds to the problem results in a huge reduction of search nodes. However, Lagrangian relaxation allows to compute lower bounds much faster, so that the approach

	OPT: CPLEX → KP-BR0-KF-CIT	SOLVE: CPLEX → KP-BR0-KF-CIT
time	**73.5%**	**229.2%**
min	9.63%	22.48%
max	259%	753.5%
variance	36.5%	356.5%
nodes	**1148.1%**	**3014.6%**
min	196.666%	100%
max	5250%	10279.5%
variance	10762.4%	73762.5%

Tab. 7.4: Comparison of the CPLEX branch-and-cut algorithm and Lagrangian relaxation (pruning and reduction based on the knapsack relaxation plus cardinality interval tightening).

presented here is still competitive when solving the CNDP. It even achieves a considerable improvement upon the running time in the optimality proof. Most important, however, is the fact that the approach we developed is able to tackle much larger instances using the variable fixing heuristic presented in Section 7.3.

We compare the non-exact version of our approach (using the α-fixing heuristic) with other heuristic approaches [45, 94, 95] that have been developed for the CNDP. A comparison with CPLEX is left out because it is not at all competitive for this benchmark set containing larger CNDP instances. In Figure 7.2, we give the percentage of instances in a benchmark set (set C in [44, 45], containing 31 instances) that have been solved within a given solution quality (in percent, compared with the best known solution). Not only are the α-fixing with and without cardinality cuts clearly superior with respect to the achieved solution quality. On top of that, the heuristic variable fixing approach was stopped after at most 300 seconds CPU time. On this benchmark set, heuristic variable fixing is on average about 6 times faster than TABU-PATH and 23 times faster than PATH-RELINKING (using SPECint values to make different architectures comparable).

7.5 Summary and Future Work

We have presented an approach for the solution of the Capacitated Network Design Problem. It is based on a tree search where lower bounds based on Lagrangian relaxation are used for pruning. Two kinds of relaxation are considered, the shortest path and the knapsack relaxation. The latter is clearly favorable with respect to the convergence of the subgradient algorithm that optimizes the Lagrangian dual.

Two different variable fixing algorithms have been proposed in the literature that belong to

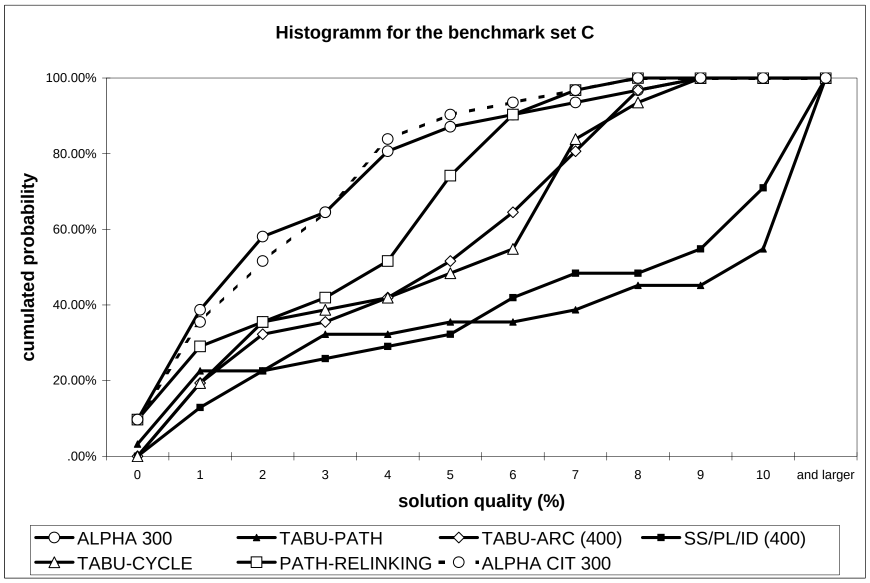


Fig. 7.2: Comparison of different heuristic solvers for the CNDP.

the kind of relaxation that is chosen. When using the knapsack relaxation, we have shown how variables can also be fixed with respect to shortest-path considerations by using dual values in the Lagrangian knapsack sub-problem. However, even in a stronger version and in combination with node-capacity constraints, the shortest path variable fixing algorithm is too ineffective to justify the additional effort that is necessary for its application.

To tighten the problem formulation in a search node, we introduced the idea of local Lagrangian cardinality cuts. Experimental results show that their application improves the overall running time, even though the time per search node increases considerably when they are applied.

Finally, we compared the heuristic variable fixing approach with other heuristic approaches developed for the CNDP. The results show that the tree-search approach that we implemented clearly outperforms other heuristics both with respect to the CPU time needed and the solution quality that is achieved.

Regarding the fact that we set up our system for the evaluation of variable fixing algorithms and local Lagrangian cardinality cuts, and taking into account that no sophisticated methods (like bundle methods for example) for the optimization of the Lagrangian dual are used, we consider these results as very encouraging. Most importantly, note that no global cuts are introduced yet to strengthen the lower bounds computed. With the help of additional cuts that provably strengthen the LP relaxation (see [33] for example), we expect that the performance of the approach presented can be further improved.

Chapter 8

The Social Golfer Problem

In Chapter 4, we introduced the Social Golfer Problem as an example for a highly symmetric constraint satisfaction problem. We revise the symmetry detection function for the symmetry breaking method that was presented in Chapter 4. Then, with the help of heuristic constraint propagation, we develop a state-of-the-art algorithm for the Social Golfer Problem.

The Social Golfer Problem has attracted much interest in recent years. That interest is mainly caused by its highly symmetric structure, that has let it become a favorite playground for research on the systematic breaking of symmetries.

In [202], Barbara Smith applied an approach that breaks symmetries during the search, i.e., she uses SBDS [93] to tackle the Social Golfer Problem. In combination with careful model selection she was able to efficiently break most of the symmetries, but still found non-unique solutions for the instances studied. Note that work is in progress which removes SBDS's need for an explicit list of symmetries [92, 152]; eventually this should allow SBDS to be used to eliminate all symmetries from the problem.

In [81], Filippo Focacci and Michela Milano presented another generic method for breaking symmetries, based on *global cut seeds*, generating *symmetry removal cuts*. With this approach it should be possible to eliminate all the symmetries of the Social Golfer Problem, but at the time of writing this has not been done.

The very similar SBDD method that we presented in Chapter 4 was developed independently. It is based on the detection of dominance relations between choice points and works particularly well for highly symmetric problems. At the time of writing, this is the only technique which has been used to completely eliminate all symmetries from non-trivial instances of the Social Golfer Problem.

In [104], Warwick Harvey compares SBDS and SBDD, and also gives some numerical results on the Social Golfer Problem.

The approach that we present in the following was the best one known for the Social Golfer Problem by the time of publication. Meanwhile, our results have been assimilated and extended by several other research groups.

In [16], Nicolas Barnier and Pascal Brisset developed an approach for the Social Golfer Problem that extends the concept of SBDD by incorporating the branching variable selection. In [174], Francois Puget also refines symmetry breaking for the Social Golfer Problem.

The work presented in this chapter was published in [191, 192]. It is structured as follows: To keep the chapter self-contained, we review the definition of the Social Golfer Problem in Section 8.1. Then, in Section 8.2, we present a refined symmetry breaking function. Incorporating heuristic constraint propagation of some redundant constraints that are introduced in Section 8.3, we present numerical results of our approach in Section 8.4.

8.1 Definition

Given natural numbers $w, g, s \in \mathbb{N}$, the *Social Golfer Problem* consists in finding w partitionings of the set $\{1, \ldots, gs\}$ into g sets of size s such that no two such sets have more than one member in common. More formally, the problem is to compute wg sets $X_{1,1}, \ldots, X_{1,g}, X_{2,1}, \ldots, X_{w,g} \subset \{1, \ldots, gs\}$ such that

- $|X_{i,k}| = s$ for all $1 \leq i \leq w$, $1 \leq k \leq g$,
- $\bigcup_{1 \leq k \leq g} X_{i,k} = \{1, \ldots, gs\}$ for all $1 \leq i \leq w$, and
- $|X_{i,k} \cap X_{j,l}| \leq 1$ for all $(i,k) \neq (j,l)$.

An instance of the Social Golfer Problem is written as a triple g-s-w from now on. When $(s-1)w = gs - 1$, we have a configuration where every player must play with every other exactly once. This corresponds to a *resolvable Balanced Incomplete Block Design*. Perhaps the most well-known of these is *Kirkman's Schoolgirl Problem*, posed (and solved) by Thomas Kirkman in 1850. This instance, which is equivalent to the golfer 5-3-7 problem, was stated as follows:

> *How can 15 schoolgirls walk in 5 rows of 3 each for 7 days so that no girl walks with any other girl in the same triplet more than once?*

To the best of our knowledge, the computational complexity of the Social Golfer Problem is not known yet. In the combinatorial design area, solutions for $s = 3$ are known as Kirkman Triple Systems or Resolvable Steiner Systems [41]. It can be shown that an instance x-x-4 is equivalent to finding two orthogonal latin squares of size x. Even more so, an instance x-x-y is equivalent to finding a set of $y-2$ mutually orthogonal latin squares, a problem that has been studied for many years now [42].

Despite its apparently simple definition, computational approaches have great difficulties solving even small instances of the Social Golfer Problem in a reasonable amount of time. In our view, there are two main aspects to the problem that cause its big complexity for enumeration approaches:

- The Social Golfer Problem is highly symmetric.
- The clique structure of the constraints ensuring that any two golfers do not play together more than once makes it hard to judge the feasibility of a partial assignment.

In the following, we will address these two points by introducing a refined symmetry detection function for SBDD and the idea of heuristic constraint propagation.

8.2 Another SBDD-Approach for the Social Golfer Problem

The Social Golfer Problem contains a remarkable number of symmetries. Players can be placed at any position within a group, groups can be rearranged within their week, and the weeks can be ordered arbitrarily. Moreover, the player names can be permuted in any way desired. To give an example: even the best models (in terms of symmetry reduction) for the original schoolgirl instance still contain more than 10^{12} symmetries.

We have chosen to apply symmetry breaking during search (SBDD) in combination with a straightforward model for the Social Golfer Problem that can be implemented with very little effort using the ILOG SOLVER environment [121]. The groups are modeled as sets of players with the cardinality of each set fixed to s. Each week contains g such sets, and the full pattern covers w weeks. Initially, we fix all the players in the first week in increasing order. Additionally, we insert the first s players into the first s groups for all weeks thereafter. Finally, the first group of the second week can be filled with the smallest indexed players possible. None of these initial labelings exclude any unique solutions.

We build up the schedule week by week, choosing as branching variable the group with the smallest domain or as branching value the player with the fewest possible groups he or she can be assigned to, depending on what leaves us with fewer choices.

To apply SBDD, we need to define a *symmetry detection function* φ that, for two given choice points c_1 and c_2 represented as *patterns* reflecting the branching decisions taken so far, returns *true* if and only if there exists a symmetry showing that c_1 defines a sub-tree of c_2 under that symmetry. Then, at every choice point we check whether it is dominated in this fashion by some previously expanded choice point, and if so, we prune the search.

In Section 4.3, to find a symmetry that proves a dominance relation between choice points, we presented a procedure that, when given a specific player permutation, checks whether there exists a week permutation that shows that one pattern dominates the other. To remove all symmetries, it was then necessary to iterate over all player permutations, a test that turned out to be extremely expensive. Now, instead of iterating over all player permutations and computing week permutations, we suggest to proceed vice versa: given a permutation of the weeks, we check whether there exists a permutation of the players such that one pattern dominates the other. That is, we iterate over all week permutations in c_2 and search for a player permutation that is feasible with respect to the currently required matching of the weeks. Thus, again we set up a nested constraint satisfaction problem to find a suitable symmetry or prove that none exists.

As this full dominance check is still very expensive for the Social Golfer Problem, we only perform it when a week is being filled completely. This idea is motivated by the experiments that we presented earlier in Section 4.3. There, the application of a full dominance check turned out to be most efficient when being applied in selected levels of the search tree only. For the remaining nodes, we fix the player permutation to the identity and apply $\Phi_{W,G}$ (see Section 4.3.2), that is, we search for symmetries according to feasible orderings of the weeks, the groups and the players within the groups. This less expensive check is implemented as a pairwise dominance check between weeks, followed by the computation of a maximum cardinality matching on a bipartite graph $(V_1 \cup V_2, E)$ where the weeks in c_1 and c_2 define the nodes in V_1 and V_2, respectively, and $(v_i^1, v_j^2) \in E$ iff week i in c_1 is dominated by week j in c_2.

8.3 Heuristic Constraint Propagation

Having introduced the Social Golfer Problem, and having presented an efficient way of handling the many symmetries in the problem, we now introduce a new idea, the heuristic propagation of additional redundant constraints by means of local search.

Assume we are given an NP-hard constraint satisfaction problem. Even though there is no proof that we cannot solve the problem efficiently, there is strong empirical evidence that we cannot compute a solution in polynomial time. The common approach then is to explore the search space in some sophisticated manner that tries to consider huge parts implicitly. For constraint satisfaction problems, that means that we try to cut off preferably large regions that do not contain any feasible solutions. In constraint programming, particularly when performing some kind of tree search, domain filtering algorithms are used for that purpose. Basically, the model and the degree of propagation determine how the work is partitioned between the choice points and the search tree as a whole. That is, we can aim at reducing the number of choice points by spending additional effort in each of the choice points, or we can choose to keep the work done per choice point small, resulting in a bigger search tree.

Thus, we face a trade-off between the time spent per choice point and the total number of choice points. To take the alternatives to extremes, on one hand we can explore the entire domain space, and on the other hand we can compute a solution or prove that none exists in the first node visited, without making any choices which might need to be backtracked. For most applications, the optimal balance will lie somewhere between the two extremes.

If we find that we expand too many choice points we may want to give more burden to the individual choice points. Revising the model by adding redundant constraints is a common way to achieve that goal. In general, we expect the redundant constraints to detect inconsistencies higher up in the search tree. However, since checking whether a given partial assignment is extensible to a full solution is usually of the same computational complexity as the original problem, redundant constraints typically still only enforce a relaxation of the actual problem.

We propose adding tight redundant constraints that may be hard to verify exactly, but that can be checked by applying some heuristic. That is, when formulating additional constraints, we do not wish to restrict ourselves to considering only those constraints which are (relatively) easy to propagate completely. Instead, we perform an incomplete check of complex redundant constraints using the rich heuristic machinery that was developed in the operations research community.

8.3.1 Literature on the Integration of CP and Local Search

In recent years, a substantial number of different approaches to the integration of constraint programming (CP) and local search (LS) have been developed. The main problem when constructing CP-LS hybrids is caused by the fact that CP uses monotonic reasoning whereas LS does not.

Fairly balanced hybrids result from sequential applications of the two methods [36, 173]. Other balanced hybrids can also be achieved by applying decomposition methods (like Lagrangian relaxation, column generation or Benders decomposition), where sub- and master problem can be solved by different solution methods.

On the other hand, there also exist many developments that favor one of the methods and just use the other one to overcome certain weaknesses. Constrained local search, for example, uses LS as the predominant approach, with CP used to find neighbors in a sparse and/or large neighborhood [5, 165].

Focusing on constraint programming instead has yielded hybrids that use local search to adapt the variable and/or value ordering in the search tree. Only recently, an approach was presented that uses local search to find dominating partial assignments that prove the sub-optimality of the current search node, information that can be used for pruning [82]. For a more complete overview on the field we refer the reader to the recent tutorial by Focacci et al. [76].

As with other methods, constraint programming forms the basis of our approach. However, our use of local search is quite different to any of the methods mentioned above. For a given model for a problem, we suggest considering the addition of complex redundant constraints, and then using local search to perform (incomplete) propagation of these constraints.

The idea of using a stochastic search method to prove unsatisfiability is not new; it was Challenge 5 in [196]. Also, heuristic search has been used when tackling certain optimization problems like maximum clique or graph coloring [15, 60]. However, it appears that the idea has never been introduced systematically. In the few examples where they are used, complex redundant constraints are only used for pruning, but not for domain filtering. To the best of our knowledge, the work presented here is the first to do it, albeit on a fairly specific set of constraints that comprise sub-problems of the real problem we are trying to solve.

We already mentioned that the constraints requiring that every golfer must not play with any other golfer more than once makes it very hard to judge the extensibility of a partial assignment. That is, the sub-tree rooted by the current choice point may not contain any feasible solution, but due to the fact that the different constraints in the problem are propagated independently from each other and only interact via domain reductions, we cannot detect infeasibilities high up in the search tree. As a matter of fact, when using the model we described above, many searches will only backtrack when the assignments for an entire week are almost complete or even after having started to do assignments in the last week only.

For most constraint satisfaction problems, to check whether a partial assignment can still be extended to a feasible solution is of the same computational complexity as the original problem itself. Therefore, in general the pruning and filtering algorithm applied in the search nodes cannot be expected to be exact. Rather, it must be looked at as a heuristic to tighten the problem formulation and to shrink the search space. Of course, there is a trade-off between the time needed to apply that heuristic and the time needed to explore the remaining sub-tree.

Now, to improve the situation for the Social Golfer Problem as well as for many other constraint satisfaction problems, we can try to formulate necessary constraints for partial assignments to be extensible to complete, feasible solutions. For the Social Golfer Problem (and, we believe, for many other problems as well), we are left with the decision to choose a weak redundant constraint that can be propagated efficiently, or to pick a condition that is more accurate but maybe much harder to verify. For the latter, we may consider applying a heuristic to perform incomplete domain filtering or pruning. Note that since the added constraints are redundant, the incompleteness of this filtering does not affect either the soundness or the completeness of the search; if some opportunity for pruning is missed, it just means that the tree searched may be larger than strictly necessary.

In the following, we describe two different types of additional constraints that we would like to add to our model to be able to detect inconsistencies quickly and as early as possible. The first type of redundant constraint is defined with respect to the possibility of completing the assignments in a given week. Since we represent weeks by rows in our schedule under construction, we use the term *horizontal constraints* to refer to these constraints. Correspondingly, by *vertical constraints* we mean those used to check necessary conditions for a partial assignment to be extensible to a w-week solution.

	group 1			*group 2*			*group 3*			*group 4*			*group 5*		
week 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
week 2	1	4	7	2	5	8	3	*	*	*	*	*	*	*	*

Tab. 8.1: A partial instantiation of the 5-3-2 Social Golfer Problem.

	group 1			*group 2*			*group 3*			*group 4*			*group 5*		
week 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
week 2	1	4	7	2	5	8	3	6	10	11	*	*	12	*	*

Tab. 8.2: A more complete partial instantiation of the 5-3-2 Social Golfer Problem.

8.3.2 Horizontal Constraints

Consider the example in Table 8.1. We are searching for a two-week schedule for 15 golfers, to be arranged in 5 groups of 3 in each week.

For the given partial assignment, suppose we add player 6 to group 3. Next observe that players 10, 11 and 12 must be separated in week 2. As there are only three more groups that have not yet been filled completely, the third player in group three must be one of players 10, 11 and 12. A possible continuation is given in Table 8.2.

Now there are only two groups left that have not been completed yet, but players 13, 14 and 15 must still be separated. Therefore, the current partial assignment is inconsistent, and we can backtrack.

We can generalize this observation. For a given incomplete week, we define a residual graph R that consists of a node for each unassigned player and an edge for each pair of such players that have already been assigned to play together in some other week. An example of such a graph, corresponding to week two from Table 8.1, is shown in Figure 8.1. Then, for the given week we count the number of groups that are not closed yet and compare that number with the size of the biggest clique in R. If the first number is smaller than the latter, then there is no way to extend the current assignment to the rest of the week, and the assignment is inconsistent.

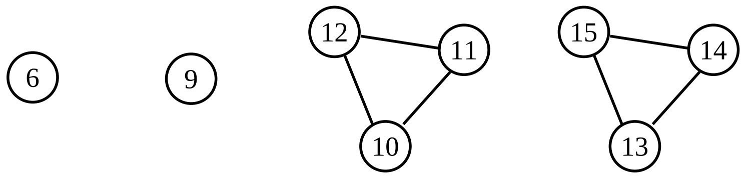


Fig. 8.1: The residual graph of week 2 from Table 8.1.

In this way we can define a sufficient condition for a witness which proves that the current partial assignment is inconsistent: a clique exceeding a certain cardinality. If we can find such a witness, we can backtrack immediately. Finding a clique of size k is known to be NP-hard for arbitrary graphs, and while the residual graphs we are dealing with have special structure that may allow the efficient computation of such a clique, we chose not to try to find a polynomial-time complete method. Instead we apply a heuristic search to find a sufficiently large clique, an approach which, as we shall see, has advantages over one which simply returns the largest possible clique.

Reconsider the situation given in Table 8.1. We can add neither player 6 nor player 9 to group 3 for the same reason: the members of groups 4 and 5 of week 1 must be separated, and to do so we require the two open positions in group 3 of week 2.

When checking the redundant constraint described above, assume we have set up the residual graph and suppose the heuristic we apply finds the two disjoint cliques of size three ($\{10,11,12\}$ and $\{13,14,15\}$). Since the sizes of these cliques are equal to the number of incomplete groups, we have not found any witnesses showing that the current partial assignment cannot be extended to a full schedule — indeed, the schedule can still be completed. However, since group 3 has only two open positions left, we can conclude that group 3 must be a subset of $\{3\} \cup \{10,11,12\} \cup \{13,14,15\}$. That is, we can use heuristic information for domain filtering.

We conclude that finding a witness for unsatisfiability is a rather complex task, but it can be looked for by applying a heuristic. Moreover, even if we do not find such a witness, we may find other “good” witnesses (namely some fairly large cliques) and their information can be combined and used for domain filtering.

Therefore, it is advantageous to use a heuristic that not only provides us with good solutions quickly, but that also gives us several solutions achieving almost optimal objective values. Local search heuristics seem perfectly suited for this purpose.

8.3.2.1 Heuristic Clique Search

To find large cliques in the residual graph, we perform a randomized local search that works in the following way: We initialize our current clique C with a random node and set $C_{\text{best}} \leftarrow C$. Next we intensify C by repeatedly searching for a random node that is adjacent to all nodes in C. If no such node exists anymore, we compare the cardinalities $|C|$ and $|C_{\text{best}}|$ and update C_{best} if necessary. We then move on with a diversification step by adding a random node $v \in V \setminus C$ to C and removing all nodes in C that are not adjacent to v. These nodes shall not be considered in the next diversification step. Now, the loop is complete and we return to the intensification phase. The process stops after having found a clique that exceeds the crucial cardinality or after a given iteration limit.

	group 1				*group 2*				*group 3*				*group 4*				*group 5*			
week 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
week 2	1	5	9	13	2	6	10	*	3	7	*	*	4	8	*	*	*	*	*	*

Tab. 8.3: A partial instantiation of the 5-4-2 Social Golfer Problem.

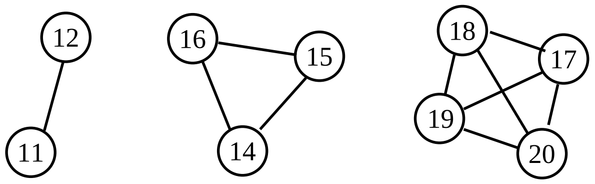


Fig. 8.2: The residual graph of week 2 from Table 8.3.

Obviously, the approach as sketched above produces a sequence of cliques that we can use for pruning and domain filtering. We do not claim that the clique search procedure we use is very sophisticated for finding maximum cardinality cliques in a given graph. In fact, many other heuristic and exact approaches can be thought of, and there has certainly been a lot of relevant research done. However, we did not aim at developing a special method that produces one clique of large cardinality, but rather that finds a large number of fairly big cliques. In any event, while the algorithm could no doubt be improved on, for the graphs we are dealing with it works well enough to prove the point.

Before we continue by presenting another type of redundant constraint developed for the Social Golfer Problem, we give a more complete example of how horizontal constraints can be used for pruning and domain filtering.

Consider the partial assignment for the social golfer instance 5-4-2 given in Table 8.3. The associated residual graph for week 2 is given in Figure 8.2.

The local search procedure that we apply returns three disjoint cliques, namely $\{11,12\}$, $\{14,15,16\}$, and $\{17,18,19,20\}$. Since there are still four groups that have not been filled up yet and the largest clique is of size four, we cannot prune right away. However, we do know that the final element in group 2 must come from the clique of size four, and so we can shrink the set of possible elements appearing in this group to $\{2,6,10,17,18,19,20\}$. This leaves just three open groups left, and we have a remaining clique of size 3 at hand. Again we cannot prune here, but we know that the remaining two elements of groups 3 and 4 must come from the cliques of size three and four, so we can shrink the set of possible elements of those groups to be $\{3,7,14,15,16,17,18,19,20\}$ and $\{4,8,14,15,16,17,18,19,20\}$, respectively. Now there is only one open group left, but we still have to separate players 11 and 12; as a result, the current assignment is inconsistent, and we can backtrack.

	group 1			*group 2*			*group 3*			*group 4*			*group 5*		
week 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
week 2	1	6	7	2	5	10	3	4	13	8	11	14	9	12	15
week 3	1	5	14	2	4	12	3	7	11	9	10	13	6	8	15
week 4	1	4	*	2	*	*	3	5	*	*	*	*	*	*	15
week 5	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
week 6	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
week 7	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

Tab. 8.4: A partial instantiation of the 5-3-7 Social Golfer Problem.

The example shows that it can even be advantageous to have several cliques at hand rather than one big clique only: all cliques together allowed us to prune the search at the current choice point. If we had known the largest clique of size four only, though, we would have had to expand the sub-tree below.

8.3.3 Vertical Constraints

Horizontal constraints are very helpful for judging the extensibility of the week currently under construction. However, they do not help much in getting a clearer view of whether a partial assignment can still be extended to a full *w*-week solution. Therefore, we added another type of redundant constraint to our model, so-called vertical constraints.

Again, we start our discussion with an example (see Table 8.4). In the current partial assignment, week 4 can still be completed consistently. Is there still a continuation of the given schedule to a full 7-week solution, though?

Looking at player 15 (let us assume she is female), we find that she has played with all players in $\{6,8,9,12,13,14\}$ already. As there are 4 weeks left that she has not yet been assigned partners for, she must still play with all players in $\{1,2,3,4,5,7,10,11\}$. To be able to do so, there must be four independent pairs of players in this set that still have not been assigned to play with each other. To check this, we define a residual graph again (this time for each player, in contrast to each week for horizontal constraints) that consists of one node for each player that player 15 has not been assigned to play with yet, and an edge between all such players that have already played with each other (see Figure 8.3).

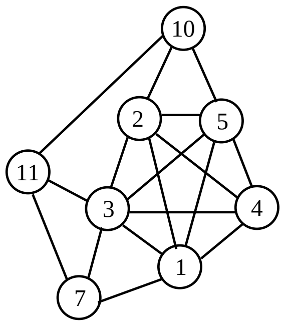


Fig. 8.3: The residual graph of player 15 from Table 8.4.

	group 1			*group 2*			*group 3*			*group 4*		
week 1	1	2	3	4	5	6	7	8	9	10	11	12
week 2	1	4	7	2	8	10	3	5	11	6	9	12
week 3	1	8	11	2	4	9	3	6	10	5	7	12
week 4	*	*	*	*	*	*	*	*	*	*	*	*
week 5	*	*	*	*	*	*	*	*	*	*	*	*

Tab. 8.5: A partial instantiation of the 4-3-5 Social Golfer Problem.

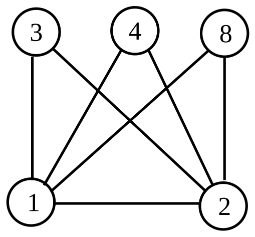


Fig. 8.4: The residual graph of player 12 from Table 8.5.

When trying to find four disjoint stable sets (the term *independent set* is also used in the literature) of size two heuristically, we find that we do not succeed. Of course, this does not prove that there is none. That is, for the vertical constraints we need a witness that not enough disjoint stable sets of a given size exist. The given example already gives an idea of what such a witness might look like. Looking at Figure 8.3, we find that the clique $\{1,2,3,4,5\}$ prevents us from finding four disjoint stable sets of size two. That is, a clique that exceeds a certain cardinality is a witness that the current assignment is inconsistent.

To find large cliques in the residual graph of a player, we can apply the local search procedure developed in Section 8.3.2.1. However, we are facing a slight drawback when using cliques as witnesses. For the Schoolgirl Problem, i.e. when we know that every player must play with every other player exactly once, the bound on the maximum cardinality clique in the residual graph can be chosen to be rather tight. However, if we look at the Social Golfer Problem instance 4-3-5 for example, every golfer does not play every other golfer; they play every other golfer except one, and a priori we cannot say which one that is. As a result, if we look for a clique large enough to guarantee that there will be too few disjoint sets of the appropriate size, that condition is stronger than we would like. This means that it does not become satisfied until later in the computation, and we are not able to prune as early as would be desirable.

To see an example of this, consider the partial schedule in Table 8.5. There are five players which player 12 has so far not been assigned to play with, and to complete the schedule we need two disjoint stable sets of size two. To prove that this is not possible with a single clique, we

	group 1	*group 2*	*group 3*	*group 4*
week 1	1 2 3	4 5 6	7 8 9	10 11 12
week 2	1 4 7	2 5 12	3 8 11	6 9 10
week 3	1 6 11	2 4 9	3 7 12	5 8 10
week 4	* * *	* * *	* * *	* * *
week 5	* * *	* * *	* * *	* * *

Tab. 8.6: A partial instantiation of the 4-3-5 Social Golfer Problem.

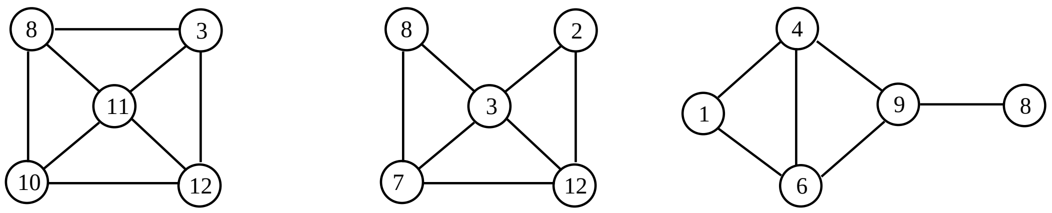


Fig. 8.5: The residual graphs of players 4 (left), 6 (middle) and 12 (right) from Table 8.6.

would need a clique of size four. Looking at the residual graph for player 12 (see Figure 8.4), there are three cliques of size three, namely $\{1,2,3\}$, $\{1,2,4\}$ and $\{1,2,8\}$, but no cliques of size four. However, there are no pairs of disjoint stable sets of size two either, so the schedule cannot be completed but this is not detected. (Note that the residual graphs for the other players all have exactly the same structure.)

As with the horizontal constraints, we can obtain better results by considering more than one clique at once. To illustrate this, consider the slightly different example in Table 8.6. Looking at the residual graph of player 12 (see Figure 8.5), we find two cliques of size three, namely $\{1,4,6\}$ and $\{4,6,9\}$. As before, there are no cliques of size four, but this time there is a pair of disjoint stable sets of size two: $\{1,9\}$ and $\{4,8\}$. So we cannot tell that the schedule cannot be extended simply by looking at this residual graph. However, we can draw inferences about which player is the one that player 12 will not be playing with: it must be an element of the intersection of all cliques of size three. This is because all cliques of size three have to be broken by the removal of a node; otherwise we are guaranteed that there is no way to partition the remaining four nodes into a pair of disjoint stable sets. Thus we can deduce that the player that player 12 must not play with is either player 4 or player 6.

We now look at the residual graphs for player 4 and player 6 (see Figure 8.5). For player 4 we find that the intersection of cliques $\{3,8,11\}$ and $\{10,11,12\}$ requires that player 4 will certainly not play with player 11. Similarly, player 6 must not play with player 3, since, for example, $\{3,7,8\} \cap \{2,3,12\} = \{3\}$. This means both of these players must play with player 12, which contradicts the fact that player 12 must not play with one of them. Hence we can prune the search.

	4-3-3	4-3-4	4-3-5	5-4-3	5-4-4	5-4-5	5-4-6	5-3-6	5-3-7
PI	0.32	0.71	0.98	46.63	83.24	132	143	>5 days	>5 days
	(102)	(182)	(227)	(7430)	(5392)	(6409)	(17129)		
	3.1	3.9	4.3	6.3	15.4	20.6	83.2		
H	0.23	0.43	0.31	40.19	43.2	43.74	33.80	13933	13311
	(76)	(87)	(90)	(6205)	(2829)	(2823)	(2818)	(266140)	(266268)
	3.0	4.9	3.4	6.5	15.3	15.5	12.0	52.4	50.0
V	0.36	0.62	0.47	57.30	90	98.25	46.69	85855	1815
	(100)	(173)	(116)	(7415)	(5282)	(5574)	(3300)	(1075165)	(153697)
	3.6	3.6	4.1	7.7	17.0	17.6	14.1	79.9	11.8
HV	0.25	0.4	0.26	47.78	47.45	44.18	23.21	6771	394
	(76)	(75)	(74)	(6205)	(2750)	(2791)	(1770)	(165238)	(85790)
	3.3	5.3	3.5	7.7	17.3	15.8	13.1	41.0	46.0

Tab. 8.7: The CPU time needed to compute all unique solutions (in seconds), in brackets the number of choice points visited, and the time per choice point (in milliseconds) when computing all unique solutions.

8.4 Numerical Results

To confirm our theoretical discussion, we implemented the model as described in Section 8.2 in C++, compiled by gcc 2.95 with maximal optimization (O3). All experiments were performed on a PC with a INTEL Pentium III/933MHz-processor and 512 MB RAM running Linux 2.4.

Regarding the additional redundant constraints, we present a comparison of four different parameter settings:

1. The plain implementation without redundant constraints (PI),
2. PI plus horizontal constraints only (H),
3. PI plus vertical constraints only (V), and
4. PI plus horizontal and vertical constraints (HV).

In Table 8.7, the variants are evaluated on several social golfer instances. To make the comparison fair and reduce the impact of other choices such as the variable and value orderings used, we compute all unique solutions of an instance (or prove that there are none, for the 4-3-5 and 5-4-6 instances), counting the number of choice points and measuring the CPU time required.

Clearly, using both types of additional constraints results in the biggest reduction of choice points, though for small numbers of weeks the vertical constraints do not give any benefit when horizontal constraints are used (and little benefit even when they are not). It is not surprising

	4-3-3	4-3-4	4-3-5	5-4-3	5-4-4	5-4-5	5-4-6	5-3-6	5-3-7
PI	111	285	358	27057	19804	22295	22673		
	36	115	137	3025	4515	5747	5814		
	28	45	48	31	71	81	80		
H	68	113	114	19264	10589	10690	10652	2013018	2012597
	36	64	58	3018	1805	1934	1870	463737	463984
	35	58	26	35	66	71	60	94	96
V	105	270	123	26970	19280	20209	10896	7969706	724678
	36	110	10	3025	4515	4757	2310	2684251	143631
	22	40	2	25	62	63	35	96	80
HV	68	89	70	19264	10262	10476	5993	1161383	386390
	36	51	0	3018	1805	1785	1074	242709	18424
	48	40	0	30	59	58	23	87	44

Tab. 8.8: The number of simple and complete symmetry checks, and the percentage of time spent in these checks when computing all unique solutions.

that the vertical constraints are of little use when the number of weeks is small, since for such instances the constraints are very weak: there is a lot of flexibility available in deciding which players any given player should play with in the remaining weeks since the number of players they will never play with is quite large. However, as the number of weeks increases, each player must play with a greater number of the other players, so the amount of flexibility is reduced and the constraints become steadily stronger.

A similar trend can be seen in the effectiveness of the horizontal constraints: while they are useful for small numbers of weeks, their effectiveness improves as the number of weeks increases. Again, this is due to the fact that as the number of weeks increases, each player must play with a greater number of the other players, so that there is less flexibility available in deciding who should play together in a given week. This means that it is more likely that a partial week assignment cannot be completed, and hence the horizontal constraints can prune more often.

Whether or not a reduction in the number of choice points also results in a reduction of CPU time is of course determined by the trade-off between the time needed to apply the incomplete propagation algorithms and the time saved by the reduction of choice points. Adding vertical constraints when the number of weeks is small can be expected to worsen the CPU time due to the ineffectiveness of these constraints on these instances, and that is indeed what happens. For (almost) all other instances, the reduction in the number of nodes is sufficiently large to offset the extra cost of the constraints. The only real surprise here is that adding the extra constraints can result in the average amount of time spent at each node *decreasing*, sometimes quite markedly.

	Number of Groups – Number of Players per Group									
Number of Weeks	2-2	3-2	3-3	4-2	4-3	4-4	5-2	5-3	5-4	5-5
2	1	1	1	2	1	1	2	2	1	1
3	1	2	1	8	4	2	23	251	40	2
4	0	1	1	16	3	1	310	13933	20	1
5		1	0	19	0	1	3468	9719	10	1
6		0		13		0	13277	49	0	1
7				6			14241	7		0
8				0			3192	0		
9							396			
10							0			

Tab. 8.9: The number of unique solutions for several social golfer instances.

This is due to some synergy between the extra constraints and the symmetry breaking method used: roughly speaking, the more pruning, the easier the average dominance check becomes, presumably because the nodes pruned tend to have more expensive checks (see Table 8.8). Since the dominance checks form a significant part of the run time, this can be quite a noticeable effect.

Using the (HV) setting, we have been able to compute all unique solutions for all instances with at most 5 groups and 5 players per group (see Table 8.9). To the best of our knowledge, this is the first computational approach that has been able to compute these numbers for instances of this size. Moreover, we found solutions for many previously unsolved (at least by constraint programming) larger instances, such as the 10-6-6 and the 9-7-4 instances.[1] Finally, we were able to solve the formerly unknown 6-5-6 and 6-5-7 instances. We computed a six week solution for the six groups of five instance, proved that there are exactly two unique solutions for that instance, and showed that these solutions are optimal by proving that no seven week solution exists.

8.5 Summary

For the Social Golfer Problem, we developed an algorithm that efficiently computes unique solutions only. Symmetry breaking is based on the concept of SBDD. We have introduced the idea of heuristic constraint propagation for complex redundant constraints. We proposed two different types of additional constraints, so-called horizontal and vertical constraints. Propagating both types of constraints exactly would require the computation of all the maximal cliques in residual graphs with certain structural properties. Instead, we perform an incomplete propagation using

[1]An overview of solutions found by constraint programming can be found at [105].

a local search method to find a number of maximal cliques (but perhaps not all of them and perhaps not the largest possible). We have shown how such sets of cliques can be used for domain filtering and pruning. The experiments clearly show that adding tight redundant constraints to the problem can be of benefit, even when they are only propagated incompletely.

Our ability to use local search for domain filtering and proving unsatisfiability relies on the fact that we have identified sufficient conditions for proving that a partial schedule cannot be extended to a complete one, and that a local search procedure can be defined such that a solution returned is a proof that the conditions have been met. It would be interesting to see what other kinds of constraints this can be done for, and whether it can be generalized far enough to pass Challenge 5 from [196].

Chapter 9

Graph Bisection

The last problem that we consider in this thesis is the NP-hard Graph Bisection Problem. We develop an algorithm that, given a graph, determines its exact bisection width. Due to its inherent hardness, we cannot hope for an efficient algorithm to tackle the problem (at least in terms of worst case running times and under the common assumption that NP $\neq$ P), but we aim at increasing the size of instances that can still be solved in an affordable amount of time.

As it is the case for any combinatorial optimization problem, the task of computing the bisection width of a graph is twofold. First, an optimal solution to the problem must be computed, and second, its optimality must be proven. Regarding the first task, efficient heuristics have been developed in the literature to compute high quality and, for small graphs, often even optimal solutions very quickly. To prove the optimality of a given solution, we use a total enumeration branch & bound approach. The key issue for the development of a good tree search algorithm is to compute tight lower bounds on the bisection width.

In [197], Sensen presented a new lower bound for the bisection width of a graph that consists in solving a generalized Maximum Multicommodity Flow Problem: We need to solve a Maximum Multicommodity Flow Problem on an undirected graph, whereby the commodities have exactly one source, but may have many sinks.

In contrast to the previous chapters, problem reduction is a minor subject of our research for the Graph Bisection Problem. However, we have seen that filtering based on cost considerations relies on a tight bound on the objective. Our contribution here is the development of time and memory efficient algorithms that approximate Sensen's lower bound. Building up on existing techniques for the solution of Multicommodity Flow Problems, we develop two bound computation routines: the first is a fully polynomial time approximation scheme (FPTAS), and the second is a Lagrangian relaxation based cost-decomposition approach. Both routines are embedded in a branch & bound-framework and compared with a barrier LP-solver on various test instances. The algorithms presented allow us to compute the bisection width of large structured graphs, such as DeBruijn 9 and Shuffle-Exchange 10, which were unknown and out of the reach of exact graph bisection algorithms before.

The work presented has not been published yet. It is joint work with Norbert Sensen and Larissa Timajev. The chapter is structured as follows: In Section 9.1, we introduce the Graph Bisection Problem. In Section 9.2, we review the lower bounds on the bisection width that have been proposed in [197], especially the so-called *VarMC-bound*. In Sections 9.3 and 9.4, we develop two algorithms for the computation/approximation of that VarMC-bound. The complete branch & bound-approach is sketched in Section 9.5. Finally, in Section 9.6, we compare the different algorithms on a set of test instances.

9.1 The Graph Bisection Problem

The Graph Bisection Problem is defined as follows:

Definition 9.1 *Let $G = (V,E,u)$ denote an undirected, edge-weighted graph, whereby u_{vw} is the weight of the edge $\{v,w\} \in E$. Since G is undirected, $u_{vw} = u_{wv}$ for any $\{v,w\} \in E$.*

- *Let $2 \leq k \in \mathbb{N}$ and $V_1,\ldots,V_k \subset V$ such that, for all $1 \leq i < j \leq k$, $V_i \cap V_j = \emptyset$ and $V = V_1 \cup \cdots \cup V_k$. Then we call $(V_1,\ldots,V_k)$ a k-partition (of G). A* balanced k-partition *of a graph G additionally satisfies the condition $||V_i| - |V_j|| \leq 1$ for any $1 \leq i < j \leq k$.*
- *Given a k-partition $(V_1,\ldots,V_k)$, we set $U := \{\{v,w\} \in V_i \times V_j \mid 1 \leq i < j \leq k\} \subseteq E$. Then the value $\sum_{\{v,w\} \in U} u_{vw}$ is called the* cut size *of the partition. The minimal cut size among all balanced k-partitions of a graph G is called the* k-section width *of G.*
- *For $k = 2$, a k-partition is also called a* bisection *(of G). The minimum cut size over all balanced bisections of G is called the* bisection width *of G.*
- *Given an edge-weighted, undirected graph G, the* Graph Bisection Problem *consists in the computation of the bisection width of G.*

For many special graph classes (such as grids, tori, hyper-cubes, butterflies, cube-connected-cycles etc.), the bisection width has been sought out by theoretical reflections. However, in general the Graph Bisection Problem is NP-hard [89]. The best known approximation algorithm for the problem is presented in [69], where a poly-logarithmic algorithm is presented that achieves an approximation quality in $O(\log^2 n)$.

In practice, the size of instances that can still be solved efficiently is rather small. Until today, optimal bisections can be computed for small graphs with a few hundred vertices only. On the other hand, there are very efficient heuristics available for the problem [109, 134, 164, 172, 214]. Therefore, the main problem is the proof of optimality, and thus, the computation of tight lower bounds on the bisection width of a graph.

In the last years, a number of approaches have been presented that solve the Graph Partitioning Problem exactly. In [72], a branch & cut algorithm for the problem is presented. The same approach is followed in [27], whereas in [124], the Graph Partitioning Problem is tackled using a column generation approach. The most recent and apparently also most successful approach is presented in [133] where semi-definite programming relaxations are used as lower bounds. An experimental comparison of our developments and the semi-definite bound in [133] is given in Section 9.6.

9.2 Bounds on Graph Bisection

Our work is based on the lower bounds on the Graph Partitioning Problem introduced in [197]. In the following, we give a small survey on the main ideas of these bounds:

A well-known lower bound on the bisection width can be achieved by embedding a clique with the same number of nodes n into the given graph G [140]. If that complete graph can be embedded with a congestion C, we know that G has a bisection width of at least $\frac{n^2}{4C}$. A similar lower bound on the bisection width can be computed by solving a Multicommodity Flow Problem: For every node in G, we introduce a commodity that originates at its corresponding node. Every other node requires exactly one unit of that commodity. That way, every node has to send one unit of its corresponding commodity to every other node. Note that we do not need to enforce integrality constraints on the flows while strengthening the bound computed.

We can improve this bound by taking two steps of generalization: First, it can be observed that every single-source multicommodity flow instance (with arbitrary demands and destinations) can be used to compute a lower bound on the bisection width. The critical point is the $CutFlow$, i.e. the amount of flow which can be ensured to cross every possible bisection of the graph. It is easy to show that $\frac{CutFlow}{C}$ is always a valid lower bound on the bisection width of a graph.

Second, we do not have to select an appropriate multicommodity flow instance by ourselves: Typically, linear programming techniques are used to solve Multicommodity Flow Problems. In [197], it is proposed to leave the selection of an appropriate multicommodity flow instance to the linear program by adding some variables and constraints. Two different possibilities with a different degree of freedom for the selection of the Multicommodity Flow Problem have been introduced: the *VarMC* and the *MVarMC* formulations. In the MVarMC formulation, every node has the freedom to send a commodity of arbitrary size to each other node. On the other hand, in the VarMC formulation, every node has to send a commodity of arbitrary size to every other node, whereby each destination gets the same share. Experiments have shown that the VarMC formulation gives equally good bounds on connected graphs as the MVarMC formulation. Therefore, we develop different algorithms for the computation of the VarMC-bound.

9.3 Approximation of the VarMC-bound

Even though the continuous Multicommodity Flow Problem (MMCF) is solvable in polynomial time, for more than a decade now researchers have tried to develop approximation schemes for the problem. The reason for this at first surprising fact is that large multicommodity flow instances have to be solved in many application areas and in combination with a whole variety of discrete optimization problems such as network design or graph bisection. Standard simplex or interior point LP-solvers, even specialized software based on primal basis partitioning [66, 67] or Lagrangian relaxation based resource- or cost-decompositions [44, 84] are simply not fast enough to tackle real-size instances of these problems in a reasonable amount of time. Therefore, there is a big interest in the development of algorithms that provide near-optimal solutions more quickly.

The first FPTAS's for maximum multicommodity flow were based on Lagrangian relaxations and linear programming. They have been improved and adapted to different models in a series of papers [100, 101, 131, 135, 141, 169, 175, 198]. While all this research was built on the idea of rerouting existing flows, the idea of augmenting flow has led to a couple of new algorithms with improved running times [74, 75, 90, 130, 217]. Several publications also report about the usability of approximation algorithms for Multicommodity Flow Problems in practice [2, 98, 101, 176, 184].

In this section, we develop an ε-approximation scheme for the VarMC-bound: Let $G = (V, E, u)$ denote an undirected, edge-weighted graph with $|V| = n$, $|E| = m$, with associated node-arc incidence matrix $N \in \{-1, 0, 1\}^{n \times 2m}$, and capacities $u \in \mathbb{Q}^m_{\geq 0}$. Furthermore, let $K \in \mathbb{N}$ denote the number of commodities with source nodes $s^k \in V$ and demands $d^k \in \mathbb{Q}^n$, $1 \leq k \leq K$, whereby $d^k_i \geq 0$ for all $i \neq s^k$, and $\sum_i d^k_i = 0$. Finally, denote the cost coefficient of commodity k by p^k. Then the problem is to

$$\begin{array}{llcll}
\text{Maximize} & \sum_k p^k \lambda^k & & & \\
\text{subject to} & N x^k & = & \lambda^k d^k & \forall\, 1 \leq k \leq K \qquad (1) \\
 & \sum_k x^k_{ij} + x^k_{ji} & \leq & u_{ij} & \forall\, \{i, j\} \in E \qquad (2) \\
 & \lambda, x & \geq & 0 &
\end{array}$$

This way to state the problem allows us directly to compute a lower bound on the bisection width of a given graph (compare with Section 9.2): the objective maximizes the cutflow, whereby the variables λ^k determine the sender volume of commodity k, and the x^k give the corresponding flow in the network. The constraints (1) ensure that x^k is a feasible flow of commodity k with volume λ^k, and restrictions (2) enforce that the capacity of the undirected edges $\{i, j\}$ is not violated.

Note that, to solve the problem, it is sufficient to look at the special case with $p^k = 1$ for all $1 \leq k \leq K$, because for $p^k \leq 0$ we set $\lambda^k = 0$ and $x^k = 0$, and otherwise we can scale the demand d^k by $1/p^k$.

9.3.1 The FPTAS

We try to keep the chapter self-contained. Note however, that the following description is analogue to the "maximum multicommodity flow"-section in [74], which itself builds directly on the analysis given in [90]. Our contribution here consists in the generalization of the existing algorithms for the Maximum Multicommodity Flow Problem to an FPTAS that can handle commodities with more than one sink. This, of course, is essential for the computation of a lower bound for the bisection width of a graph (see Section 9.2). At the same time, since we focus on graph bisection here, we enable the FPTAS to handle undirected edges. However, the changes that are necessary for that purpose are not of fundamental nature, so that the theory we present can also be used for the development of an approximation scheme for generalized Maximum Multicommodity Flow Problems on directed graphs with multi-sink commodities.

For all $1 \leq k \leq K$, denote the set of all paths from the source s^k to the sink $i \in V \setminus \{s^k\}$ by $\pi^k(i)$. Furthermore, set $\pi^k = \cup_{i \neq s^k} \pi^k(i)$. Using variables y_P^k representing the flow of commodity k along some path $P \in \pi^k$, we achieve a *path-based* formulation of the problem

$$\begin{array}{lrcll}
\text{Maximize} & \sum_k \lambda^k & & & \\
\text{subject to} & \sum\limits_{P \in \pi^k(i)} y_P^k & = & \lambda^k d_i^k & \forall\, 1 \leq k \leq K,\ i \in V \setminus \{s^k\} \\
 & \sum_k \sum\limits_{P \in \pi^k : \{i,j\} \in P} y_P^k & \leq & u_{ij} & \forall\, \{i,j\} \in E \\
 & \lambda, y & \geq & 0 &
\end{array}$$

The dual of the previous LP can be written as

$$\begin{array}{lrcll}
\text{Minimize} & \sum\limits_{\{i,j\} \in E} u_{ij} l_{ij} & & & \\
\text{subject to} & \sum\limits_{\{i,j\} \in P} l_{ij} & \geq & z_h^k & \forall\, 1 \leq k \leq K,\ h \in V \setminus \{s^k\},\ P \in \pi^k(h) \\
 & \sum\limits_{i \neq s^k} d_i^k z_i^k & \geq & 1 & \forall\, 1 \leq k \leq K \\
 & l & \geq & 0 &
\end{array}$$

That is, we have to assign lengths $l_{ij} \geq 0$ to each edge $\{i,j\} \in E$, such that $\sum_{i \neq s^k} d_i^k dist_i^k(l) \geq 1$ for all $1 \leq k \leq K$, and $\sum_{\{i,j\} \in E} u_{ij} l_{ij}$ is minimized, whereby $dist_i^k(l)$ denotes the shortest path distance from s^k to the sink $i \in V$ in G under length function l.

The approximation scheme that we investigate in the following is sketched in Algorithm 5. We start with length function $l \equiv \delta$ for an appropriately defined $\delta = \delta(\varepsilon, G, d)$, and the primal solution $x \equiv 0$. The length function l is defined on the undirected edge set E, whereas we maintain flow values x_{ij}^k and x_{ji}^k for each edge $\{i,j\} \in E$ and all $1 \leq k \leq K$. While there is still a tree T along which we can route the demand of a commodity k with costs less than 1, the algorithm selects such a tree and augments flow along this tree. More precisely, the algorithm selects a

Algorithm 5 APPROXIMATION SCHEME

```
1: $x :\equiv 0, l :\equiv \delta$
2: $\hat{\alpha} \leftarrow \min_k \sum_i d_i^k \delta$, $oldSource \leftarrow -1$
3: repeat
4:   for all $1 \leq k \leq K$ do
5:     if $oldSource \neq s^k$ then
6:       $T \leftarrow$ SHORTEST_PATH_TREE($s^k$, $l$)
7:       $oldSource \leftarrow s^k$
8:     while $\sum_i d_i^k dist_i^k(l) < \min\{1, (1+\varepsilon)\hat{\alpha}\}$ do
9:       for all $(i,j) \in T$ do
10:        $c_{ij}^k \leftarrow \sum_{h \in T_j} d_h^k$
11:      $\varphi \leftarrow \min_{(i,j)\in T} (2x_{ji}^k + u_{ij})/c_{ij}^k$, $\Delta :\equiv 0$
12:      for all $(i,j) \in T$ do
13:        $\Delta_{ji} \leftarrow -\min\{x_{ji}^k, \varphi c_{ij}^k\}$
14:        $\Delta_{ij} \leftarrow \varphi c_{ij}^k + \Delta_{ji}$
15:        if $\Delta_{ij} + \Delta_{ji} > 0$ then
16:          $l_{ij} \leftarrow l_{ij}(1 + \varepsilon(\Delta_{ij} + \Delta_{ji})/u_{ij})$
17:      $T \leftarrow$ SHORTEST_PATH_TREE($s^k$, $l$)
18:      $x \leftarrow x + \Delta$
19:   $\hat{\alpha} \leftarrow \hat{\alpha}(1+\varepsilon)$
20: until $\hat{\alpha} \geq 1$
```

tree with approximately minimal costs up to an approximation factor of $1+\varepsilon$. This property is achieved by maintaining a lower bound $\hat{\alpha}$ on the current minimal routing costs of any commodity.

The amount of flow sent along tree T is determined in the following way: Let T_j denote all nodes in the sub-tree rooted at node $j \in V$ (including j). For each edge $\{i,j\} \in T$, we compute the congestion c_{ij}^k when routing the demand of commodity k along that tree, i.e., we set $c_{ij}^k = \sum_{h \in T_j} d_h^k$. Basically, we achieve a feasible routing by scaling the flow by $\min_{\{i,j\}\in T} u_{ij}/c_{ij}^k$. However, since we are working on an undirected network here, we would like to consider only flows with $\min\{x_{ij}^k, x_{ji}^k\} = 0$ for all $1 \leq k \leq K$ and $\{i,j\} \in E$. When we also incorporate and change the current flow x_{ji}^k of commodity k in the opposite direction, we achieve an even bigger scaling factor of $\min_{(i,j)\in T}(2x_{ji}^k + u_{ij})/c_{ij}^k$. Formally, we can prove Lemma 9.1 regarding the change $\Delta_{ij} + \Delta_{ji}$ of the current flow on edge $\{i,j\} \in E$ of commodity k.

In case of a positive flow change on an edge $\{i,j\} \in E$ (i.e., when $\Delta_{ij} + \Delta_{ji} > 0$), we update the dual variables by setting $l_{ij} \leftarrow (1 + \varepsilon(\Delta_{ij} + \Delta_{ji})/u_{ij})l_{ij}$, i.e., we increase the lengths of an edge with respect to the congestion of that edge.

Finally, the primal solution is updated by $x_{ij}^k \leftarrow x_{ij}^k + \Delta_{ij}$. This setting may yield an infeasible solution, since it may violate some capacity constraint $\sum_k (x_{ij}^k + x_{ji}^k) \leq u_{ij}$ for some edge $\{i, j\} \in E$. However, the mass balance constraints are still valid. This allows us, at the end of the algorithm, to scale the final flows x^k so that they build a feasible solution to the problem.

For the algorithm sketched above, we are able to prove

Theorem 9.1 *Let $T_{SP} = m + n \log n$, and $S = \{d_i^k \mid d_i^k > 0,\ 1 \leq k \leq K,\ i \in V\}$. An ε-approximate VarMC-bound on the bisection width of a graph can be computed in time $O^*((mT_{SP} + |S|)/\varepsilon^2)$.*

Following the analysis in [74], we prove the previous theorem with the help of a sequence of lemmata. We set $\rho = \min_{k,i}\{d_i^k \mid d_i^k > 0\}$, and $\sigma = \log_{1+\varepsilon}((1+\varepsilon)/(\delta\rho))$. Then,

Lemma 9.1 *In every iteration in which the current flow is changed, it holds that:*

a) $\Delta_{ij} + \Delta_{ji} \leq u_{ij}$ for all $\{i, j\} \in E$, and

b) there exists an edge $\{i, j\} \in E$ such that $\Delta_{ij} + \Delta_{ji} = u_{ij}$.

Proof: Let $\varphi = \min_{(i,j)\in T}(2x_{ji}^k + u_{ij})/c_{ij}^k$.

a) For $(i, j), (j, i) \notin T$ there is no change in the flow. Let $(i, j) \in T$. In case of $x_{ji}^k = 0$, we have $\Delta_{ij} + \Delta_{ji} = \Delta_{ij} = \varphi c_{ij}^k \leq (2x_{ji}^k + u_{ij})c_{ij}^k / c_{ij}^k = u_{ij}$. In case of $x_{ji}^k \geq \varphi c_{ij}^k$, we have $\Delta_{ji} = -\varphi c_{ij}^k$ and $\Delta_{ij} = 0$. Thus, $\Delta_{ij} + \Delta_{ji} = \Delta_{ji} = -\varphi c_{ij}^k \leq 0 \leq u_{ij}$. In case of $0 < x_{ji}^k < \varphi c_{ij}^k$, we have $\Delta_{ij} + \Delta_{ji} = (\varphi c_{ij}^k - x_{ji}^k) + (-x_{ji}^k) \leq (2x_{ji}^k + u_{ij})c_{ij}^k / c_{ij}^k - 2x_{ji}^k = u_{ij}$. Finally, the case $(j, i) \in T$ is analogue to $(i, j) \in T$.

b) Consider the edge $(i, j) \in T$ with $\varphi = (2x_{ji}^k + u_{ij})/c_{ij}^k$. In case of $x_{ji}^k = 0$, we have $\Delta_{ij} + \Delta_{ji} = \Delta_{ij} = \varphi c_{ij}^k = (2x_{ji}^k + u_{ij})c_{ij}^k / c_{ij}^k = u_{ij}$. In case of $x_{ji}^k > 0$, it holds that $x_{ji}^k < 2x_{ji}^k + u_{ij} = (2x_{ji}^k + u_{ij})c_{ij}^k / c_{ij}^k = \varphi c_{ij}^k$. Thus, $\Delta_{ij} + \Delta_{ji} = (\varphi c_{ij}^k - x_{ji}^k) + (-x_{ji}^k) = (2x_{ji}^k + u_{ij})c_{ij}^k / c_{ij}^k - 2x_{ji}^k = u_{ij}$.

■

Lemma 9.2 *The flow obtained by scaling the final flow by $1/\sigma$ is primal feasible.*

Proof: Let $\Delta_{ij}(t)$ denote the change of the flow on edge $(i, j) \in T$ in iteration t. Then, $\sum_{t \leq I}(\Delta_{ij}(t) + \Delta_{ji}(t))$ equals the flow on edge $\{i, j\} \in E$ after iteration I. It is sufficient to show that, at the end of the algorithm, it holds that $\sum_{t \leq I}(\Delta_{ij}(t) + \Delta_{ji}(t)) \leq \sigma u_{ij}$.

In each iteration with $\Delta_{ij}(t) + \Delta_{ji}(t) > 0$, the length l_{ij} of edge $\{i, j\} \in E$ increases by a factor of $1 + \varepsilon(\Delta_{ij}(t) + \Delta_{ji}(t))/u_{ij}$. Denote the set of all iterations t in which $\Delta_{ij}(t) + \Delta_{ji}(t) > 0$ by I. Then we have that

$$l_{ij} = \delta \prod_{t \in I} \left(1 + \varepsilon \frac{\Delta_{ij}(t) + \Delta_{ji}(t)}{u_{ij}}\right). \tag{9.1}$$

With Lemma 9.1 and $1+\varepsilon x \geq (1+\varepsilon)^x$ for all $0 \leq x \leq 1$, we have that

$$l_{ij} \geq \delta \prod_{t\in I} (1+\varepsilon)^{\frac{\Delta_{ij}(t)+\Delta_{ji}(t)}{u_{ij}}} = \delta(1+\varepsilon)^{\sum_{t\in I} \frac{\Delta_{ij}(t)+\Delta_{ji}(t)}{u_{ij}}}. \tag{9.2}$$

Thus, at the end of the algorithm we have

$$\sum_{t\in I} (\Delta_{ij}(t)+\Delta_{ji}(t)) \leq u_{ij} \log_{1+\varepsilon} \frac{l_{ij}}{\delta}. \tag{9.3}$$

Since the left hand side is a valid upper bound for the final congestion on edge $\{i,j\} \in E$, it remains to show that $l_{ij} \leq (1+\varepsilon)/\rho$. Assume the opposite. With Lemma 9.1, l_{ij} always increases by a factor of at most $1+\varepsilon$. Thus, before the last change it must hold $l_{ij} > 1/\rho$. Consider the iteration in which l_{ij} increases the last time during the increase of the flow regarding some commodity $1 \leq k \leq K$. We know that then $\sum_{h\neq s^k} d_h^k dist_h^k(l) < 1$. However, since $l_{ij} > 1/\rho$ and (i,j) is an edge on the shortest path from s^k to one of its sinks, we have that $\sum_{h,\, d_h^k \in S} dist_h^k(l) > 1/\rho$. Thus,

$$\sum_{h\neq s^k} d_h^k dist_h^k(l) \geq \rho \sum_{h,\, d_h^k\in S} dist_h^k(l) > \frac{\rho}{\rho} = 1, \tag{9.4}$$

which is a contradiction. ■

Lemma 9.3 *Let $\tau = (1+\varepsilon)/\rho$, and denote the maximum number of edges in a simple path from a source s^k to one of its sinks $i \in V$ by L. When setting $\delta = \tau(\tau L \max_k \sum_{i\neq s^k} d_i^k)^{-1/\varepsilon}$, the final flow scaled by $1/\sigma$ is optimal with a relative error of at most 3ε.*

Proof: We prove the desired accuracy of the solution computed by comparing it against the objective value D of a dual feasible solution, which our algorithm produces as a byproduct, and which gives us a valid upper bound on the primal optimal solution value Z_{opt}.

For any given length function $l : E \to \mathbb{Q}_{\geq 0}$, let α denote the minimal routing costs of all commodities, i.e., $\alpha = \alpha(l) = \min_k \sum_{i\neq s^k} d_i^k dist_i^k(l)$. Furthermore, denote the dual objective value corresponding to the current choice of l by $D(l) = \sum_{\{i,j\}\in E} u_{ij} l_{ij}$. Then the optimal dual objective value is $Z_{opt} = \min_l D(l)/\alpha(l)$.

Consider iteration t, and let $l(t)$, k, φ, c and T denote the current choice of the length function, commodity, scaling factor, congestion vector and routing tree, respectively. For ease of notation, we set $\alpha(t) = \alpha(l(t))$, $D(t) = D(l(t))$, and $dist_i^k(t) = dist_i^k(l(t))$. For any edge $\{i,j\} \in E$, define $\Gamma_{ij} = \max\{\Delta_{ij}(t)+\Delta_{ji}(t), 0\}$. Note that, in every iteration t, it holds that $\Gamma_{ij} \leq \varphi c_{ij}^k$. Then,

$$\begin{aligned} D(t) &= D(t-1) + \varepsilon \sum_{\{i,j\}\in T} \Gamma_{ij} l_{ij}(t-1) \leq D(t-1) + \varepsilon \sum_{\{i,j\}\in T} \varphi c_{ij}^k l_{ij}(t-1) \\ &= D(t-1) + \varepsilon\varphi \sum_{h\neq s^k} d_h^k \sum_{\{i,j\}\in T;\, h\in T_j} l_{ij}(t-1) \\ &\leq D(t-1) + \varepsilon\varphi \sum_{h\neq s^k} d_h^k dist_h^k(t-1) < D(t-1) + \varepsilon\varphi(1+\varepsilon)\alpha(t-1). \end{aligned}$$

Denote the primal objective value $\sum_k \lambda^k$ (corresponding to the — possibly infeasible — flows x^k) after the iteration t by $Z(t)$ ($Z(0) = 0$). Obviously, it holds that $Z(t) = Z(t-1) + \varphi$. Thus, $D(t) < D(t-1) + \varepsilon(1+\varepsilon)(Z(t) - Z(t-1))\alpha(t-1)$, and therefore

$$D(t) < D(0) + \varepsilon(1+\varepsilon) \sum_{1 \le h \le t} (Z(h) - Z(h-1))\alpha(h-1). \tag{9.5}$$

Consider the length function $l(t) - l(0) = l(t) - \delta$. Then, for the dual objective, we have $D(l(t) - l(0)) = D(t) - D(0)$, and for the primal objective, we get $\alpha(l(t) - l(0)) \ge \alpha(t) - L\delta \max_k \sum_{i \ne s^k} d_i^k$, where L denotes the maximum number of edges on a simple path from a source s^k to one of its sinks $i \in V$. Hence,

$$Z_{opt} \le \frac{D(l(t) - l(0))}{\alpha(l(t) - l(0))} \le \frac{D(t) - D(0)}{\alpha(t) - L\delta \max_k \sum\limits_{i \ne s^k} d_i^k}. \tag{9.6}$$

Thus,

$$\alpha(t) < L\delta \max_k \sum_{i \ne s^k} d_i^k + \frac{\varepsilon(1+\varepsilon)}{Z_{opt}} \sum_{1 \le h \le t} (Z(h) - Z(h-1))\alpha(h-1). \tag{9.7}$$

Denote the right hand side of Inequality 9.7 by $A(t)$. Then,

$$\begin{aligned} A(t) &= A(t-1) + \frac{\varepsilon(1+\varepsilon)}{Z_{opt}}(Z(t) - Z(t-1))\alpha(t-1) \\ &< A(t-1)(1 + \frac{\varepsilon(1+\varepsilon)}{Z_{opt}}(Z(t) - Z(t-1))) \le A(t-1)e^{\frac{\varepsilon(1+\varepsilon)}{Z_{opt}}(Z(t) - Z(t-1))} \\ &\le A(0)e^{\frac{\varepsilon(1+\varepsilon)Z(t)}{Z_{opt}}}, \end{aligned}$$

because of $1 + x \le e^x$ for all $x \in \mathbb{Q}$ and $Z(0) = 0$. Now consider the last iteration t. Then, $\alpha(t) \ge 1$. With $A(0) = L\delta \max_k \sum_{i \ne s^k} d_i^k$, we get

$$1 \le \alpha(t) < A(t) < L\delta \max_k \sum_{i \ne s^k} d_i^k e^{\frac{\varepsilon(1+\varepsilon)Z(t)}{Z_{opt}}}. \tag{9.8}$$

Thus,

$$Z(t) > \frac{Z_{opt}}{\varepsilon(1+\varepsilon)} \ln \frac{1}{L\delta \max_k \sum\limits_{i \ne s^k} d_i^k}. \tag{9.9}$$

When setting δ according to Lemma 9.3, a simple calculation shows that for the scaled objective value we get

$$\frac{Z(t)}{\sigma} > Z_{opt} \frac{1-\varepsilon}{\varepsilon(1+\varepsilon)} \ln 1 + \varepsilon \quad \Rightarrow \quad \frac{Z(t)}{\sigma} > Z_{opt}(1 - 3\varepsilon), \tag{9.10}$$

for all $\varepsilon < 1$. ■

Proof of Theorem 9.1: In Lemmas 9.2 and 9.3, we have shown that the algorithm returns a feasible flow of the desired accuracy. It remains to show that the running time is polynomial.

The running time is dominated by the operations in Lines 6, 8, 10 and 17. Setting $\nu = \min_k \sum_{i \neq s^k} d_i^k$ and $\mu = \log_{1+\varepsilon}((1+\varepsilon)/(\nu\delta)) < \sigma$, the following holds:

6) Since $\hat{\alpha}$ is initially set to $\nu\delta$, and $\hat{\alpha} < 1+\varepsilon$ at the end of the computation, we know that the outer while loop is executed at most $\log_{1+\varepsilon}((1+\varepsilon)/(\nu\delta)) = \mu$ times. If we order the commodities according to the corresponding source nodes s^k, we have to compute Line 6 at most $n\mu$ times. Thus, this line adds a workload of $O(n\sigma T_{SP})$.

8) Obviously, every execution of Line 8 may require time $O(n)$. However, the simple estimation of a cost of kn per outer while iteration can be strengthened by observing that in each such iteration all flow destinations will be investigated exactly once. Thus, for every outer while iteration this line adds a workload of $O(|S|)$. For every inner while iteration the work to be done in Line 8 is dominated by Line 15. Thus, it is sufficient to add a workload of $O(|S|\sigma)$ for Line 8.

9,10) At first, the computation of the current congestion c_{ij}^k on each edge $(i,j) \in T$ seems to require a running time quadratic in n. However, when computing the shortest path tree T, we get a topological ordering of T for free (simply by using the ordering in which Dijkstra's algorithm labels the nodes). Using this topological ordering, we can compute the values $\sum_{h \in T_j} d_h^k$ bottom up, requiring a total running time in $O(n)$. Thus, Lines 9 and 10 are dominated by Line 17.

17) Whenever a shortest-tree computation results in a flow change along the computed tree, we know from Lemma 9.1 that at least for one edge the length increases by a factor of $1+\varepsilon$. At the beginning, all lengths are equal to δ, and the proof of Lemma 9.2 shows that, at the end of the algorithm, it holds that $l_{ij} \leq (1+\varepsilon)/\rho$. Therefore, Line 17 is executed at most $m\log_{1+\varepsilon}((1+\varepsilon)/(\delta\rho)) = m\sigma$ times. Thus, Line 17 adds a workload of $O(m\sigma T_{SP})$.

Putting the results together and assuming $m \geq n$, we get a running time in

$$O(n\sigma T_{SP} + |S|\sigma + m\sigma T_{SP}) = O((mT_{SP} + |S|)\sigma). \tag{9.11}$$

Thus, when we set δ according to Lemma 9.3, we achieve a running time in

$$O((mT_{SP} + |S|)/(\log n + \log(\max_k \sum_{i \neq s^k} d_i^k/\rho))\varepsilon^2) = O^*((mT_{SP} + |S|)/\varepsilon^2). \tag{9.12}$$

■

9.3.2 Implementation Details

The previous theoretic work gives us an approximation scheme for a lower bound on the bisection width of a graph. Note that, with respect to Section 9.2, we know that $|S| \in O(n^2)$.[1] Therefore, for any connected graph we can achieve an ε-approximation of the VarMC-bound on the bisection width in time $O^*(m^2/\varepsilon^2)$.

Even though our theoretical work does not give any further guarantees, in practice, the approximation scheme presented can be improved. Most importantly, the final scaling factor σ used to make the final flow primal feasible should not be determined by the formula given in Lemma 9.2. Instead, we can easily determine a scaling factor $\hat{\sigma} \leq \sigma$ by computing the congestion c_{ij} of the final flow on each edge $\{i,j\} \in E$ and setting $\hat{\sigma} \leftarrow \max_{(i,j)\in E} c_{ij}/u_{ij}$. This improvement does not affect the overall running time of the algorithm.

In practice, we may consider to do even more and to apply an idea that we call *enhanced scaling*: before scaling the final flow, at the end of the algorithm, for each commodity k we obtain a flow x^k and possibly also a scalar $\lambda^k = -|x^k|/d^k_{s^k} \geq 0$. In order to construct a feasible flow, instead of scaling all x^k equally, we could also set up another optimization problem to find scalars ξ^k that solve the following LP:

$$\begin{array}{lrcll} \text{Maximize} & \sum_k \lambda^k \xi^k & & & \\ \text{subject to} & \sum_k \xi^k (x^k_{ij} + x^k_{ji}) & \leq & u_{ij} & \forall\, \{i,j\} \in E \\ & \xi & \geq & 0 & \end{array}$$

That way, the final bound obtained can be improved in practice. However, this gain has to be paid for by an additional computational effort that, in theory, dominates the overall running time. However, as we shall see in Section 9.6, when solving an instance of the Graph Bisection Problem, the effort is taken worthwhile.

9.4 Cost-Decomposition Approach

Another way of computing the VarMC-bound on the bisection width is to use cost-based decomposition techniques, that were originally developed for the Min-Cost Multicommodity Flow Problem. The idea is to relax the capacity constraints (another term used in the literature is *bundle constraints*) in order to decompose the problem into a set of Shortest Path Problems.

Recall from Section 9.2 that we need to compute flows that guarantee a certain cutflow $\hat{F}$ while keeping the maximum congestion on any edge within a given limit $\hat{C}$. More formally, we

[1]Without going into details here, we would like to add that it even holds that $|S| \leq n^2$, and this remains true even when new commodities are added when branching.

need to find flows x^k such that

$$\begin{array}{lcll} Nx^k & = & \lambda^k d^k & \forall\, 1 \le k \le K \\ \sum_k x^k_{ij} + x^k_{ji} & \le & u_{ij}\hat{C} & \forall\, \{i,j\} \in E \\ \sum_k \lambda^k & \ge & \hat{F} & \\ \lambda, x & \ge & 0 & \end{array} \qquad (LP\ 1)$$

We will investigate a transformation of this problem into an optimization problem, namely by trying to minimize the congestion while guaranteeing that the required cutflow is achieved. We develop an approach on that optimization problem using an integration of column generation and Lagrangian relaxation.

9.4.1 Column Generation

The path-based formulation in Section 9.3.1 could be used to build a column generation approach on. However, taking into account the large number of source-sink pairs that we are facing when computing the VarMC-bound on the bisection width of a given graph — it is in $\Theta(n^2)$ — we would have to cope with an LP with extremely many constraints. For example, for the DeBruijn 9, the simplex tableau would have more than 2^{18} rows. In contrast to the number of columns that can be controlled first by generating only columns with negative reduced costs and second by successive matrix compressions, such a huge number of rows cannot be handled efficiently. Thus, in practice we cannot afford to generate columns that represent paths in the graph, even though this would be advantageous with respect to the total number of columns that have to be generated in order to achieve a near-optimal solution. For the same reason, a master problem consisting of *key paths* and *cycles* as it was proposed in [12] cannot be handled efficiently for our application. Thus, we will use a master problem that is based on trees.

Let $M^k = \{T^k_g \mid 1 \le g \le G^k\}$ denote the set of all trees rooted at s^k and routing the demand of commodity k to its sinks (where $|M^k| = G^k$ denotes the number of all such tree routings). Define $M := \cup_k M^k$, and let $c^k_g(j) = \sum_{h \in T^k_g(j)} d^k_h$ denote the congestion on an edge $(i,j) \in T^k_g$ (i.e., i is the unique predecessor of j in T^k_g) for all $1 \le k \le K$ and $T^k_g \in M^k$. Then the problem can be written as

$$\begin{array}{llcll} \text{Minimize} & LP_C = C & & & \\ \text{subject to} & \sum\limits_{k \le K} \sum\limits_{g \le G^k} \xi_g c^k_g(j) & \le & u_{ij}C & \forall\, \{i,j\} \in E \\ & \sum\limits_{k \le K} \sum\limits_{g \le G^k} \xi_g & \ge & \hat{F} & \\ & \xi & \ge & 0 & \end{array}$$

Starting with a subset $\hat{M} \subset M$, we solve the reduced *master problem*. Using dual variables $r_{ij} \le 0$ for the capacity constraints and $f \ge 0$ for the cut-flow restriction, we generate new columns with negative reduced costs by solving the sub-problem

$$\begin{array}{lrcll} \text{Minimize} & \multicolumn{4}{l}{\sum_{\{i,j\}\in E}(x_{ij}^k + x_{ji}^k)(-r_{ij}) - f} \\ \text{subject to} & Nx^k & = & d^k & \forall\, 1 \le k \le K \\ & x & \ge & 0 & \end{array}$$

Note that the sub-problem decomposes into K single-source Shortest Path Problems with non-negative edge costs. We can prune the search, if $LP_C < \hat{C}$.

The process of generating columns and solving the master problem is iterated until we cannot compute trees with associated negative reduced costs anymore or until a master iteration limit is reached. The number of columns in the master matrix is controlled by a frequent compressing step that reduces the number of columns with respect to the current associated reduced costs. However, our experiments showed that the compression must not be carried out too aggressively, because we need a fairly large number of columns in the master matrix in order to keep the total number of master iterations within reasonable limits. This is a clear drawback of the tree-based formulation of the master problem that results in a relatively high solution time for the master problem.

Thus, we try to keep the number of master iterations small by adding a whole set of columns between two master problem solutions. The only question that must be answered is how to generate meaningful new columns without the help of new dual variables. We use Lagrangian relaxation for this purpose, first on a min-congestion formulation of the problem, and second, on a max-cutflow representation. The idea to use Lagrangian relaxation to generate new columns is motivated by the fact that the optimal Lagrangian multipliers in the min-congestion formulation are also optimal dual values for the column generation procedure and vice versa.

The whole procedure works as follows: we start a subgradient optimization of the Lagrangian dual and achieve an upper bound for the cutflow or a lower bound on the congestion, respectively. At the same time, we feed the tree-flows computed in the successive Lagrangian sub-problems into the matrix of the master problem. If the upper bound on the maximum cutflow is smaller than $\hat{F}$, or if the lower bound on the minimum congestion as greater than $\hat{C}$, respectively, we have proven that the VarMC-bound does not allow to prune the current search node, and we can branch right away. Otherwise, we solve the master problem and achieve a feasible flow that yields an upper bound on the congestion. If we achieve a flow with an associated congestion smaller than $\hat{C}$, we can prune the current search node. Otherwise, we restart the Lagrangian sub-routine with the current dual values again. This process is iterated until we find optimal flows or an iteration limit is reached.

9.4.2 Lagrangian Relaxation Based Column Generation

To complete the description, we briefly present the two Lagrangian relaxations that we consider to generate columns in a column generation framework in more detail.

By relaxing the capacity constraints in LP 1 using Lagrangian multipliers $r_{ij} \leq 0$, we get a max-cutflow formulation

$$\begin{array}{lrcll} \text{Maximize} & \multicolumn{4}{l}{\sum_k (\lambda^k + \sum_{\{i,j\}\in E} (x^k_{ij} + x^k_{ji}) r_{ij}) - \hat{C} r^T u} \\ \text{subject to} & Nx^k & = & d^k & \forall\, 1 \leq k \leq K \\ & x & \geq & 0 & \end{array}$$

Or, we can aim at a min-congestion formulation

$$\begin{array}{lrcll} \text{Minimize} & \multicolumn{4}{l}{\sum_k \sum_{\{i,j\}\in E} (x^k_{ij} + x^k_{ji})(-r_{ij}) + C(1 + r^T u)} \\ \text{subject to} & Nx^k & = & d^k & \forall\, 1 \leq k \leq K \\ & \sum_k \lambda^k & \geq & \hat{F} & \\ & x & \geq & 0 & \end{array}$$

In both cases, we need to solve K Single-Source Shortest Path Problems again. Thus, both formulations allow us to use the shortest path trees computed in the Lagrangian sub-problems to generate columns for the master problem. Note that both Lagrangian sub-problems may be unbounded. This problem is overcome by setting upper bounds on λ^k (for example the out-capacity of s^k) or on C (for example $1.1\hat{C}$), respectively.

The update of the Lagrangian multipliers can be done by different subgradient algorithms using different formulas for the computation of the new search direction d^t. Let s^t denote a subgradient in iteration t. We can set $d^t = s^t$ (pure subgradient); or $d^t = \alpha d^{t-1} + s^t$, whereby $0 < \alpha < 1$ is fix (so-called *Crowder rule* [46]); or we may set $d^t = \alpha^t d^{t-1} + s^t$, with $\alpha^t = ||s^t||/||d^{t-1}||$ if $(s^t)^T d^{t-1} \leq 0$, and $\alpha = 0$ otherwise (*modified Camerini-Fratta-Maffioli rule* [28]); another possibility is to set $d^t = \alpha d^{t-1} + (1-\alpha)s^t$, whereby $0 < \alpha < 1$ is fix (so-called *volume algorithm* [11]). All variants will be evaluated in the numeric section.

Interestingly, there are some similarities to the approximation scheme presented in Section 9.3. In both cases, we compute a sequence of upper and lower bounds. In the approximation scheme as well as in column generation and Lagrangian relaxation we compute shortest path trees with respect to some changing length function. The only difference is how we change that length function. In the approximation scheme, it is increased exponentially with respect to the current congestion on an edge. In the subgradient algorithm for Lagrangian relaxation, it is changed with respect to search directions that reflect a current subgradient and possibly parts of the search history. In column generation, new lengths are simply set to the current dual values in the master problem. Thus, we consider an experimental comparison of these different strategies of interest.

9.5 A Branch & Bound Algorithm

Our main goal is the computation of exact solutions for Graph Bisection Problems. Therefore, we construct a branch&bound algorithm using the described VarMC-bound as lower bound for the problems. A detailed description of the branch & bound implementation can be found in [197]. In the following, we give a brief survey on the main ideas:

First, we heuristically compute a graph bisection using *PARTY* [172]. Since the initial solution obtained is optimal in most cases, usually we only need to prove optimality. A pure depth first search tree traversal is sufficient for that purpose. The branching is done on the decision whether two specific vertices $\{v,w\}$ stay in the same partition (join) or if they are separated (split). A join is performed by adapting the graph by merging these two vertices into one vertex. A split is performed by introducing an additional commodity from vertex v to vertex w whose entire scaled amount is known to cross the cut. Thus, it can be added to the $CutFlow$ completely. The selection of the pair $\{v,w\}$ for the next branching is done with the help of an upper bound on the lower bound. Additionally to this idea described in [197], the selection is restricted to pairs $\{v,w\}$ (if any) where one node (say, v) has been split with some other node $x \neq w$ before. Then the split of $\{v,w\}$ implies a join of $\{x,w\}$, of course.

Problem reduction is done by improving the so-called "Forcing Moves" strategy described in [197], Lemma 2. In the original version, only adjacent vertices $\{v,w\}$ were considered. Now, we look at a residual graph with edge-capacities that equal the amount of capacity which is not used by a given VarMC solution. Two vertices v and w can be joined if the maximal flow in the residual graph exceeds a specific value.

9.6 Numerical Results

In this section, we present the results of our computational experiments. Before comparing the algorithms with each other and with the semi-definite bound developed in [133], we first present some experiments regarding the effects of different parameter settings for the FPTAS in Section 9.3 and the cost-decomposition approach in Section 9.4. The following experiments were executed on systems with INTEL Pentium-III, 850 MHz, and 512 MB memory. To show the performance on different kinds of graphs, we use four different sets of 20 randomly generated graphs: The set *RandPlan* contains random maximal planar graphs with 100 vertices; the graphs are generated using the same principle as it is used in LEDA [153]. Benchmark set *RandReg* consists of random regular graphs with 100 vertices and degree four; for its generation, the algorithm from Steger and Wormald [206] is used. The set *Random* contains graphs with 44 vertices where every pair $\{v,w\}$ is adjacent with probability 0.2. Finally, the set *RandW* consists of complete graphs with 24 vertices where every edge has a random weight in the interval $\{0,\dots,99\}$.

In most works on exact graph partitioning (see e.g. [27, 133]), sets like *Random* and *RandW*

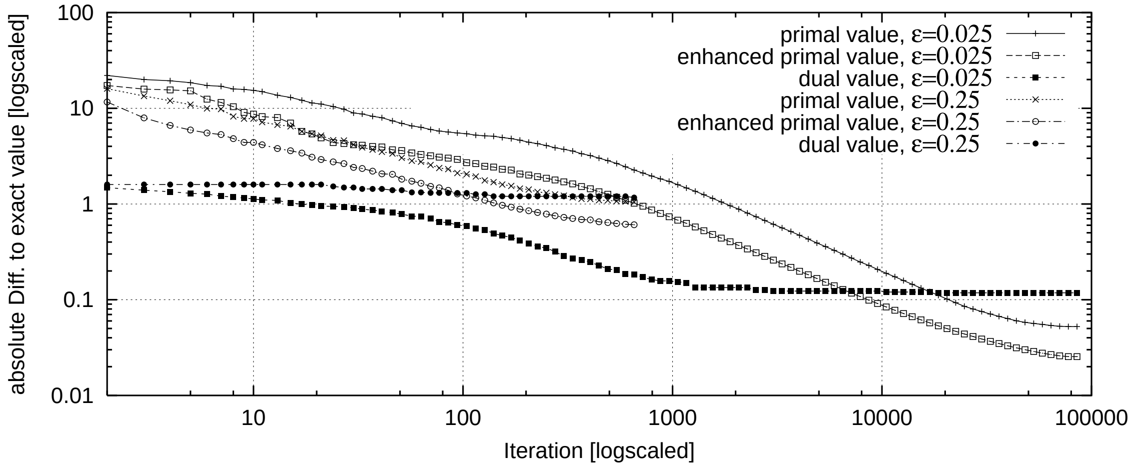


Fig. 9.1: Progression of the bounds with $\varepsilon = 0.025$ and $\varepsilon = 0.25$

are used for the experiments. We added the sets of random regular and random planar graphs here, because we believe that more structured graph classes should also be considered taking into account their bigger relevance for practical applications.

9.6.1 Approximating Lower Bounds

First, we show the results regarding the FPTAS developed in Section 9.3. To illustrate the behavior of the algorithm, Figure 9.1 shows the progression of the primal and dual value, and the enhanced primal value for $\varepsilon = 0.025$ and $\varepsilon = 0.25$. Recall from Section 9.3.2 that the enhanced primal value is achieved by solving a linear program to compute the optimal scaling factors of the final flows. The run was made on one specific RandReg graph, but similar results are obtained by using any of the other graphs we considered in our experiments. Thus, we consider this example as representative.

Interestingly, it appears that the improvement caused by enhanced scaling is a specific factor that is almost independent of the number of iterations. As one might have expected, a closer look at the data shows that the improvement usually gets slightly smaller with more iterations, whereby the effect is most visible for graphs in the *Random* set. Only for planar graphs we found that the gain by enhanced scaling becomes clearly greater with increasing iterations. Regarding the dependency of the chosen ε, we find that for the sets *RandReg* and *RandPlan* the improvement achieved by enhanced scaling becomes smaller the greater ε is, and for (weighted) random graphs it is almost constant.

In Figure 9.1, for both settings of ε, the dual value converges to its final error fairly quickly and then remains nearly unchanged. When comparing the primal values with $\varepsilon = 0.25$ and $\varepsilon = 0.025$, we find that the first has a smaller error at the beginning, but is not improved anymore after few iterations. Using $\varepsilon = 0.025$, the convergence of the primal value is slower, but reaches a clearly better result at the end.

ε	$\hat{\varepsilon}$	$\hat{\varepsilon_s}$	$\hat{\varepsilon_d}$	num its.	time
0.0125	0.0008	0.00035	0.0022	361,014	3360.0
0.025	0.0019	0.00091	0.0042	89,026	888.3
0.05	0.0043	0.0021	0.0084	21,642	232.2
0.1	0.011	0.0062	0.016	5,110	56.0
0.2	0.029	0.016	0.033	1,130	12.9
0.4	0.069	0.040	0.057	213	2.4
0.6	0.12	0.067	0.057	66	0.8
0.8	0.20	0.11	0.057	23	0.3

Tab. 9.1: Real errors and computational effort depending on the given ε.

The best choice of ε for the use in a branch & bound environment is a trade-off. If ε is too big, the bound approximation computed is too bad, and the number of sub-problems in the search tree explodes. On the other hand, if ε is chosen too small, the bound approximation for each sub-problem is too time consuming. Table 9.1 shows this trade-off for the same graph as in Figure 9.1. Again, we consider this example as representative. We denote the approximation parameter the algorithm is started with by ε, $\hat{\varepsilon}$ is the final error relative to the real VarMC-bound value, $\hat{\varepsilon_s}$ denotes the final error when enhanced scaling is used, and $\hat{\varepsilon_d}$ gives the final error of the dual value. Furthermore, the number of iterations and the running time in seconds are given.

It gets clear that the error that is actually reached is much better than the approximation guarantee ε which the algorithm is started with. The figure also shows that the theoretical factor of $\frac{1}{\varepsilon^2}$ in the running time of the algorithm is confirmed by the experiments.

Note that the running time for the enhanced scaling version is not explicitly stated in the table. It increases the running time by only a hundredth of a second, and is therefore negligible in our comparison. In this experiment, however, the linear program for the computation of enhanced scaling factors was only solved once at the end of the approximation. When using the FPTAS for lower bound computations in a branch & bound, we are also interested in intermediate primal values that may allow us to prune the current choice point even before the approximation of the current VarMC-bound is finished. Then we found that it is a good choice to compute enhanced primal values only every hundredth iteration, which almost does not increase the overall running time but improves the approximation quality significantly.

In Table 9.2, we give the resulting running times and the number of sub-problems of the branch & bound algorithm using approximated bounds. The results given are averages over all 20 instances for every set of graphs.

The results without forcing moves show the expected behavior for the choice of ε. The smaller ε is, the smaller is the number of search nodes. This rule is not strict when using forcing moves: looking at the random planar graphs, we see that less good solutions of the VarMC-bound can result in stronger forcing moves so that the number of sub-problems may even decrease. The figure also shows that the effects of enhanced scaling and forcing moves are different for the different classes of graphs and also for changing ε's. Altogether, the experiments show that setting ε to 50% is favorable, which is a surprisingly high value.

		ε = 0.025		ε = 0.05		ε = 0.1		ε = 0.25		ε = 0.5		ε = 0.75	
	graph	time	subp.	time	subp.	time	subp.	time	subp.	time	subp.	time	subp.
(1)	RandPlan	6513	462	2350	463	1058	468	632	582	**438**	1536	524	5740
	RandReg	3728	24	1902	29	1001	37	**522**	101	551	552	1194	4063
	Random	2857	114	727	119	334	129	**175**	212	230	1124	781	15038
	RandW	1487	47	534	60	217	89	108	297	**102**	2034	118	11498
(2)	RandPlan	6395	461	2158	461	962	463	587	557	**450**	1466	562	5273
	RandReg	2192	22	1107	23	561	26	196	37	**126**	90	163	489
	Random	2620	113	525	114	228	118	98	139	**80**	280	186	2188
	RandW	1383	37	472	46	176	62	**76**	150	85	788	1289	3859
(3)	RandPlan	3013	412	1009	406	587	381	133	239	54	117	**35**	78
	RandReg	2186	22	1083	23	622	25	181	34	**117**	67	160	173
	Random	2249	107	465	108	209	111	83	121	49	164	**47**	328
	RandW	737	14	283	17	122	25	44	54	**35**	188	41	582

Tab. 9.2: Times and sizes of the search trees of the branch & bound algorithm using the approximation algorithm with different ε's. (1): without enhanced scaling, without forcing moves, (2): with enhanced scaling, without forcing moves, (3): with enhanced scaling and forcing moves.

9.6.2 Lower Bounds using Cost-Decomposition

Now, we evaluate the cost-decomposition approach developed in Section 9.4. Analogically to Table 9.2, the Table 9.3 shows the results of the branch & bound algorithm when using the cost-decomposition technique for the computation of the VarMC-bound.

Both, the running times and the sizes of the search trees produced by the various subgradient algorithms and Lagrangian formulations differ considerably on the different benchmark sets. Thus, it is not an easy task to draw valid conclusions out of these experiments. As a tendency, the max-cutflow formulation looks better than the min-congestion formulation (except for the random planar graphs). When using the max-cutflow formulation, the Crowder rule gives a good overall performance.

9.6.3 Comparison of Lower Bound Algorithms

After having investigated the developed algorithms solitarily, we now want to compare them with each other and with the semi-definite bound presented in [133]. We start with a presentation of time and quality of the four different lower bound algorithms:

- The VarMC-bound using the ILOG CPLEX 7.0 standard barrier solver [117].
- The VarMC-bound using the max-cutflow decomposition with the Crowder rule.

		pure subgr.		Crowder		mod. CFM		Volume	
	graph	time	subp.	time	subp.	time	subp.	time	subp.
(1)	RandPlan	11318	85	**6626**	85	24986	85	7752	85
	RandReg	1192	38	589	28	737	29	**568**	32
	Random	748	117	**694**	119	722	116	849	127
	RandW	151	11	**128**	11	134	10	166	17
(2)	RandPlan	2032	85	**1276**	85	1831	85	1307	85
	RandReg	**3768**	64	17320	299	4374	72	17633	292
	Random	**1776**	160	3366	239	1941	169	6211	418
	RandW	1710	183	**1394**	139	1591	166	3661	588

Tab. 9.3: Average running times in seconds and average number of search nodes using cost-decomposition without forcing moves. (1): max-cutfbw formulation, (2): min-congestion formulation.

- The VarMC-bound using the FPTAS with a desired approximation guarantee of 50% and enhanced scaling.
- The semi-definite bound presented in [133] and available as CUTSDP-package (program bis0) at [132]. The program uses parameters *maxlarge* and *maxsmall*. According to the setting in [133], we set *maxlarge* := 1, and *maxsmall* := 10.

In Tables 9.4, 9.5, and 9.6, we apply the different algorithms to grids, tori, shuffle-exchange graphs, DeBruijn graphs, and graphs stemming from a real-world finite elements application. The experiments were performed on a SUN Enterprise 450 Model 4400 machine with 1 GB main memory and a SUN UltraSparc-II 400 MHz processor. The tables give the number of nodes, the number of edges, the exact bisection width, and, for each algorithm, the bound computed and the time needed for its computation (in seconds).

First, by comparing the VarMC-bound achieved by CPLEX and the semi-definite bound, we see that the VarMC-bound is indeed superior with respect to its quality on sparse and structured graphs like the ones that we consider here. As a slight drawback, the bound is not suited for disconnected graphs like the BCR graphs *ma, m1*, and *m4*. For them, the MVarMC-bound discussed in Section 9.2 yields much better results [197]. Work is in progress that tries to extend the work presented here to the MVarMC bound as well. However, here we focus on the VarMC-bound only.

For the remaining connected, and especially the larger graphs, we see that the VarMC-bound dominates the semi-definite bound, sometimes quite remarkably. Moreover, we see that the CPLEX barrier solver[2] computes the VarMC-bound always faster than CUTSDP obtains the semi-definite bound - except for large shuffle-exchange and DeBruijn graphs. For these graphs, the

[2]We also experimented with the primal and dual simplex, but, while the dual simplex outperforms the primal simplex algorithm, both algorithms could not at all compete with the interior point solver.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	bw	CPLEX		Decomp.		Approx (50%)		CUTSDP	
				Bound	Time	Bound	Time	Bound	Time	Bound	Time
Grid 9x4	36	59	5	4.50	<1	4.50	2	4.32	<1	5.00	1
Grid 10x5	50	85	5	5.00	1	5.00	4	5.00	<1	5.00	3
Grid 10x9	90	161	9	9.00	15	9.00	21	9.00	1	8.12	17
Grid 10x10	100	180	10	10.00	26	10.00	32	10.00	1	8.28	24
Grid 10x11	110	199	11	11.00	16	11.00	54	10.59	2	8.48	30
Grid 11x11	121	220	12	11.48	19	11.48	67	11.11	2	8.98	42
Torus 9x4	36	72	10	8.10	<1	8.10	3	7.78	<1	7.60	1
Torus 10x5	50	100	10	10.00	3	10.00	4	9.84	<1	9.88	3
Torus 10x9	90	180	18	18.00	19	18.00	32	17.56	1	17.22	19
Torus 10x10	100	200	20	20.00	11	20.00	38	18.95	2	17.98	27
Torus 10x11	110	220	22	20.17	15	20.17	48	19.66	2	17.63	30
Torus 11x11	121	242	24	22.18	15	22.00	65	20.62	3	18.50	39

Tab. 9.4: Comparison of bounds on Grids and Tori.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	bw	CPLEX		Decomp.		Approx (50%)		CUTSDP	
				Bound	Time	Bound	Time	Bound	Time	Bound	Time
SE 2	4	3	2	2.00	<1	2.00	<1	2.00	<1	2.00	<1
SE 3	8	10	2	2.00	<1	2.00	<1	2.00	<1	2.00	<1
SE 4	16	21	4	3.10	<1	3.10	<1	3.08	<1	3.49	<1
SE 5	32	46	6	5.23	<1	5.20	2	5.10	<1	5.08	2
SE 6	64	93	10	8.91	8	8.87	11	8.69	<1	7.40	21
SE 7	128	190	16	15.12	43	15.01	53	14.73	3	11.31	120
SE 8	256	381	28	26.15	484	25.69	244	24.94	16	18.14	453
DB 2	4	5	4	4.00	<1	4.00	<1	4.00	<1	4.00	<1
DB 3	8	13	4	4.00	<1	4.00	<1	4.00	<1	4.00	<1
DB 4	16	29	6	6.00	<1	6.00	<1	5.89	<1	6.00	<1
DB 5	32	61	10	10.00	<1	10.00	3	9.82	<1	9.70	2
DB 6	64	125	18	16.96	9	16.90	14	16.26	<1	14.94	20
DB 7	128	253	30	28.98	62	28.80	65	27.45	4	22.58	49
DB 8	256	509	54	49.54	652	49.05	322	46.95	21	35.08	443

Tab. 9.5: Comparison of bounds on shuffle-exchange (SE) and DeBruijn (DB) graphs.

Graph	$\|V\|$	$\|E\|$	bw	CPLEX		Decomp.		Approx (50%)		CUTSDP	
				Bound	Time	Bound	Time	Bound	Time	Bound	Time
BCR ma	54	72	2	0	–	0	–	0	–	1.93	4
BCR mb	74	120	4	3.08	8	3.08	9	3.08	4	3.12	10
BCR mc	74	125	6	5.10	8	5.10	14	5.10	4	5.45	10
BCR md	80	129	4	3.16	9	3.16	11	3.15	4	3.20	13
BCR me	60	96	3	3.00	3	3.00	6	3.00	3	3.00	5
BCR mf	90	146	4	3.21	14	3.21	18	3.21	4	2.86	18
BCR m1	100	155	4	0	–	0	–	0	–	2.46	26
BCR m4	32	50	6	0	–	0	–	0	–	5.68	1
BCR m6	70	120	7	6.36	7	6.36	14	6.36	4	6.03	9
BCR m8	148	265	7	7.00	22	7.00	64	6.96	5	5.98	80

Tab. 9.6: Comparison of bounds on real-world graphs stemming from a fi nite elements application.

quality of the VarMC-bound is much better, though. Therefore, even when using a standard barrier solver for its computation, the VarMC-bound is clearly preferable to the semi-definite bound on sparse, structured graphs.

However, for larger graphs containing 500 nodes or more, the memory consumption of CPLEX becomes critical. For example, the bound on DeBruijn 9 could not be computed within 2 GB main memory. As stated in Section 9.4, cost-decomposition can help to cope with that situation. We see that the Lagrangian relaxation based column generation approach yields slightly worse lower bounds, and there is also more time needed, except for large shuffle-exchange and DeBruijn graphs. In exchange, the memory requirements are much lower, and the cost-decomposition approach allows us to tackle large graphs like DeBruijn 9 and Shuffle-Exchange 10.

When using the approximation scheme with an approximation guarantee of 50%, we lose even more of the bounds quality. However, in most cases, the bounds obtained are still better than those computed by CUTSDP, and the computation time is drastically reduced. At the same time, the memory consumption is comparable to the cost-decomposition approach. Therefore, the FPTAS developed in Section 9.3 is the algorithm of choice for the bisection of sparse and structured graphs.

Next, we embed the three different algorithms for the VarMC-bound into the branch & bound approach with problem reduction as sketched in Section 9.5. Again, the experiments were executed on systems with 850 MHz INTEL Pentium-III processors. Table 9.7 shows the results on the four previously described benchmark sets.

We see that, also within a branch & bound approach, the enhanced approximation algorithm gives the best running times, even though the bounds are worst and the search trees are largest. Recall from the formulation of the the master problem in Section 9.4, that the cost-decomposition approach was especially designed to be memory efficient, which makes it less competitive with respect to the running time.

Finally, we note that, using the FPTAS in Section 9.3, we are able to compute the bisection widths of DeBruijn 9 (92), Shuffle-Exchange 9 (48), and Shuffle-Exchange 10 (82) with the additional help of the symmetry breaking method SBDD developed in Chapter 4. In our view, these results impressively show the efficiency of the VarMC-bound as well as the FPTAS for its approximation.

	CPLEX		Decomp.		Approx. (50%)	
graph	time	subp.	time	subp.	time	subp.
RandPlan	74	7	889	26	**54**	117
RandReg	993	21	557	28	**117**	67
Random	350	99	612	106	**49**	164
RandW	**30**	9	120	11	35	188

Tab. 9.7: Average running times (seconds) and sizes of the search trees using the different methods for computing the VarMC-bound.

9.7 Summary and Future Work

We developed two specialized algorithms for the computation/approximation of the VarMC-bound on the bisection width of an undirected, edge-weighted graph. The first algorithm is based on an approximation scheme (FPTAS) for maximum multicommodity flows and yields an ε-approximation in time $O^*(m^2/\varepsilon^2)$. We could show empirically that the real error obtained is usually much better than the approximation guarantee, especially when using an enhanced scaling method at the end of the algorithm.

The second algorithm that we developed uses the idea of cost-decomposition in an integration of Lagrangian relaxation and column generation. We compared two different Lagrangian formulations for the generation of columns and four different rules to determine the search direction in a subgradient algorithm. The performance of the different algorithms varies a lot on different graph classes, and it is a hard task to find a set of robust parameter settings that guarantee a stable performance.

When comparing the two algorithms with a barrier LP-solver and a semi-definite program, we found that it is clearly favorable to use the approximation scheme that yields very good bounds on sparse, structured graphs in very little time. It allowed us to compute the bisection widths of large graphs, such as DeBruijn 9, Shuffle-Exchange 9, and Shuffle-Exchange 10, which were unknown and out of the reach of exact graph bisection algorithms before.

As a subject of future work, we investigate the possibility to adapt the FPTAS that we developed for the VarMC-bound for an approximation of the MVarMC-bound. Then, we hope to be able to tackle also disconnected graphs for which the VarMC-bound is inefficient.

Chapter 10

Conclusion

In the introduction, we observed that there exists an incongruity of the needs for optimization software on one hand and, on the other hand, the solutions that algorithmic computer science is able to offer. From an algorithmic point of view, the optimization abilities of todays software libraries are often more than satisfactory for many real-life applications. However, the need for complex problem modeling appears as a major obstacle for a broader use of optimization software.

We tried to improve upon this situation by providing filtering algorithms for higher level symbolic constraints. They allow to model real-life problems more intuitively while preserving the strong optimization abilities of mathematical programming. We believe that the set of optimization constraints that we considered covers very important substructures that arise frequently in real-life applications. However, our presentation is not exhaustive, and more work has to be done to provide practitioners with a more complete set of higher level building blocks for problem modeling.

Whenever a problem is decomposed into substructures — even when they are larger than usual — the question arises of how a global view on the entire problem can be achieved. Especially with respect to tight bounds on the objective, the answer to this question is crucial. We have proposed to link optimization constraints via well-known decomposition techniques from operations research. When the user is able to provide a solver with information about the substructures of a problem, we believe that the linking of optimization constraints via CP-based column generation and CP-based Lagrangian relaxation can also be automated and hidden from the user.

Regarding symmetry breaking, the method that we proposed does not by itself detect the symmetries in a problem model. Instead, the representation of a search node and the symmetry detection function still must be provided. Therefore, a deeper knowledge is required from the user. However, the idea to think of symmetries algorithmically by asking: When are two choice points symmetric? is much more intuitive than to develop a problem model that contains no symmetries. Therefore, we believe that our method can help inexperienced users to cope with symmetry more efficiently.

When tackling the optimization problems that we considered in the second part of this thesis, the methods and reduction algorithms developed in Part I accelerated the software development process considerably. Moreover, as we have seen, they can yield to competitive algorithms. However, note that, especially for the Capacitated Network Design Problem, the Social Golfer Problem, and the Airline Crew Assignment Problem, we added problem specific knowledge to improve the efficiency. In our view, as a subject of future work, it would be desirable to generalize the ideas of local Lagrangian cuts, heuristic constraint propagation and the repair techniques for column generation. Finally, the work on the Graph Bisection Problem motivates the question whether approximation algorithms can be exploited for problem reduction as well, which may give yield to a notion of relaxed ε-consistency for optimization constraints.

In the end, a brief note that goes beyond the scientific scope of this thesis. We aimed at giving a broader access to optimization power. However, we strongly believe that efficiency is no value by itself. Optimization is to be seen as a tool that can be used and misused. We therefore ask to exploit it with care and responsibility for the good of the people.

List of Figures

List of Tables

Bibliography

[1] R.K. Ahuja, T.L. Magnati, and J.B. Orlin. *Network Flows.* Prentice Hall, 1993.

[2] C. Albrecht. Provably good global routing by a new approximation algorithm for multicommodity flow. *International Conference on Physical Design*, pp. 19–25, 2000.

[3] E. Andersson, E. Housos, N. Kohl, and D. Wedelin. Crew pairing optimization. *International Series in Operations Research and Management Science*, 9:228–258, Kluwer Academic Publishers, 1998.

[4] Y. Aneja, V. Aggarwal, and K. Nair. Shortest chain subject to side conditions. *Networks*, 13:295-302, 1983.

[5] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3:149–156, 1991.

[6] K. R. Apt. The Rough Guide to Constraint Propagation. *5th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 1713:1–23, 1999.

[7] A. Atamtürk, D. Rajan. On Splittable and Unsplittable Capacitated Network Design Arc-Set Polyhedra. To appear in *Mathematical Programming*, 2001.

[8] G. Ausiello, G.F. Italiano, A.M. Spaccamela, and U. Nanni. Incremental Algorithms for Minimal Length Paths. *Journal of Algorithms*, 12(4): 615–638, 1990.

[9] *mediaTV*, Technical Description, Axcent AG. `http://www.axcent.de`.

[10] E. Balas and E. Zemel. An algorithm for large-scale zero-one knapsack problems. *Operations Research*, 28:119–148, 1980.

[11] F. Barahona and R. Anbil. The Volume Algorithm: producing primal solutions with a subgradient algorithm. *Mathematical Programming*, 87:385–399, 2000.

[12] C. Barnhart, C.A. Hane, E.L. Johnson, and G. Sigismondi. A column generation and partitioning approach for multi-commodtiy flow problems. *Telecommunication Systems*, 3:239–258, 1995.

[13] 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.

[14] 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.

[15] N. Barnier and P. Brisset. Graph Coloring for Air Traffic Flow Management. *4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, pp. 133–147, 2002.

[16] N. Barnier and P. Brisset. Solving the Kirkman's Schoolgirl Problem in a Few Seconds. *8th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2470:477–491, 2002.

[17] J. Beasley and N. Christofides. An Algorithm for the Resource Constrained Shortest Path Problem. *Networks*, 19:379-394, 1989.

[18] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. *Logic Programming: Formal Methods and Practical Applications*, pp. 245–272, Elsevier, 1995.

[19] J.C. Bermond and C. Peyrat. De Bruijn and Kautz networks: a competitor for the hypercube? *1st European Workshop on Hypercubes and Distributed Computers*, pp. 279–293, North-Holland, 1989.

[20] M. Bern and P. Plassmann. The Steiner problem with edge lengths 1 and 2. *Information Processing Letters (IPL)*, 32:171–176, 1989.

[21] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[22] D. Bienstock, O. Günlük, S. Chopra, and C.Y. Tsai. Mininum cost capacity installation for multicommodity flows. *Mathematical Programming*, 81:177-199, 1998.

[23] D. Bienstock. Experiments with a network design algorithm using epsilon-approximate linear programs. *Technical Report*, CORC Report 1999-4, 1999.

[24] 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.

[25] R. Borndörfer and A. Löbel. Scheduling duties by adaptive column generation. *Technical Report*, Konrad-Zuse-Zentrum für Informationstechink Berlin, ZIB-01-02, 2001.

[26] C. Bornstein, A. Litman, B. Maggs, R. Sitaraman, and T. Yatzkar. On the Bisection Width and Expansion of Butterfly Networks. *1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, IEEE, pp. 144–150, 1998.

[27] L. Brunetta, M. Conforti, and G. Rinaldi. A branch-and-cut algorithm for the equicut problem. *Mathematical Programming*, 78:243–263, 1997.

[28] P. Camerini, L. Fratta, and F. Maffioli. On Improving Relaxation methods by Modified Gradient Techniques. *Mathematical Programming Studies*, 3:26–34, 1975.

[29] A. Caprara, M. Fischetti, and P. Toth. A heuristic algorithm for the set covering problem. *5th International Conference on Integer Programming and Combinatorial Optimization (IPCO)*, LNCS 1084:72–84, 1996.

[30] 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.

[31] A. Caprara, D. Pisinger, and P. Toth. Exact Solution of the Quadratic Knapsack Problem. *INFORMS Journal on Computing*, 11:125–137, 1999.

[32] A. Caprara, P. Toth, D. Vigo, and M. Fischetti. Modeling and solving the crew rostering problem. *Operations Research*, 46(6):820–830, 1998.

[33] R.D. Carr, L.K. Fleischer, V.J. Leung, and C.A. Phillips. Strengthening Integrality Gaps for Capacitated Network Design and Covering Problems. *11th Symposium on Discrete Algorithms (SODA)*, 2000.

[34] Y. Caseau and F. Laburthe. Solving Various Weighted Matching Problems with Constraints. *3rd International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 1330:17–31, 1997.

[35] Y. Caseau and F. Laburthe. Solving Small TSPs with Constraints. *14th International Conference on Logic Programming (ICLP)*, pp. 316–330, The MIT Press, 1997.

[36] Y. Caseau and F. Laburthe. Heuristics for large constrained routing problems. *Journal of Heuristics*, 5:281–303, 1999.

[37] 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.

[38] C. Chu and J. Antonio. Approximation algorithm to solve real-life multicriteria cutting stock problems. *Operations Research*, 47(4):495–508, 1999.

[39] H.D. Chu, E. Gelman, and E.L. Johnson. Solving large scale crew scheduling problems. *European Journal of Operational Research*, 97:260–268, 1997

[40] L.W. Clarke and P. Gong. Capacitated Network Design with Column Generation. *Research Report*, Georgia Institute of Technology, 1998.

[41] M.B. Cohen, C.J. Colbourn, L.A. Ives, and A.C.H. Ling. Kirkman triple systems of order 21 with non-trivial automorphism group. *Mathematics of Computation*, 71:873–881, 2001.

[42] C. Colbourn and J. Dinitz. *The CRC Handbook of Combinatorial Designs*. CRC Press, 1996.

[43] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[44] T.G. Crainic, A. Frangioni, and B. Gendron. Bundle-based relaxation methods for multicommodity capacitated fixed charge network design. *Discrete Applied Mathematics*, 112:73–99, 2001.

[45] T.G. Crainic, M. Gendreau, and J.M. Farvolden. A simplex-based tabu search method for capacitated network design. *INFORMS Journal on Computing*, 12(3):223–236, 2000.

[46] H. Crowder. Computational improvements for subgradient optimization. *Symposia Mathematica*, XIX:357–372, 1976.

[47] *CSPLib: a problem library for constraints*, maintained by I.P. Gent, T. Walsh, and B. Selman, `http://www-users.cs.york.ac.uk/~tw/csplib/`

[48] J. Czyzyk, S. Mehrotra, M. Wagner, and S.J. Wright. PCx user guide (Version 1.1). *Technical Report*, Optimization Technology Center, Aragone National Laboratory and Northwestern University, 1996.

[49] J. Czyzyk, S. Mehrotra, M. Wagner, and S.J. Wright. PCx: An interior-point code for linear programming. *Optimization Methods and Software*, 11(2):397–430, 1999.

[50] G.B. Dantzig. Discrete variable extremum problems. *Operations Research*, 5:266–277, 1957.

[51] G.B. Dantzig and P. Wolfe. The decomposition algorithm for linear programs. *Econometrica*, 29(4):767–778, 1961.

[52] P.R. Day and D.M. Ryan. Flight attendant rostering for short-haul airline operations. *Operations Research*, 45(5):649–661, 1997.

[53] R.S. Dembo and P.L. Hammer. A reduction algorithm for knapsack problems. *Methods of Operations Research*, 36:49–60, 1980.

[54] C. Demetrescu and G.F. Italiano. Fully Dynamic All Pairs Shortest Paths with Real Edge Weights. *42nd Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, pp. 260–267, 2001.

[55] G. Desaulniers, J. Desrosiers, Y. Dumas, S. Marc, B. Rioux, M.M. Solomon, and F. Soumis. Crew pairing at Air France. *European Journal of Operational Research*, 97:245–259, 1997.

[56] J. Desrosiers, Y. Dumas, M.M. Solomon, and F. Soumis. Time constrained routing and scheduling. *Network Routing — Handbooks in Operations Research and Management Science*, 8:35–139, North-Holland, 1995.

[57] I. Dumitrescu and N. Boland. The weight-constrained shortest path problem: preprocessing, scaling and dynamic programming algorithms with numerical comparisons. *17th International Symposium on Mathematical Programming (ISMP)*, 2000.

[58] ECLIPSE. Parc Technologies Limited. `http://www.icparc.ic.ac.uk/eclipse/`.

[59] H. Everett. Generalized lagrange multiplier method for solving problems of optimum allocation of resource. *Operations Research*, 11:399–417, 1963.

[60] T. Fahle. Cost Based Filtering vs. Upper Bounds for Maximum Clique. *4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, pp. 93–107, 2002.

[61] T. Fahle, U. Junker, S.E. Karisch, N. Kohl, M. Sellmann, and B. Vaaben. Constraint Propagation for Complex Column Generation Subproblems. *17th International Symposium on Mathematical Programming (ISMP)*, 2000.

[62] 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.

[63] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry Breaking. *7th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2239:93–107, 2001.

[64] T. Fahle and M. Sellmann. Constraint Programming Based Column Generation with Knapsack Subproblems. *2nd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, Paderborn Center for Parallel Computing, Technical Report tr-001-2000:33–44, 2000.

[65] T. Fahle and M. Sellmann. Cost-Based Filtering for the Constrained Knapsack Problem. *Annals of Operations Research*, 115:73–93, 2002.

[66] J. Farvolden, K. Jones, I. Lustig, and W. Powell. Multicommodity Network Flows — The Impact Of Formulation On Decomposition. *Mathematical Programming*, 62:95–117, 1993.

[67] J. Farvolden, W. Powell, and I. Lustig. A primal partitioning solution for the arc-chain formulation of a multicommodity network flow problem. *Operations Research*, 41(4):669–693, 1993.

[68] D. Fayard and G. Plateau. An algorithm for the solution of the 0-1 knapsack problem. *Computing*, 28:269–287, 1982.

[69] U. Feige and R. Krauthgamer. A Polylogarithmic Approximation of the Minimum Bisection. *Journal on Computing*, 31(4):1090–1118, 2002.

[70] R. Feldmann, B. Monien, P. Mysliwietz, and S. Tschöke. A Better Upper Bound on the Bisection Width of de Bruijn Networks. *14th International Symposium on Theoretical Aspects of Computer Science (STACS)*, LNCS 1200:511–522, 1997.

[71] T. Feo and M. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[72] C.E. Ferreira, A. Martin, C.C. de Souza, R. Weismantel, and L.A. Wolsey. The node capacitated graph partitioning problem: a computational study. *Journal of Mathematical Programming*, 81:229–256, 1998.

[73] P.O. Fjällström. Algorithms for graph partitioning: A survey. *Linköping Electronic Articles in Computer and Information Science,* `http://www.ep.liu.se/ea/cis/-1998/010/`, 1998.

[74] L.K. Fleischer. Approximating Fractional Multicommodity Flow Independent of the Number of Commodities. *SIAM Journal on Discrete Mathematics*, 13(4):505–520, 2000.

[75] L. Fleischer and K.D. Wayne. Fast and simple approximation schemes for generalized flow. *Mathematical Programming*, 91(2):215–238, 2002.

[76] F. Focacci, F. Laburthe, and A. Lodi. Local Search and Constraint Programming. *Handbook of Metaheuristic*, Kluwer Academic Publishers, to appear.

[77] F. Focacci, A. Lodi, and M. Milano. Solving TSP through the Integration of OR and CP Techniques. *Workshop on Large Scale Combinatorial Optimization and Constraints*, Electronic Notes in Discrete Mathematics, 1998.

[78] F. Focacci, A. Lodi, and M. Milano. Integration of CP and OR methods for Matching Problems. *1st International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR),* `http://www.deis.unibo.it/Events/Deis/Workshops/-Proceedings.html`, 1999.

[79] F. Focacci, A. Lodi, and M. Milano. Cost-Based Domain Filtering. *5th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 1713:189–203, 1999.

[80] F. Focacci, A. Lodi, and M. Milano. Cutting Planes in Constraint Programming: An Hybrid Approach. *2nd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, Paderborn Center for Parallel Computing, Technical Report tr-001-2000:45–51, 2000.

[81] F. Focacci and M. Milano. Global Cut Framework for Removing Symmetries. *7th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2239:77–92, 2001.

[82] F. Focacci and P. Shaw. Pruning sub-optimal search branches using local search. *4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, pp. 181–189, 2002.

[83] S. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, 1980.

[84] A. Frangioni. A Bundle type Dual-ascent Aproach to Linear Multi-Commodity Min Cost Flow Problems. *Technical Report*, Dipartimento di Informatica, Universita di Pisa, TR-96-01, 1996.

[85] A. Frangioni. Dual Ascent Methods and Multicommodity Flow Problems. *Doctoral Thesis*, Dipartimento di Informatica, Universita di Pisa, TD-97-05, 1997.

[86] M.L. Fredmann and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[87] M. Gamache, F. Soumis, D. Villeneuve, J. Desrosiers, and E. Gélinas. The preferential bidding system at Air Canada. *Transportation Science*, 32(3):246–255, 1998.

[88] M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[89] M.R. Garey, D.S. Johnson, and L. Stockmeyer.. Some simplified NP-complete graph problems. *Theoretical Comuter Sience*, 1:237–267, 1976.

[90] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *39th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, pp. 300–309, 1998.

[91] B. Gendron and T.G. Crainic. Relaxations for multicommodity capacitated network design problems. *Technical Report*, Centre de recherche sur les transports, Université de Montréal, CRT-96-05, 1994.

[92] I.P. Gent, W. Harvey, and T. Kelsey. Groups and Constraints: Symmetry Breaking During Search. *8th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2470:415–430, 2002.

[93] I.P. Gent and B.M. Smith. Symmetry Breaking During Search in Constraint Programming. *14th European Conference on Artificial Intelligence (ECAI)*, pp. 599–603, 2000.

[94] I. Ghamlouche, T.G. Crainic, and M. Gendreau. Cycle-based neighbourhoods for fixed-charge capacitated multicommodity network design. *Technical Report*, Centre de recherche sur les transports, Université de Montréal, CRT-2001-01, 2001.

[95] I. Ghamlouche, T.G. Crainic, and M. Gendreau. Path relinking, cycle-based neighbourhoods and capacitated multicommodity network design. *Technical Report*, Centre de recherche sur les transports, Université de Montréal, CRT-2002-01, 2002.

[96] P.C. Gilmore and R.E. Gomory. A linear programming approach to the cutting stock problem. *Operations Research*, 9:849–859, 1961.

[97] F. Glover, D. Klingman, and N.V. Phillips. *Network Models in Optimization and Their Applications in Practice*. Wiley, 1992.

[98] A.V. Goldberg, J.D. Oldham, S.A. Plotkin, and C. Stein. An Implementation of a Combinatorial Approximation Algorithm for Minimum-Cost Multicommodity Flow. *6th International Conference on Integer Programming and Combinatorial Optimization (IPCO)*, LNCS 1412:338–352, 1998.

[99] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1991.

[100] M.D. Grigoriadis and L.G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM Journal on Optimization*, 4:86–107, 1994.

[101] M.D. Grigoriadis and L.G. Khachiyan. Approximate minimum-cost multicommodity flows. *Mathematical Programming*, 75:477–482, 1996.

[102] O. Günlük. A branch-and-cut algorithm for capacitated network design problems. *Mathematical Programming*, 86(1):17–39, 1999.

[103] G. Handler and I. Zang. A Dual Algorithm for the Restricted Shortest Path Problem. *Networks,* 10:293–310, 1980.

[104] W. Harvey. Symmetry Breaking and the Social Golfer Problem. *Workshop on Symmetry in Constraints (SymCon),* 2001.

[105] W. Harvey. *Warwick's Results Page for the Social Golfer Problem*. `http://-www.icparc.ic.ac.uk/~wh/golf/`.

[106] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. *14th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 607–613, 1997.

[107] M. Held and R.M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[108] M. Held and R.M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1:6–25, 1971.

[109] B. Hendrickson and B. Leland. The chaco user's guide: Version 2.0. *Technical Report*, Sandia National Laboratories, Albuquerque, SAND94-2692, 1994.

[110] D.S. Hochbaum. *Approximation Algorithms for NP-hard Problems.* PWS Publishing Company, 1997.

[111] K.L. Hoffman and M. Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39(6):657–682, 1993.

[112] K. Holmberg and D. Yuan. A Lagrangean Heuristic Based Branch-and-Bound Approach for the Capacitated Network Design Problem. *Operations Research*, 48:461–481, 2000.

[113] J. Hooker. Unifying optimization and constraint satisfaction. *Invited talk at the 16th International Joint Conference on Artificial Intelligence (IJCAI).* Slides available at `http://ba.gsia.cmu.edu/jnh/ijcai.ppt`.

[114] P.D. Hudson. Improving the branch and bound algorithm for the knapsack problem. *Queen's University Research Report*, Belfast, 1977.

[115] J.Y. Hsiao, C.Y. Tang, and R.S. Chang. An efficient algorithm for finding a maximum weight 2-independent set on interval graphs. *Information Processing Letters,* 43(5):229–235, 1992.

[116] ILOG CPLEX 6.5. Reference manual and user manual. ILOG, 1999.

[117] ILOG CPLEX 7.0. Reference manual and user manual. ILOG, 2000.

[118] ILOG CPLEX 7.5. Reference manual and user manual. ILOG, 2001.

[119] ILOG Planner 3.3. Reference manual and user manual. ILOG, 1999.

[120] ILOG Solver 4.4. Reference manual and user manual. ILOG, 1999.

[121] ILOG Solver 5.0. Reference manual and user manual. ILOG, 2000.

[122] G.P. Ingargiola and J.F. Korsh. A reduction algorithm for zero-one single knapsack problems. *Management Science*, 20:460–463, 1973.

[123] O. Jahn, R. Möhring, and A. Schulz. Optimal routing of traffic flows with length restrictions in networks with congestion. *Technical Report*, TU Berlin, 658-1999, 1999.

[124] E. Johnson, A. Mehrotra, and G. Nemhauser. Min-cut clustering. *Mathematical Programming*, 62:133–151, 1993.

[125] H. Joksch. The Shortest Route Problem with Constraints. *Journal of Mathematical Analysis and Application*, 14:191–197, 1966.

[126] M. Jünger and D. Naddef (Editors). *Computational Combinatorial Optimization.* LNCS 2241, 2001.

[127] M. Jünger and S. Thienel. The ABACUS system for Branch and Cut and Price Algorithms in Integer Programming and Combinatorial Optimization. *Software Practice and Experiments*, 30:1325–1352, 2000.

[128] U. Junker, S.E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. A Framework for Constraint programming based column generation. *5th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 1713:261–274, 1999.

[129] U. Junker, S.E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. A Framework for Constraint Programming Based Column Generation. *16th International Joint Conference on Artificial Intelligence (IJCAI)*, Workshop on Non-Binary Constraints, 1999.

[130] G. Karakostas. Fast Approximation Schemes for Fractional Multicommodity Flow Problems. *13th Symposium on Discrete Algorithms (SODA)*, 2002.

[131] D. Karger and S. Plotkin. Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows. *27th Symposium on Theory of Computing*, pp. 18–25, 1995.

[132] S. Karisch. CUTSDP — A toolbox for a cutting-plane approach based on semidefinite programming. *User's guide, Version 1.0,* Department of Mathematical Modeling, Technical University of Denmark, 10/98, 1998.

[133] S.E. Karisch, F. Rendl, and J. Clausen. Solving graph bisection problems with semidefinite programming. *INFORMS Journal on Computing*, 12(3):177–191, 2000.

[134] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. *Technical Report*, Deptartment of Computer Science, University of Minnesota, TR 98-019, 1998.

[135] P. Klein, S. Plotkin, C. Stein, and E. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing*, 23:466–487, 1994.

[136] G. Kliewer, M. Sellmann, and A. Koberstein. Solving the capacitated network design problem in parallel. *3rd meeting of the PAREO Euro working group on Parallel Processing in Operations Research (PAREO),* 2002.

[137] N. Kohl and S.E. Karisch. Airline Crew Rostering: Problem Types, Modeling and Optimization. *Carmen Research and Technology Report*, CRTR-2001-1, 2001.

[138] V. Kumar. Algorithms for Constraints-Satisfaction problems: A Survey. *The AI Magazine*, AAAI, 13:32–44, 1992.

[139] M. Lehradt. Basisalgorithmen für ein TV Anytime System. *Diploma Thesis*, University of Paderborn, 2000.

[140] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, 1992.

[141] T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos, and S. Tragoudas. Fast Approximation Algorithms for Multicommodity Flow Problems. *Journal of Computer and System Sciences*, 50(2):228–243, 1995.

[142] M. Lübbecke and U. Zimmermann. Computer aided scheduling of switching engines. *8th International Conference on Computer-Aided Scheduling of Public Transport (CASPT),* 2000.

[143] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence,* 8(1):99–118, 1977.

[144] T.L. Magnanti and R.T. Wong. Network design and transportation planning: Models and algorithms. *Transportation Science*, 18:1–55, 1984.

[145] K. Marriott and P.J. Suckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.

[146] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and tight bounds for the 0-1 knapsack problem. *Management Science*, 45:414–424, 1999.

[147] S. Martello and P. Toth. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. *European Journal of Operational Research*, 1:169–175, 1977.

[148] S. Martello and P. Toth. A new algorithm for the 0-1 knapsack problem. *Management Science*, 34:633–644, 1988.

[149] S. Martello and P. Toth. *Knapsack Problems — Algorithms and Computer Implementations.* Wiley, 1990.

[150] S. Martello and P. Toth. Upper Bounds and Algorithms for hard 0-1 knapsack problems. *Operations Research*, 45(5):768–778, 1997.

[151] R.D. McBride. Progress made in solving the multicommodity flow problem. *SIAM Journal on Optimization*, 8:947–955, 1998.

[152] I. McDonald. Unique Symmetry Breaking in CSPs Using Group Theory. *Workshop on Symmetry in Constraints (SymCon)*, 2001.

[153] K. Mehlhorn and S. Nähler. LEDA: A Platform for Combinatorial and Geometric Computing. *Communications of the ACM*, 38(1):96–102, 1995.

[154] K. Mehlhorn and M. Ziegelmann. Resource Constrained Shortest Paths. *8th Annual European Symposium on Algorithms (ESA)*, LNCS 1879:326–337, 2000.

[155] P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129(1–2):133–163, 2001.

[156] M. Milano. Integration of Mathematical Programming and Constraint Programming for Combinatorial Optimization Problems. *Tutorial at the 6th International Conference on Principles and Practice of Constraint Programming (CP)*, 2000.

[157] U. Montanari. Networks of constraints: fundamental properties and applications. *Information Science*, 7(2):95–132, 1974.

[158] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, 1988.

[159] G.L. Nemhauser, M.W.P. Savelsberg, and G.C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

[160] W.P.M. Nuijten and E.H.L. Aarts. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operational Research*, 90(2):269–284, 1996.

[161] A. Orda. Routing with end to end QoS guarantees in broadband networks. *Conference on Computer Communications (Infocom)*, IEEE, pp. 27–34, 1998.

[162] G. Ottosson and E.S. Thorsteinsson. Linear Relaxation and Reduced-Cost Based Propagation of Continuous Variable Subscripts. *2nd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, Paderborn Center for Parallel Computing, Technical Report tr-001-2000:129–138, 2000.

[163] PARROT. Executive Summary. ESPRIT 24 960, 1997.

[164] F. Pellegrini and J. Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *4th European Conference on High Performance Computing and Networking (HPCN)*, pp. 493–498, 1996.

[165] G. Pesant and M. Gendreau. A view of local search in constrained programming. *2nd International Conference on Principles and Practice of Constrained Programming (CP)*, LNCS 1118:353–366, 1996.

[166] S. Pettie and V. Ramachandran. Computing undirected shortest paths using comparisons and additions. *13th Symposium on Discrete Algorithms (SODA)*, 2002.

[167] D. Pisinger. An expanding-core algorithm for the exact 0-1 knapsack problem. *European Journal of Operational Research*, 87:175–187, 1995.

[168] D. Pisinger. An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114:528–541, 1999.

[169] S.A. Plotkin, D. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. of Operations Research*, 20:257–301, 1995.

[170] O. Porto, M. de Moraes, and A. Lucena. A relax and cut algorithm for the quadratic knapsack problem. *17th International Symposium on Mathematical Programming (ISMP)*, 2000.

[171] R. Preis and R. Dieckmann. The PARTY Partitioning Library — User Guide – Version 1.1. *Technical Report*, University of Paderborn, tr-rsfb-96-024, 1996.

[172] R. Preis and R. Diekmann. PARTY - A Software Library for Graph Partitioning. *Advances in Computational Mechanics with Parallel and Distributed Processing*, Civil-Comp Press, pp. 63–71, 1997.

[173] S. Prestwich. A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences. *6th International Conference on Principles and Practice of Constrained Programming (CP)*, LNCS 1894:337–352, 2000.

[174] J.-F. Puget. Symmetry Breaking Revisited. *8th International Conference on Principles and Practice of Constrained Programming (CP)*, LNCS 2470:446–461, 2002.

[175] T. Radzik. Fast deterministic approximation for the multicommodity flow problem. *6th Symposium on Discrete Algorithms (SODA)*, pp. 486–492, 1995.

[176] T. Radzik. Experimental study of a solution method for multicommodity flow problems. *2nd Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 79–102, 2000.

[177] G. Ramalingam and T. Reps. An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *Journal of Algorithms*, 21(2): 267–305, 1992.

[178] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1–2): 233–277, 1995.

[179] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. *12th National Conference on Artificial Intelligence*, AAAI, pp. 362–367, 1994.

[180] 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.

[181] E. Rothberg. Using Cuts to Remove Symmetry. *17th International Symposium on Mathematical Programming (ISMP)*, 2000.

[182] R.A. Rushmeier, K.L. Hoffman, and M. Padberg. Recent advances in exact optimization of airline scheduling problems. *Technical Report*, George Mason University, 1995.

[183] 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.

[184] M. Sato. Efficient implementation of an approximation algorithm for multicommodity flows. *Master Thesis*, Graduate School of Engineering Science, Osaka University, 2000.

[185] 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.

[186] SCIL. Symbolic Constraints in Integer Linear Programming. `http://www.mpi-sb.-mpg.de/SCIL/`.

[187] M. Sellmann. An Arc-Consistency Algorithm for the Weighted All Different Constraint. *8th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2470:744–749, 2002.

[188] M. Sellmann and T. Fahle. CP-Based Lagrangian Relaxation for a Multimedia Application. *3rd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, pp. 1–14, 2001.

[189] M. Sellmann and T. Fahle. Coupling Variable Fixing Algorithms for the Automatic Recording Problem. *9th Annual European Symposium on Algorithms (ESA)*, LNCS 2161:134–145, 2001.

[190] M. Sellmann and T. Fahle. Constraint Programming Based Lagrangian Relaxation for the Automatic Recording Problem. *Annals of Operations Research*, to appear.

[191] M. Sellmann and W. Harvey. Heuristic Constraint Propagation. *4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, pp. 191–204, 2002.

[192] M. Sellmann and W. Harvey. Heuristic Constraint Propagation. *8th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 2470:738–743, 2002.

[193] M. Sellmann, G. Kliewer, and A. Koberstein. Lagrangian Cardinality Cuts and Variable Fixing for Capacitated Network Design. *10th Annual European Symposium on Algorithms (ESA)*, LNCS 2461:845–858, 2002.

[194] M. Sellmann, K. Zervoudakis, P. Stamatopoulos, and T. Fahle. Integrating Direct CP Search and CP-based Column Generation for the Airline Crew Assignment Problem. *2nd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, Paderborn Center for Parallel Computing, Technical Report tr-001-2000:163–170, 2000.

[195] 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*, 115:207–225, 2002.

[196] B. Selman, H. Kautz, and D. McAllester. Ten Challenges in Propositional Reasoning and Search. *14th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 50–54, 1997.

[197] N. Sensen. Lower Bounds and Exact Algorithms for the Graph Partitioning Problem using Multicommodity Flows. *9th Annual European Symposium on Algorithms (ESA)*, LNCS 2161:391–403, 2001.

[198] F. Shahrokhi and D.W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37:318–334, 1990.

[199] F. Shahrokhi and L. Szekely. On Canonical Concurrent Flows, Crossing Number and Graph Expansion. *Combinatorics, Probability and Computing*, 3:523–543, 1994.

[200] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. *4th International Conference on Principles and Practice of Constraint Programming (CP)*, LNCS 1520:417–431, 1998.

[201] H.D. Sherali and J. Cole Smith. Improving Discrete Model Representation Via Symmetry Considerations. *17th International Symposium on Mathematical Programming (ISMP)*, 2000.

[202] B. Smith. Reducing Symmetry in a Combinatorial Design Problem. *3rd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, pp. 351–360, 2001.

[203] C. Souza, R. Keunings, L.A. Wolsey, and O. Zone. A new approach to minimising the frontwidth in finite element calculations. *Computer Methods in Applied Mechanics and Engineering*, 111:323–334, 1994.

[204] V. Sridhar and J.S. Park. Benders-and-cut algorithm for fixed-charge capacitated network design problems. *European Journal of Operational Research*, 125(3):622–632, 2000.

[205] P. Stamatopoulos, G. Boukeas, K. Zervoudakis, V. Stoumpos, and C. Halatsis. Parallel CP-based direct crew rostering. ESPRIT 24 960 (PARROT), University of Athens and University of Paderborn, Deliverable D-TEC2.1, 1999.

[206] A. Steger and N.C. Wormald. Generating random regular graphs quickly. *Combinatorics, Probability and Computation*, 8:377–396, 1999.

[207] M. Thorup. Undirected single source shortest paths in linear time. *38th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, pp. 12–21, 1997.

[208] TIVO. TV your way. TIVO, Inc., `http://www.tivo.com/home.asp`.

[209] M. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *3rd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, pp. 113–124, 2001.

[210] UP-TV. European Research Project, IST-1999-20751. `http://www.up-tv.de/`.

[211] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[212] 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, ES31-83, 1983.

[213] T. Walsh. Depth-bounded discrepancy search. *14th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1388–1393, 1997.

[214] C. Walshaw, M. Cross, and M. Everett. A localised algorithm for optimising unstructured mesh partitions. *International Journal of Supercomputer Applications and High Performance Computing*, 9(4):280–295, 1995.

[215] H.P. Williams. *Model Building in Mathematical Programming.* Wiley, 1978.

[216] G. Xue. Primal-dual algorithms for computing weight-constrained shortest paths and weight-constrained minimum spanning trees. *International Performance, Computing, and Communications Conference (IPCCC)*, IEEE, pp. 271–277, 2000.

[217] N. Young. Randomized rounding without solving the linear program. *6th Symposium on Discrete Algorithms (SODA)*, pp. 170–178, 1995.

[218] G. Yu (Editor). *Operations Research in the Airline Industry*. International Series in Operations Research and Management Science, Kluwer Academic Publishers, 1998.

[219] T.H. Yunes, A.V. Moura, and C.C. Souza. A hybrid approach for solving large crew scheduling problems. *International Workshop on Practical Aspects of Declarative Languages (PADL)*, LNCS 1753:293–307, 2000.