Conflict Graphs in Mixed-Integer Linear Programming: Preprocessing, Heuristics and Cutting Planes

Samuel Souza Brito
Universidade Federal de Ouro Preto

Orientador: Haroldo Gambini Santos

UNIVERSIDADE FEDERAL DE OURO PRETO

Conflict Graphs in Mixed-Integer Linear Programming: Preprocessing, Heuristics and Cutting Planes

Samuel Souza Brito
Universidade Federal de Ouro Preto

Orientador: Haroldo Gambini Santos

Tese submetida ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas e Biológicas da Universidade Federal de Ouro Preto, como requisito parcial para obtenção do título de Doutor em Ciência da Computação.

Ouro Preto, Fevereiro de 2020

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

B862c Brito, Samuel Souza .
Conflict graphs in mixed-integer linear programming [manuscrito]: preprocessing, heuristics and cutting planes. / Samuel Souza Brito. - 2020.
110 f.: il.: color., gráf., tab..

Orientador: Prof. Dr. Haroldo Gambini Santos.
Tese (Doutorado). Universidade Federal de Ouro Preto. Departamento de Computação. Programa de Ciência da Computação.
Área de Concentração: Ciência da Computação.

1. Programação linear. 2. Heurística. 3. Programação (Computadores) - Processamento. 4. Programação heurística. 5. Teoria dos grafos. I. Santos, Haroldo Gambini. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004.42

Bibliotecário(a) Responsável: Celina Brasil Luiz - CRB6-1589

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Samuel Souza Brito

Conflict Graphs in Mixed-Integer Linear Programming: Preprocessing, Heuristics and Cutting Planes

Membros da banca

Haroldo Gambini Santos - Dr. - UFOP
George Henrique Godim da Fonseca - Dr. - UFOP
Geraldo Robson Mateus - Dr. - UFMG
Marcus Vinicius Soledade Poggi de Aragao - Dr. - PUC-RIO
Tulio Angelo Machado Toffolo - Dr. - UFOP

Versão final
Aprovado em 28 de Fevereiro de 2020

De acordo

Prof. Dr. Haroldo Gambini Santos

Dedico este trabalho a minha esposa Tamara e a minha mãe Maria, pessoas de suma importância em minha vida.

Conflict Graphs in Mixed-Integer Linear Programming: Preprocessing, Heuristics and Cutting Planes

Abstract

This thesis addresses the development of conflict graph-based algorithms for Mixed-Integer Linear Programming, including: (*i*) an efficient infrastructure for the construction and manipulation of conflict graphs; (*ii*) a preprocessing routine based on a clique strengthening scheme that can both reduce the number of constraints and produce stronger formulations; (*iii*) a clique cut separator capable of obtaining dual bounds at the root node LP relaxation that are 19.65% stronger than those provided by the equivalent cut generator of a state-of-the-art commercial solver, 3.62 times better than those attained by the clique cut separator of the GLPK solver and 4.22 times stronger than the dual bounds obtained by the clique separation routine of the COIN-OR Cut Generation Library; (*iv*) an odd-cycle cut separator with a new lifting module to produce valid odd-wheel inequalities; (*v*) two diving heuristics capable of generating integer feasible solutions in restricted execution times. Additionally, we generated a new version of the COIN-OR Branch-and-Cut (CBC) solver by including our conflict graph infrastructure, preprocessing routine and cut separators. The average gap closed by this new version of CBC was up to four times better than its previous version. Moreover, the number of mixed-integer programs solved by CBC in a time limit of three hours was increased by 23.53%.

Keywords: Mixed-Integer Linear Programming, Conflict Graphs, Preprocessing, Cutting Planes, Clique Inequalities, Odd-cycle inequalities, Diving Heuristics.

Declaração

Esta tese é resultado de meu próprio trabalho, exceto onde referência explícita é feita ao trabalho de outros, e não foi submetida para outra defesa nesta nem em outra universidade.

Samuel Souza Brito

Agradecimentos

Agradeço a Deus por me dar saúde, disposição e calma para enfrentar os desafios desta caminhada.

Agradeço a minha esposa Tamara pelo carinho, companheirismo e apoio irrestritos. Por compartilhar alegrias, tristezas, esperanças e medos durante toda esta etapa.

Agradeço a minha mãe Maria pelo apoio incondicional e por ser minha fonte de incentivo e perseverança. Ao meu pai Elias, pela motivação durante essa jornada. Ao meu irmão Thalles, pelo apoio contínuo, conversas e palavras de incentivo.

Agradeço ao professor Haroldo pela confiança, paciência e dedicação. Pela impecável orientação durante uma década, desde a época da minha graduação, sempre acreditando em mim e em meu potencial. Por ter me proporcionado, durante todo esse tempo, um imensurável conhecimento, que vai além da simples formação acadêmica. Agradeço ainda pelas oportunidades concedidas.

Agradeço a Universidade Federal de Ouro Preto (UFOP), por prover educação pública, gratuita e de qualidade. Aos professores do Departamento de Computação (DECOM/UFOP), pelos ensinamentos transmitidos. Ao Departamento de Computação e Sistemas (DECSI/UFOP), por ter me permitido dedicar exclusivamente ao doutorado durante um ano. À Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG), pelo apoio financeiro concedido no estágio inicial deste trabalho.

Por fim, agradeço a todos que me ajudaram direta ou indiretamente neste trabalho.

Preface

If we turn from battle because there is little hope for victory, where then would valor be? Let it ever be the goal that stirs us, not the odds.

Silver Surfer in "The Silver Surfer: Parable",
Stan Lee and Moebius

Contents

List of Figures **xi**

List of Tables **xiii**

Nomenclature **1**

1 Introduction **2**

1.1 Objectives and Contributions 4

1.1.1 Published Papers and Conference Presentations 5

1.2 Text Organization 6

2 Background and Literature Review **7**

2.1 Combinatorial Optimization 7

2.2 Mixed-Integer Linear Programming 8

2.2.1 Preprocessing 9

2.2.2 Primal Heuristics 10

2.2.3 Branch-and-bound 12

2.2.4 Cutting Planes 13

2.2.5 Branch-and-cut 15

2.3 Conflict Graphs in Mixed-Integer Linear Programming 16

2.4 Literature Review 17

2.5 Instance Sets 23

3 Building Conflict Graphs **25**

3.1 Probing Technique Based on Feasibility Conditions 25

3.2 Fast Detection of Conflicts 27

3.2.1 Space Efficient Data Structures 30

3.2.2 Query Efficient Data Structures 32

3.3 Computational Results 35

3.4 Conclusion 38

4 Clique Strengthening **40**

4.1 Computational Results 44

4.2 Conclusion 47

5 Cutting Planes **48**

5.1 Clique Inequalities 49

5.1.1 Bron-Kerbosch Algorithm 51

5.2 Odd-Cycle Inequalities 54

5.3 Cut Pool 58

5.4 Computational Results 59

5.4.1 Pivoting Rules of Bron-Kerbosch Algorithm 59

5.4.2 Clique Cut Separator Experiments 61

5.4.3 Odd-Cycle Cut Separator Experiments 65

5.5 Conclusion 67

6 Improving the COIN-OR Branch-and-Cut Solver **68**

6.1 Computational Results 68

6.1.1 Individual Impact of Each Routine 69

6.1.2 Results of the New Version of CBC Solver 71

6.2 Conclusion 74

7 Diving Heuristics **75**

7.1 Conflict-Based Diving Heuristics 77

7.1.1 Conflict Diving 78

7.1.2 Modified Degree Diving 81

7.2 Computational Results 83

7.3 Conclusion 84

8 Final Considerations **86**

8.1 Further Research 87

Bibliography **89**

A Detailed Results of the Computational Experiments **94**

List of Figures

2.1 Improving the LP relaxation of (2.1) through the inclusion of the valid inequality $x_1 + x_2 \leq 4$. 14

2.2 An example of a conflict graph. 17

3.1 Filled data structures for the cliques of Table 3.1. 32

3.2 Execution times and memory usages in the construction of CGs. 37

4.1 Conflict graph for constraints (4.1) to (4.3). For practical purposes, vertices that have only trivial conflicts were omitted. 43

4.2 Results of the execution of clique strengthening. 46

5.1 Example of a K_3 in which the extension module could be applied, transforming it into a K_4. 51

5.2 Example of an odd cycle with the inclusion of a wheel center. Vertices x_6, x_7 and x_8 are connected to all the vertices of the odd cycle formed by $\{x_1, x_2, x_3, x_4, x_5\}$. 57

5.3 Execution times and gap closed by the clique cut separators. 63

5.4 Average gap closed by each clique cut separator. 64

5.5 Execution times and gap closed by the odd-cycle cut separator. 66

6.1 Results of the new version of CBC when each conflict-based routine is individually removed. 70

6.2 Execution times and gap closed by the two versions of CBC solver. . . . 72

6.3 Evolution of the average gap closed over time for each instance set. . . . 73

6.4 Number of instances solved over three hours. 74

7.1 Number of feasible solutions found by each diving heuristic over the time. 85

List of Tables

2.1 All possible logical relations between binary variables x_j and x_k. 16

2.2 A summary of related works. 22

2.3 Characteristics of the instance sets used in the experiments. 24

3.1 An example of cliques obtained after running Algorithm 3.1 in three constraints. 31

3.2 Instances where new conflicts were discovered. 39

5.1 Summarized results of the execution of BK algorithm with different pivoting rules. 60

7.1 Summarized results of the conflict-based diving heuristics. 83

A.1 Instance set characteristics. 94

A.2 Results of the execution of the COIN-OR Branch-and-Cut solver. 101

List of Algorithms

3.1 Clique Detection 28

3.2 Checking if two variables are conflicting. 34

3.3 Getting the variables conflicting with x_j. 35

4.1 Clique Extension 41

5.1 Clique Cut Separator 50

5.2 Bron-Kerbosch algorithm for detecting maximal cliques with weights above a threshold. 52

5.3 Odd-Cycle Cut Separator 55

7.1 Generic Diving Heuristic 76

7.2 Conflict Diving Heuristic 81

Nomenclature

BP	Binary Program
B&B	Branch-and-bound Algorithm
B&C	Branch-and-cut Algorithm
BK	Bron-Kerbosch Algorithm
CBC	COIN-OR Branch-and-Cut
CG	Conflict Graph
CGL	COIN-OR Cut Generation Library
CLP	COIN-OR Linear Program
GLPK	GNU Linear Programming Kit
LHS	Left-hand side
MILP	Mixed-Integer Linear Programming
RHS	Right-hand side
SAT	Boolean Satisfiability Problem

Chapter 1

Introduction

Over the last years, Mixed-Integer Linear Programming (MILP) has proven to be a powerful technique for modeling and solving a wide variety of combinatorial optimization problems, most of them with practical interest. Some notable applications include telecommunication network design (Orlowski et al., 2010), protein structure prediction (Xu et al., 2003) and production planning (Pochet and Wolsey, 2006).

Improvements in computer hardware and the development of several techniques such as preprocessing (Achterberg et al., 2016; Mészáros and Suhl, 2003; Savelsbergh, 1994), heuristics (Danna et al., 2005; Fischetti et al., 2005) and cutting planes (Hoffman and Padberg, 1993; Rebennack, 2009) have contributed toward large-scale MILP models being solved effectively. Preprocessing and cutting planes are part of a mechanism called automatic reformulation (Van Roy and Wolsey, 1987), which is a key component of modern MILP solvers. The works of Bixby and Rothberg (2007), Achterberg and Wunderling (2013), and more recently Achterberg et al. (2016) show that disabling these features in two state-of-the-art commercial solvers results in large performance degradation.

An implicit structure used by modern MILP solvers in preprocessing and cut separation routines is the conflict graph (Savelsbergh, 1994). Such graphs represent the logical relations between binary variables. There is a vertex for each binary variable and its complement, with an edge between two vertices indicating that the variables involved cannot both be equal to one without violating the constraints.

In this thesis, we present conflict graph-based algorithms and data structures for Mixed-Integer Linear Programming problems. Initially, we proposed and implemented a conflict graph infrastructure, characterized by the efficient construction and handling of such graphs. Our routine for building conflict graphs is an improved version of the conflict extraction algorithm presented by Achterberg (2007), which extracts conflicts from knapsack constraints. The basis for the improvement is a new step for detecting additional maximal cliques without increasing the computational complexity of the algorithm. We also developed optimized data structures that selectively store conflicts pairwise or grouped in cliques to handle dense conflict graphs without incurring excessive memory usage. The sequence in which similar cliques are discovered is exploited to store them compactly.

After developing the infrastructure for conflict graphs, we used the information provided by this structure to implement a preprocessing routine and two cut separators. The preprocessing routine is based on the concept of clique merging proposed by Achterberg et al. (2016) and consists of extending set packing constraints by the inclusion of additional conflicting variables. A greedy algorithm uses the information from the conflict graph to augment the cliques formed by the set packing constraints. After executing the clique extension algorithm, all constraints that become dominated are removed. Computational results show that our routine was able to reduce the number of constraints and strengthen the initial dual bounds for a great number of instances.

The two conflict-based cut separators that we developed are responsible for separating clique and odd-cycle cuts. Our clique cut separator is capable of obtaining dual bounds at the root node which are stronger than those provided by the clique cut separation routine of the COIN-OR Cut Generation Library (CGL)[1] and those obtained by the equivalent cut separators present in a state-of-the-art commercial MILP solver and the solver of GNU Linear Programming Kit (GLPK)[2]. The improvements in the dual bounds obtained by including only odd-cycle cuts were relatively small. However, the execution of the routine to separate odd-cycle cuts is computationally inexpensive, allowing its use in a cutting plane strategy without a significant increase in the execution times.

Our conflict graph infrastructure, preprocessing routine and cut separators were included in a new version of the COIN-OR Branch-and-Cut (CBC) solver[3]. CBC is one of the fastest open-source MILP solvers nowadays and it is also a fundamental component used by Mixed-Integer Nonlinear solvers, such as Bonmin (Belotti et al., 2009) and Couenne (Bonami et al., 2008). In our experiments, the average gap closed by the new

[1]`https://github.com/coin-or/Cgl`
[2]`https://www.gnu.org/software/glpk/`
[3]`https://github.com/coin-or/Cbc`

version of CBC was noticeably better than the previous version of this solver. Moreover, the time spent proving the optimality for the MILP models decreased and more instances were solved in restricted execution times.

Additionally, we proposed and implemented two conflict-based diving heuristics. These heuristics first adjust the bounds of the variables which are more likely to cause infeasibilities. In this case, one heuristic considers the degree and the other uses the modified degree of the variables at the conflict graph to implement the variable selection strategies. Both proposed diving heuristics presented execution times smaller than the classical diving heuristics that we evaluated in our experiments. Moreover, the heuristic that uses the modified degree in its variable selection strategy found the greatest number of feasible solutions among the heuristics evaluated.

1.1 Objectives and Contributions

The present thesis is motivated by the importance of MILP in solving a wide variety of combinatorial optimization problems. The development of techniques that improve the performance of MILP solvers contributes directly to solve different classes of problems, including real-world ones. Given this motivation, the main objective of this thesis is to develop conflict graph-based techniques to accelerate the process of solving MILP models.

The specific objectives of this thesis are:

1. Evaluate the performance of conflict-graph based techniques provided by MILP solvers, identifying possible improvements;
2. Propose and implement effective algorithms and data structures to construct, store and use conflict graphs;
3. Investigate, propose and implement automatic reformulation techniques that use the information provided from conflict graphs to reduce the MILP model dimensions, produce stronger formulations and accelerate the convergence to optimal solutions;
4. Explore the logical relations from conflict graphs to develop algorithms that are capable of generating integer feasible solutions.

In order to achieve the objectives, different approaches were developed. Thus, the main contributions of this thesis are:

1. An efficient infrastructure to construct, store and handle conflict graphs. Our algorithm for building conflict graphs is able to detect more conflicts than the state-of-the-art conflict detection algorithm, with the same worst-case complexity. Additionally, the data structures that we implemented are efficient to store and handle dense conflict graphs without incurring excessive memory usage.

2. An improved version of the Bron-Kerbosch algorithm for finding cliques in vertex-weighted graphs. Our version implements a new pivoting rule, defines a pruning strategy and uses efficient data structures to reduce the number of recursive calls and the running time of the algorithm.

3. A clique cut separator that generates a set of violated cliques. This cut separator obtained better dual bounds than the equivalent cut separators used by CBC, GLPK and CPLEX solvers.

4. A new strategy for lifting odd-cycle inequalities, which considers the inclusion of a clique into the center of an odd wheel.

5. A new version of CBC solver that contains our conflict graph infrastructure, pre-processing routine and cut separators. The average gap closed by this version was up to four times better than the previous version. Furthermore, the new version of CBC is capable of solving more problems than the previous one.

6. Two diving heuristics capable of generating integer feasible solutions in restricted execution times. These heuristics presented competitive results in comparison with some classical diving heuristics.

1.1.1 Published Papers and Conference Presentations

This thesis is a continuation of the research addressed by the author in his master's thesis (Brito, 2015). The following manuscripts, publications and presentations were derived from the obtained results.

- Brito, S. S.; Santos, H. G.; Poggi, M. *A Computational Study of Conflict Graphs and Aggressive Cut Separation in Integer Programming.* VIII Latin-American Algorithms, Graphs and Optimization Symposium (LAGOS15). May/2015. Beberibe, Brazil. DOI: 10.1016/j.endm.2015.07.059

- Brito, S. S.; Santos, H. G. *Improving COIN-OR CBC MIP Solver Using Conflict Graphs.* 23rd International Symposium on Mathematical Programming (ISMP2018). July/2018. Bordeaux, France.
- Brito, S. S.; Santos, H. G.; Vanden Berghe, G. *Machine Learning Based Diving for Mixed Integer Programming: Decision Trees.* 30th European Conference on Operational Research (EURO2019). June/2019. Dublin, Ireland.
- Brito, S. S.; Santos, H. G. *Preprocessing and Cutting Planes with Conflict Graphs.* Manuscript [4] submitted to Computers & Operations Research. September/2019.

1.2 Text Organization

This thesis is divided into eight chapters. Chapters where algorithms are proposed include computational experiments and specific analysis of these proposals. The remaining of the text is structured as follows:

Chapter 2: presents the basic concepts employed in this thesis, a literature review and the instance sets used in computational experiments;

Chapter 3: describes the probing technique for constructing conflict graphs as well as our conflict graph infrastructure;

Chapter 4: presents our conflict-based preprocessing routine;

Chapter 5: details the implementations of our clique and odd-cycle cut separators;

Chapter 6: presents the results obtained with the integration of our conflict graph-based algorithms and data structures in the CBC solver;

Chapter 7: presents the two diving heuristics that we proposed and implemented;

Chapter 8: concludes this thesis and presents possible future research directions.

[4] `https://arxiv.org/pdf/1909.07780.pdf`

Chapter 2

Background and Literature Review

This chapter presents concepts and techniques for understanding the construction and use of conflict graphs in Mixed-Integer Linear Programming. A literature review and the instance sets used in the computational experiments are also presented.

2.1 Combinatorial Optimization

Combinatorial Optimization is a field extensively studied by many researchers of Computer Science and Applied Mathematics. It aims to use combinatorial techniques to solve discrete optimization problems. A discrete optimization problem consists of finding the best possible solution from a finite set of possibilities. It works with deterministic models, where the relevant information is assumed to be known (without uncertainty). Examples of some classical combinatorial optimization problems are the traveling salesman problem (Applegate et al., 2006), project scheduling (Araujo et al., 2020), vehicle routing (Toth and Vigo, 2002) and timetabling problems (Fonseca et al., 2017).

A combinatorial optimization problem is formed by an objective function related to a set of decision variables. The objective function is a real-valued function that can be either minimized or maximized. The decision variables are limited by the constraints imposed on them, generating a discrete set of feasible solutions.

Due to its potential for modeling real-world problems, combinatorial optimization has significative advances over the last decades. Some techniques that can be used for solving such problems are Mixed-Integer Linear Programming (Jünger et al., 2009), Constraint Programming (Rossi et al., 2006), heuristics (Glover and Laguna, 1997a,b), approximation algorithms (Kolliopoulos and Young, 2005; Lenstra et al., 1990) and hybrid algorithms (e.g., Mixed-Integer Linear Programming combined with heuristic methods (Ahuja et al., 2002; Boschetti et al., 2009), Constraint Integer Programming (Achterberg, 2007), and others). This thesis focus on Mixed-Integer Linear Programming for solving combinatorial optimization problems.

2.2 Mixed-Integer Linear Programming

Mixed-Integer Linear Programming (MILP) deals with the minimization (or maximization) of a linear objective function subject to one or more linear constraints, where at least one of the decision variables can only assume integer values. A MILP model is formally defined as:

$$c^* = min\ \{c^T x \mid Ax \leq b,\ l \leq x \leq u,\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z}\ \forall j \in I\} \tag{MILP}$$

where $c \in \mathbb{R}^n$ represents the objective function coefficients, $A \in \mathbb{R}^{m \times n}$ is the constraint matrix and $b \in \mathbb{R}^m$ is the right-hand side (RHS) of the constraints. Vectors $l \in \mathbb{R}^n$ and $u \in \mathbb{R}^n$ are the lower and upper bounds for the decision variables, respectively. Furthermore, $N = \{1, ..., n\}$ is the index set of the decision variables x and $I \subseteq N$ contains the indices of the variables that need to be integral in every feasible solution. A MILP model whose all of its decision variables are binary (i.e., $0 \leq x_j \leq 1, x_j \in \mathbb{Z}, \forall j \in N$) is also called Binary Program (BP).

A feasible solution of a MILP model is a vector in the set:

$$X = \{x \in \mathbb{R}^n \mid Ax \leq b,\ l \leq x \leq u,\ x_j \in \mathbb{Z}\ \forall j \in I\}$$

A feasible solution $x^* \in X$ of a MILP model is optimal if its objective value satisfies $c^T x^* = c^*$. A lower bound on the optimal solution value of a MILP model can be obtained by solving its LP relaxation. The LP relaxation of a MILP model is obtained when the integrality requirements are omitted:

$$\check{c} = min \ \{c^T x \mid Ax \leq b, \ l \leq x \leq u, \ x \in \mathbb{R}^n\} \tag{LP}$$

This information is commonly used by the MILP solvers to explore promising nodes of the branch-and-bound search tree and to prove the optimality of a given solution.

There is extensive research in the field of MILP in order to develop effective and efficient solution methods. Some of the most common solution methods used in the literature are:

Branch-and-bound: consists of a systematic enumeration of the candidate solutions by evaluating smaller subproblems of the original problem;

Cutting plane: tries to iteratively refine a feasible set or objective function by including linear inequalities, referred to as cuts;

Branch-and-cut: combines a branch-and-bound algorithm with cutting planes;

Column generation: uses a decomposition scheme to create a master problem and subproblems; iteratively solves the subproblems and uses the reduced cost to select variables to add to the master problem; this process repeats until no more columns with attractive reduced cost exist;

Branch-and-price: combination of branch-and-bound and column generation;

Heuristics and Metaheuristics: use strategies based on the knowledge of the problem to quickly find feasible, and hopefully good, solutions without ensuring that an optimal (or even a feasible) solution is found.

2.2.1 Preprocessing

Preprocessing is an essential component in modern MILP solvers that reformulates a problem in an effort to accelerate the solution process. Several preprocessing strategies have been proposed in the literature and one of the precursors was the work of Brearley et al. (1975), which describes some preprocessing techniques for mathematical programming systems. Later, these techniques were inserted in the specific context of MILP.

A preprocessing component tries to reduce the dimension of a MILP model, strengthen its LP relaxation and extract information that can be used in other solving steps. The most common techniques are:

detection of inconsistent constraints: consists of discovering constraints that are inconsistent with respect to the bounds of the variables, proving the infeasibility of the problem;

elimination of redundant constraints: refers to the detection of constraints whose removal from the problem does not change the feasible region;

strengthening the bounds of variables: tries to increase the lower bound or decrease the upper bound of the variables.

coefficient improvement: updates the coefficients of the constraint matrix, improving its LP relaxation.

probing: refers to the identification of relationships between some variables by analyzing the impact of fixing them to one of their bounds.

Generally, a modern MILP solver employs some preprocessing techniques before starting the branch-and-bound algorithm. However, these techniques can also be applied in internal nodes of the branch-and-bound tree. In this case, the impact of node preprocessing is limited to its child nodes. Thus, it is important to evaluate whether the time spent in node preprocessing is worthwhile. The works of Savelsbergh (1994), Gamrath et al. (2015) and Achterberg et al. (2016) present detailed explanations of preprocessing techniques.

2.2.2 Primal Heuristics

In the context of MILP, primal heuristics are algorithms used to find feasible, and hopefully good, solutions in short execution times. However, there is no guarantee that these methods can find an optimal solution and, in some cases, even finding a feasible solution is a hard task.

Primal heuristics play an important role in a MILP solver, since obtaining a feasible solution in the early stages of the solving process has many advantages (Achterberg, 2007):

- it proves that the model is feasible;
- the solving process can be stopped earlier if one can be satisfied with the quality of the solution;
- it helps to prune nodes in the branch-and-bound search tree, improving the performance of this algorithm.

There exists a large variety of heuristics for MILP in the literature. One common strategy is to use information from the LP relaxation to decide the next step. A way to classify these methods is dividing them into four categories:

Rouding heuristics: try to round the solution values given by the LP relaxation, aiming to find an integer solution that satisfies the constraints of the problem; examples: Relaxation Enforced Neighborhood Search (RENS) (Berthold, 2006) and Octane (Balas et al., 2001);

Diving heuristics: iteratively solve the LP relaxation and fix an integer variable to an integral value. examples: Fractional Diving (Berthold, 2006) and Guided Diving (Danna et al., 2005);

Objective diving heuristics: iteratively solve the LP relaxation and change the objective value of an integer variable in order to drive this variable to a desired direction; example: Feasibility Pump (Fischetti et al., 2005);

Improvement heuristics: try to construct a better solution starting from an initial feasible one; examples: Local Branching (Fischetti and Lodi, 2003) and Relaxation Induced Neighborhood Search (RINS) (Danna et al., 2005).

In addition to the techniques mentioned above, metaheuristics can be applied to generate feasible solutions. A metaheuristic is a general framework that provides a set of strategies to develop heuristic optimization algorithms. Some examples of metaheuristics are Simulated Annealing (Kirkpatrick et al., 1983), Tabu Search (Glover and Laguna, 1998), Variable Neighborhood Search (Hansen and Mladenović, 1999) and evolutionary algorithms (Bäck et al., 1997). Instead of their general purpose, it is possible to design effective heuristics based on metaheuristic rules for solving MILP models, such as the Tabu Search heuristic presented by Lokketangen and Glover (1998) and the Variable Neighborhood Search heuristic proposed by Hansen et al. (2006).

2.2.3 Branch-and-bound

The branch-and-bound (B&B) algorithm was proposed by Land and Doig (1960) and is used to solve combinatorial optimization problems. It is a recursive divide-and-conquer approach that consists of conducting a search in the solution space for a given problem, aiming to find an optimal solution. The calculation of upper and lower bounds of the objective function allows the algorithm to search only a part of the solution space (Lawler and Wood, 1966).

The term *branch* refers to the fact that the method partitions the solution space, and the term *bound* emphasizes that the proof of optimality of the solution uses valid limits to prune some nodes of the search tree. This algorithm recursively divides the problem into smaller subproblems, based on the fact that the subproblems must be easier to solve than the original problem.

A generic branch-and-bound algorithm for solving a given MILP model with an objective function of minimization can be defined as:

```
1. Set $L = \infty$.
2. Insert the original problem on the candidate list.
3. Select and remove a problem $P$ from the candidate list.
4. Solve the LP relaxation of $P$ to obtain the bound $b(P)$.
    (a) If the LP relaxation of $P$ is infeasible, delete $P$.
    (b) Otherwise, if $b(P) \geq L$, delete $P$.
    (c) Otherwise, if $b(P) < L$ and the solution is feasible for the original MILP
        model, set $L = b(P)$.
    (d) Otherwise, divide $P$ into two or more subproblems and add them to the
        candidate list.
5. If the candidate list is empty, the algorithm finishes. Otherwise, go to step 3.
```

Three main steps compose a recursive call of a classical B&B algorithm: node selection, production of two or more subproblems (branching) and evaluation of the new subproblems (bounding). The algorithm maintains the current best integer solution and updates it whenever a subproblem gives a better integer solution. At each recursive call, a lower bound for the current subproblem is calculated by solving its LP relaxation. This subproblem is stored in the candidate list if its lower bound is better than the best solution. Otherwise, it is discarded. The search ends when there are no more subproblems to be processed. Thus, the best current solution is the optimal solution to the problem.

2.2.4 Cutting Planes

Instead of splitting a problem into smaller subproblems, one can try to tighten the LP relaxation of a problem to obtain a stronger one. The LP relaxation can be tightened by including linear constraints that are violated by the current solution of the LP relaxation but do not cut off feasible solutions from the original problem. These linear constraints are known as valid inequalities or cuts.

A cutting plane is a method that iteratively tries to generate and insert valid inequalities into a problem, allowing to tighten the LP relaxation of a MILP model. A generic cutting plane algorithm for solving a given MILP model can be defined as:

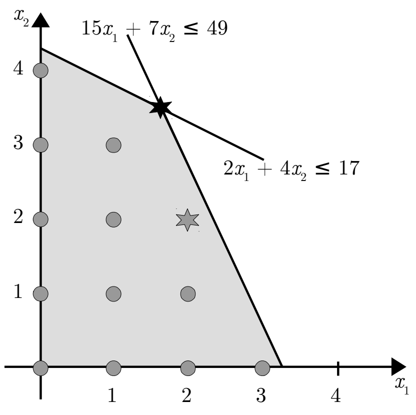
1. Solve the LP relaxation of the problem.
2. If the current LP relaxation is feasible for the original problem and the integrality conditions of the variables are satisfied, an optimal solution is found. Stop.
3. Otherwise, generate a linear constraint that is violated by the current LP relaxation but which does not cut off any feasible solution of the original problem. Go to step 1.

The insertion of valid inequalities "cuts" parts of a region that satisfies all constraints of a problem but does not lead to feasible integer solutions. The process of generating these inequalities is repeated until the LP relaxation is feasible and integrality constraints are still violated for one or more variables. When the cutting plane algorithm finishes, it returns an optimal solution for the considered MILP model.

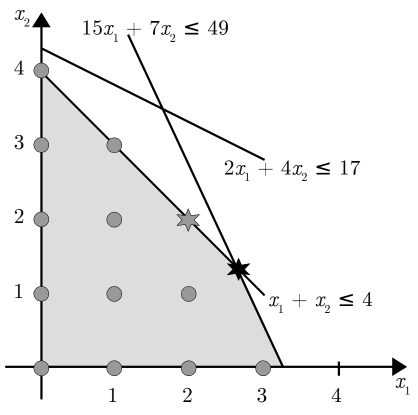
As an example, consider the following MILP model:

$$
\begin{aligned}
\text{min:} \quad & -6x_1 - 5x_2 \\
\text{subject to:} \quad & 15x_1 + 7x_2 \leq 49 \\
& 2x_1 + 4x_2 \leq 17 \\
& x_1, x_2 \geq 0 \\
& x_1, x_2 \in \mathbb{Z}
\end{aligned}
\tag{2.1}
$$

The gray area of Figure 2.1a corresponds to a feasible region of the LP relaxation of (2.1). Still, the stars in black and gray represent an optimal solution for the LP relaxation and for the original problem, respectively. Figure 2.1b shows the impact of adding the valid inequality $x_1 + x_2 \leq 4$ to (2.1).



(a) Initial LP relaxation of (2.1).



(b) LP relaxation of (2.1) after inserting a cut.

Figure 2.1: Improving the LP relaxation of (2.1) through the inclusion of the valid inequality $x_1 + x_2 \leq 4$.

The initial LP relaxation of (2.1) is $\check{c} = -27.109$, with $\check{x}_1 = 1.674$ and $\check{x}_2 = 3.413$. The inclusion of the valid inequality $x_1 + x_2 \leq 4$ allows to eliminate a region that do not contains any integer feasible solution. Thus, the value of the LP relaxation changes to $\check{c} = -22.625$, with $\check{x}_1 = 2.625$ and $\check{x}_2 = 1.375$. These values approximate to the optimal solution for the considered MILP model, whose objective value $c^* = -22$ is obtained from $x_1^* = 2$ and $x_2^* = 2$.

2.2.5 Branch-and-cut

Usually, solving MILP models with pure cutting plane methods shows a slow convergence to the optimal solution. On the other hand, the performance of the branch-and-bound algorithm can be considerably improved by the inclusion of cutting planes. The combination of these two approaches generates one of the most successful methods for solving MILP models: the branch-and-cut (B&C) algorithm. Most modern MILP solvers are based on this method.

A generic B&C algorithm for solving a given MILP model with an objective function of minimization can be defined as:

```
1. Set $L = \infty$.
2. Insert the original problem on the candidate list.
3. Select a problem $P$ from the candidate list.
4. Solve the LP relaxation of $P$ to obtain the bound $b(P)$.
    (a) If the LP relaxation of $P$ is infeasible, delete $P$.
    (b) Otherwise, if $b(P) \geq L$, delete $P$.
    (c) Otherwise, if $b(P) < L$ and the solution is feasible for the original MILP
        model, set $L = b(P)$.
    (d) Otherwise, try to generate valid inequalities for $P$.
        i. If at least one valid inequality is inserted in the LP relaxation of $P$,
           go to 4.
        ii. Otherwise, create new subproblems and add them to the candidate
            list.
5. If the candidate list is empty, the algorithm finishes. Otherwise, go to step 3.
```

In a branch-and-cut procedure, the branch-and-bound algorithm selectively explores the search space with its divide and conquer approach, and the cutting planes strengthen the bounds of the LP relaxations (step 4d). Heuristics can be performed when an integer solution is found (step 4c) to improve this solution or before separating cuts (step 4d) to generate a feasible solution.

The cutting planes can be carried out either at the initial LP relaxation or during the branching phase. Cuts generated by considering branching decisions are valid only in a local part of the branch-and-bound search tree. These cuts have to be removed from the LP relaxation after the search leaves the subtree for which they are valid. Globally cuts can be used during the whole execution of the algorithm. Since several cuts can be generated, maintain a cut pool is an essential task in the branch-and-cut execution.

The cut generation loop is the process of iteratively generating cuts, inserting them into the model and resolving the LP relaxation. Many executions of this process can be computationally expensive. For this reason, MILP solvers frequently define a maximum number of iterations of the cut generation loop.

2.3 Conflict Graphs in Mixed-Integer Linear Programming

A Conflict Graph (CG) is a structure that stores assignment pairs of binary variables which cannot occur in any feasible solution. It consists of an undirected graph with a set of vertices $V = \{x_j, \bar{x}_j \ : \ j = 1, ..., n\}$ and a set of edges $E = \{(u, v) \ : \ (u, v) \subset V^2\}$. In such structure, vertex x_j represents the assignment of the associated variable to one ($x_j = 1$), while vertex $\bar{x}_j$ corresponds to set the variable to zero ($x_j = 0$). Thus, the notation $\bar{x}_j$ is used to denote the binary complement of variable x_j (i.e., $\bar{x}_j = 1 - x_j$). The assignment pairs represented by the edges in a CG are used to derive logical relations and, consequently, the edge inequalities provided in Table 2.1.

Table 2.1: All possible logical relations between binary variables x_j and x_k.

edge	logical relations	edge inequality	
(x_j, x_k)	$x_j = 1 \implies x_k = 0$	$x_j + x_k$	≤ 1
	$x_k = 1 \implies x_j = 0$		
$(\bar{x}_j, \bar{x}_k)$	$x_j = 0 \implies x_k = 1$	$(1 - x_j) + (1 - x_k)$	≤ 1
	$x_k = 0 \implies x_j = 1$		
$(\bar{x}_j, x_k)$	$x_j = 0 \implies x_k = 0$	$(1 - x_j) + x_k$	≤ 1
	$x_k = 1 \implies x_j = 1$		
$(x_j, \bar{x}_k)$	$x_j = 1 \implies x_k = 1$	$x_j + (1 - x_k)$	≤ 1
	$x_k = 0 \implies x_j = 0$		

An example of a conflict graph with three variables $\{x_1, x_2, x_3\}$ is presented in Figure 2.2. There is an edge linking each variable to its complement since only one must be equal to one in any feasible solution. Dashed lines in Figure 2.2 denote these trivial conflicts. The conflict graph of this figure generates three edge inequalities:

$$\begin{aligned}
&x_1 + x_2 \leq 1 \\
&x_2 + \bar{x}_3 \leq 1 \Rightarrow x_2 - x_3 \leq 0 \\
&\bar{x}_2 + \bar{x}_3 \leq 1 \Rightarrow x_2 + x_3 \geq 1
\end{aligned}$$

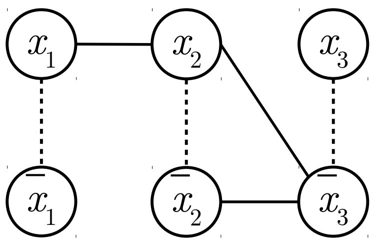


Figure 2.2: An example of a conflict graph.

Generally, a graph with only a subset of all conflict edges is constructed, since building the full CG is NP-hard. Deciding the feasibility of a binary program is NP-complete (Garey and Johnson, 1979), and this task can be done by constructing and analyzing the full CG: a binary program is infeasible if and only if its associated CG is a complete graph.

A clique is an important concept related to conflict graphs and employed in this thesis. A clique C of a graph G is a complete subgraph of G, i. e., each pair of vertices in C is connected by an edge. A maximal clique is a clique to which no more vertices can be added. In the graph of Figure 2.2, the subset of vertices $\{x_2, \bar{x}_2, \bar{x}_3\}$ defines a maximal clique of size three.

2.4 Literature Review

The primary use of CGs is in the generation of cutting planes. However, there are works in the literature that use these structures in different stages of the MILP solving process, such as in preprocessing steps. The following paragraphs present a literature review regarding the construction and use of conflict graphs and their variants.

A structure used to manipulate conflicts involving binary variables of MILP models is the intersection graph, introduced by Padberg (1973). An intersection graph is an undirected graph with a set of vertices $V = \{x_j \ : \ j = 1, ..., n\}$ and a set of edges $E = \{(u, v) \ : \ (u, v) \subseteq V^2\}$, constructed from the analysis of set packing constraints ($\sum_{j \in C} x_j \leq 1, \ x_j \in \{0, 1\} \ \forall j \in C$). There is an edge linking two vertices if their corresponding variables appear together in at least one set packing constraint. From this definition, it is possible to conclude that the concept of conflict graph is a generalization of the intersection graph: for a given MILP model, any edge of the intersection graph is also an edge of the conflict graph. The author uses these graphs to study and identify facets of the set packing polyhedron. The first set of facets identified for this polyhedron is formed by cliques, while the second set comprises the odd cycles without chords.

Johnson and Nemhauser (1992) summarize some advances in mathematical programming, including improvements in the MILP methodology. An approach to the detection of logical relations in knapsack constraints ($\sum_{j \in C} a_j x_j \leq b, \ a_j > 0, \ x_j \in \{0, 1\} \ \forall j \in C$) is given. Considering two particular variables x_j and x_k of a knapsack constraint, a logical relation $x_j + x_k \leq 1$ is detected if $a_j + a_k > b$, where b is right-hand side of this constraint and a_j and a_k are the coefficients of the variables x_j and x_k. The authors also point that these logical relations can be used to generate clique inequalities, contributing to improve the value of the LP relaxation of a MILP model.

Hoffman and Padberg (1993) construct and use intersection graphs for solving the airline crew scheduling problem. The intersection graphs are constructed similarly as presented by Padberg (1973). The authors present a branch-and-cut approach that uses these graphs to preprocess the problems and to generate clique and odd-cycle inequalities. The graph-based preprocessing routine tries to extend the cliques formed by set partitioning constraints ($\sum_{j \in C} x_j = 1, \ x_j \in \{0, 1\} \ \forall j \in C$). If the extension is successfully performed, then it is possible to remove some variables of the original problem. Clique inequalities are generated by a routine that combines heuristics and enumeration schemes. The generation of odd-cycle inequalities is done by a shortest path algorithm that runs in an auxiliary bipartite graph. Computational results show that the developed branch-and-cut is able to solve even large-size instances of the airline crew scheduling problem.

Aiming to improve the representation of MILP models, Savelsbergh (1994) presents a framework for describing preprocessing and probing techniques. This work gives an overview of simple and advanced preprocessing and probing techniques. Such techniques are used to derive logical implications between variables and, consequently, to build implication graphs. Different from the concept of logical relation explored before, that only considers the relation between two binary variables, a logical implication can also involve the relationship between binary and continuous variables or between binary and integer variables. The logical implications discovered during the execution of preprocessing and probing techniques are used to eliminate variables and generate clique and implication inequalities. Computational results demonstrate the effectiveness of these techniques in reducing the integrality gap and the overall effort required to solve most of the considered problem instances.

Bixby and Lee (1998) use information from the conflict graphs in a branch-and-cut algorithm for solving the truck dispatching scheduling problem. This problem is modeled as knapsack equality constraints ($\sum_{j \in C} a_j x_j = b$, $a_j > 0$, $x_j \in \{0, 1\}$ $\forall j \in C$), and the developed branch-and-cut algorithm is based on the ideas of Hoffman and Padberg (1993). The construction of the conflict graphs is given by the analysis of knapsack constraints, considering the same approach presented by Johnson and Nemhauser (1992). Clique and odd-cycle inequalities are generated from a subgraph induced by the variables whose values at the solution of the LP relaxation are fractional. Both cut generation routines are based on greedy heuristics. Experiments indicate that the branch-and-cut algorithm significantly reduces the total CPU time to solve some hard instances of the truck dispatching scheduling problem.

Borndorfer (1998) explores the concept of intersection graphs and presents a branch-and-cut algorithm for the solution of set partitioning problems. One initial application that uses intersection graphs is a preprocessing routine that eliminates binary variables by extending set partitioning constraints (the same idea was presented by Hoffman and Padberg (1993)). The developed cut separation routines work with subgraphs induced by the variables whose values at the solution of the LP relaxation are fractional. Clique inequalities are generated by three procedures. The first procedure is responsible for heuristically extending a clique extracted directly from a constraint. The second procedure uses a greedy strategy, which constructs a clique by iteratively selecting variables in non-decreasing order of their values at the solution of the LP relaxation. The last clique separator combines a branch-and-bound algorithm with a heuristic, avoiding the exploration of a large set of nodes. Odd-cycle inequalities are also separated during the execution of the branch-and-cut algorithm. This separator runs Dijkstra's algorithm in an auxiliary bipartite graph for finding the shortest paths and, consequently, discovering odd cycles from the intersection graph. Computational experiments show that the branch-and-cut algorithm is able to solve very large set-partitioning problems. Furthermore, the results demonstrate the importance of combining heuristics, preprocessing and cut separation routines for solving problems of this nature.

Atamtürk et al. (2000) use conflict graphs for improving the performance of integer programming solvers. The authors develop algorithms and data structures that allow the construction, management and use of dynamically changing conflict graphs. They construct conflict graphs from the detection of generalized upper bound constraints ($\sum_{j \in C} x_j \leq 1,\ x_j \in \{0,1\}\ \forall j \in C$) and using probing techniques based on feasibility and optimality considerations. A two-dimensional linked list structure is responsible for storing generalized upper bound constraints since all variables are conflicting in this type of constraint. In addition to saving memory, this structure supports fast checking of whether two variables are conflicting. Edges derived by probing are stored in a data structure composed of three one-dimensional arrays, which allows for easy addition of new edges and easy access to the edges incident to a given vertex. Once constructed and stored, the graphs are used in a preprocessing step to improve the lower and upper bound of the variables, and in a cut separation routine to generate clique inequalities. Computational results show that the proposed conflict graph storage and management are effective and efficient. Furthermore, the results confirm the importance of preprocessing and cut separation routines in solving integer programs.

Achterberg (2007) presents Constraint Integer Programming, a new paradigm that integrates Constraint Programming and MILP modeling and solving techniques. It is a generalization of MILP that supports the notion of general constraints as in Constraint Programming. The author also describes the software SCIP, a solver and framework for this new paradigm. An algorithm detects logical implications of a constraint integer program and stores them in an implication graph. Conflicts involving binary variables are discovered during the presolving step, where cliques are extracted from knapsack constraints. The clique extraction procedure is faster than the pairwise inspection performed by the probing techniques used by Savelsbergh (1994) and Atamtürk et al. (2000), since several conflicts are detected just traversing a constraint once. Conflicts involving several variables simultaneously are stored in a clique table to avoid excessive memory consumption. The implication graph is used to derive preprocessing and presolving algorithms, branching strategies and cut generation routines. A cut generation routine that uses information from the implication graph is the clique separator, which separates clique inequalities in a heuristic fashion. Several computational experiments are performed to measure the impact of each component in solving constraint integer programs. Furthermore, the results show that SCIP is almost competitive to current state-of-the-art commercial MILP solvers.

Santos et al. (2016) present some integer programming techniques to solve the nurse rostering problem. One of these techniques uses information from conflict graphs to generate valid inequalities. The authors construct conflict graphs by detecting generalized upper bound constraints and performing probing techniques. They also derive some implications from specific constraints of the problem. Then, these graphs are used to separate clique and odd-cycle cuts. The clique cut separator uses the Bron-Kerbosch algorithm (Bron and Kerbosch, 1973) with a pivoting rule to separate all violated cliques in the subgraph induced by the fractional variables. The odd cycles are separated by constructing an auxiliary bipartite graph and running the Dijkstra's algorithm. Computational experiments show that the clique cut separator plays an important role in reducing the gap between the LP relaxation and the optimal solution. However, the insertion of odd-cycle cuts has no significant impact on the reduction of this gap.

Following a similar strategy of the work mentioned before, Araujo et al. (2020) construct conflict graphs according to the analysis of problem-specific constraints. The authors derive four types of conflicts that appear in a MILP formulation for the resource-constrained project scheduling problem. Routines for separating conflict-based cuts are presented. The clique and odd-cycle cut separators are the same presented by Santos et al. (2016). They also present a routine that considers conflict graphs for generating strengthened Chvátal-Gomory cuts. Results show a considerable improvement in the LP relaxation bounds, allowing a state-of-the-art MILP solver to find optimal solutions for several open instances of the considered problem.

Table 2.2 presents a summary of the related works discussed above. Columns "pre-proc", "heur" and "cut gen" indicate if the related works use conflict graphs (or their variants) to implement preprocessing algorithms, heuristics or cut generation routines, respectively. The last line of Table 2.2 contains information about the use of conflict graphs in this thesis.

Table 2.2: A summary of related works.

work	graph type	construction method	preproc	heur	cut gen
Padberg (1973)	intersection graph	logical relations from set packing constraints			✓
Johnson and Nemhauser (1992)	conflict graph	logical relations from knapsack constraints			✓
Hoffman and Padberg (1993)	intersection graph	logical relations from set packing constraints	✓		✓
Savelsbergh (1994)	implication graph	probing techniques	✓		✓
Bixby and Lee (1998)	conflict graph	logical relations from knapsack constraints	✓		✓
Borndorfer (1998)	intersection graph	logical relations from set packing constraints	✓		✓
Atamtürk et al. (2000)	conflict graph	probing techniques	✓		✓
Achterberg (2007)	implication graph	probing techniques and extraction of cliques from knapsack constraints	✓		✓
Santos et al. (2016)	conflict graph	probing and logical relations from problem-specific constraints			✓
Araujo et al. (2020)	conflict graph	logical relations from problem-specific constraints			✓
this thesis	conflict graph	logical relations from knapsack constraints	✓	✓	✓

It is worth mentioning that there is a wide range of works in the literature that use conflict graphs in solving MILP models. The main application for conflict graphs is the generation of clique and odd-cycle inequalities. In fact, the process of separating these inequalities always requires the definition of an implicit or explicit graph. In general, the works in the literature that use conflict graphs employ one of the previously mentioned forms of construction of these structures.

In this thesis, the conflict graphs are constructed by extracting cliques from knapsack constraints. It is used the clique extraction algorithm described by Achterberg (2007) with the insertion of a new step that enables the detection of additional maximal cliques without increasing the computation effort. Optimized data structures for the conflict storage are also designed, allowing to handle dense conflict graphs without incurring excessive memory usage.

This thesis also presents conflict graph-based routines that contribute to solving MILP models: a preprocessing algorithm, two diving heuristics for obtaining feasible integer solutions and two cut separation routines. The cut separation routines are improved versions of those presented by Santos et al. (2016) and Araujo et al. (2020): the Bron-Kerbosch algorithm used in the clique separator has optimized data structures and a different pivoting rule, while the odd-cycle separator has a new lifting module.

2.5 Instance Sets

The instances used in the computational experiments of this thesis consist of 320 mixed integer programs found in the literature, most of which belong to the current and previous versions of the Mixed Integer Problem Library (MIPLIB) benchmark set (Gleixner et al., 2018). MIPLIB is a standard library of tests used to compare the performance of MILP solvers, containing a collection of challenging real-world instances from academic and industrial applications.

Our instance set also contains some classical problems of optimization such as Bin Packing with Conflicts (Sadykov and Vanderbeck, 2013), Nurse Rostering (Haspeslagh et al., 2014), Bandwidth Multicoloring Problem (Dias et al., 2016) and Educational Timetabling (Fonseca et al., 2017). Thus, the instances were divided into five instance sets:

bmc: instances of Bandwidth Multicoloring Problem;

bpwc: instances of Bin Packing Problem with Conflicts;

miplib: instances of MIPLIB;

rostering: instances of Nurse Rostering problem;

timetabling: instances of Educational Timetabling.

The objective function of all MILP models considered in this thesis are of minimization type. Table 2.3 contains summarized information concerning the instance sets. In this table, column "size" presents the number of instances of each instance set and "cols" contains the average number of variables. Columns "int", "bin" and "con" present, respectively, the average number of integer, binary and continuous variables of each instance set. Finally, columns "rows" and "nz" detail information with respect to the average number of constraints and nonzeros coefficients of each instance set.

Table 2.3: Characteristics of the instance sets used in the experiments.

group	size	cols	int	bin	con	rows	nz
bmc	9	15,606.33	0.00	15,605.33	1.00	398,899.89	813,363.11
bpwc	20	13,223.40	0.00	13,223.40	0.00	148,569.05	322,656.10
miplib	253	38,603.17	343.98	24,953.26	13,305.94	44,159.98	515,170.28
rostering	22	35,054.77	4.45	35,050.32	0.00	14,082.41	626,280.73
timetabling	16	20,768.25	10,828.75	9,939.50	0.00	40,902.94	159,929.25

We do not consider infeasible instances and instances which do not contain binary variables. Detailed information about the instance sets is presented in Table A.1 in the appendix.

Chapter 3

Building Conflict Graphs

CGs can be constructed using a probing technique based on feasibility considerations. This technique consists of tentatively setting binary variables to one of their bounds and checking whether the problem becomes infeasible as a result (Savelsbergh, 1994). Thus, the edges of CGs can be obtained by analyzing the impact of fixing pairs of variables to different combinations of values.

This chapter explains the probing technique presented by Atamtürk et al. (2000) and details a faster approach to construct CGs. For ease of presentation and understanding, the remainder of this chapter only considers binary programs. Despite this, all of the techniques presented can be applied to any MILP model containing binary variables.

3.1 Probing Technique Based on Feasibility Conditions

Suppose we are analyzing a constraint with the format:

$$\sum_{j \in B} a_j x_j \leq b, \tag{3.1}$$

where B is the index set of binary variables x with non-zero coefficients in this constraint. Suppose also that we are investigating the impact of fixing two binary variables x_p and

x_q to values v_1 and v_2, respectively. A valid lower bound for the left-hand side (LHS) of this constraint considering the assignments $x_p = v_1$ and $x_q = v_2$ is:

$$L^{x_p=v_1,x_q=v_2} = v_1 \cdot a_p + v_2 \cdot a_q + \sum_{j \in B^- \setminus \{p,q\}} a_j,$$

where B^- is the index set of variables with negative coefficients in the considered constraint. In this case, we consider the activation of the variables with negative coefficients to decrease the value of $L^{x_p=v_1,x_q=v_2}$ as much as possible. If $L^{x_p=v_1,x_q=v_2} > b$, then there is a conflict between the assignments of x_p and x_q. Thus, we insert the corresponding edge in the graph.

This computation is performed for each pair of variables in each constraint in order to obtain a CG. Therefore, given a MILP model with m constraints and n variables, its associated CG is constructed in $O(mn^2)$ steps. For this reason, probing may be computationally expensive for MILP models with a large number of variables and dense constraints. Nevertheless, for some constraint types, a large number of conflicts can be quickly discovered without having to conduct a pairwise inspection. For instance, in set packing and set partitioning constraints each variable has a conflict with all others, explicitly forming a clique. These constraints can be written as:

$$\sum_{j \in C} x_j \leq 1, x_j \in \{0,1\} \; \forall j \in C \qquad \text{(set packing)}$$

$$\sum_{j \in C} x_j = 1, x_j \in \{0,1\} \; \forall j \in C \qquad \text{(set partitioning)}$$

Set packing and set partitioning constraints often appear in MILP models to represent the choice of at most one (or exactly one) decision over a set of possibilities. As mentioned in Section 2.4, graphs that are constructed by only considering these constraints are denoted as intersection graphs.

Depending on the problem instance, intersection graphs can be very sparse, containing, for example, only trivial conflicts. In these cases, it may be necessary to execute an algorithm that analyzes other types of constraints and finds additional conflicts.

Considering the importance of CGs in solving MILP models and aiming to accelerate the process of building these structures, Achterberg (2007) developed a fast algorithm to extract cliques from constraints. We improved this algorithm by inserting an additional step that detects a higher number of maximal cliques. Additionally, we designed and implemented data structures that selectively store conflicts pairwise or grouped in cliques to handle dense CGs without incurring excessive memory usage. Details of our conflict graph infrastructure are given in the following section.

3.2 Fast Detection of Conflicts

One way to accelerate the construction of CGs is detecting conflicts involving several variables simultaneously without using the pairwise inspection scheme. Following this idea, Achterberg (2007) developed an algorithm that extracts cliques in less-structured constraints, that is, constraints that do not form a clique explicitly, by only traversing the constraint once. In addition to improve the process of building CGs, the early detection of cliques also allows for more efficient storage of the conflicts since explicit pairwise conflict storage can prove impractical for dense graphs. Thus, one can make use of special data structures where large cliques are not stored as multiple edges, like the one proposed by Atamtürk et al. (2000). Alternatively, graph compression techniques such as GraphZIP (Rossi and Zhou, 2018) could be employed to represent CGs succinctly.

We developed an improved version of the algorithm presented by Achterberg (2007) to construct CGs. This algorithm exploits the fact that any linear constraint involving only binary variables can be rewritten as a knapsack constraint similar to (3.1), with $b > 0$ and $a_j > 0$ for each j in the index set B of binary variables x. Sometimes, transformations on the linear constraints are necessary to rewrite them in this format: for a variable x_j with a negative coefficient a_j, we must consider the absolute value $|a_j|$, replace the variable by its complement $\bar{x}_j$ and update the RHS by adding $|a_j|$. For instance, the linear constraint $x_1 + x_2 - 2x_3 \leq 0$ can be rewritten as $x_1 + x_2 + 2\bar{x}_3 \leq 2$.

Algorithm 3.1 presents our strategy to detect cliques on a given knapsack constraint. The first step is to sort the index set of variables B in non-decreasing order of their coefficients. Next, we check if there are cliques in the constraint, by considering the activation of the two variables with the largest coefficients (line 2). If this assignment does not violate the RHS of the constraint, we can ignore the possibility of the existence of conflicts and the algorithm finishes (line 3). Otherwise, we perform a binary search to find the smallest k in B such that $a_{j_k} + a_{j_{k+1}} > b$ (line 5). Once we found the value of k, a clique C involving variables $\{x_{j_k}, x_{j_{k+1}}, ..., x_{j_n}\}$ is detected (line 6). This clique is then stored in clique set $\mathcal{S}$ (line 7) and the algorithm continues.

Algorithm 3.1: Clique Detection

```
   Input: Linear constraint $\sum_{j \in B} a_j x_j \leq b$.
   Output: Set of cliques $\mathcal{S}$.
1  Sort index set $B = \{j_1, ..., j_n\}$ by non-decreasing coefficient value $a_{j_1} \leq ... \leq a_{j_n}$;
2  if $a_{j_{n-1}} + a_{j_n} \leq b$ then
3      return $\emptyset$;
4  $\mathcal{S} \leftarrow \emptyset$;
5  Find the smallest $k$ such that $a_{j_k} + a_{j_{k+1}} > b$;
6  $C \leftarrow \{x_{j_k}, ..., x_{j_n}\}$;
7  $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$;
8  for $o = k - 1$ downto 1 do
9      Find the smallest $f$ such that $a_{j_o} + a_{j_f} > b$;
10     if $f$ exists then
11         $A \leftarrow \{x_{j_o}\} \cup \{x_{j_f}, ..., x_{j_n}\}$;
12         $\mathcal{S} \leftarrow \mathcal{S} \cup \{A\}$;
13     else
14         break;
15 return $\mathcal{S}$;
```

After finding C, the algorithm then attempts to detect additional maximal cliques. The strategy proposed by Achterberg (2007) consists of iteratively trying to replace the variable with the smallest coefficient in clique C by one of the variables outside C, maintaining the clique property. The disadvantage of this approach is that the additional cliques always differ in only one variable from the initial clique C. As such, cliques formed by a subset of variables of C and a variable outside C are not detected on the current constraint. This situation is solved using our new step for detecting additional maximal cliques, which occurs at lines 8 to 14 of Algorithm 3.1. For each variable at position o outside clique C, a binary search is performed to find the smallest f such that the assignment pair $(x_{j_o} = 1, x_{j_f} = 1)$ violates the constraint. If f exists, then an additional clique A formed by variable x_{j_o} and the subset $\{x_{j_f}, ..., x_{j_n}\}$ of C is detected and stored. The algorithm stops when the binary search finds no results. The failure to find a position f indicates that there are no additional cliques on the constraint since the coefficients are ordered.

Algorithm 3.1 detects and stores cliques in $O(n \log n)$ steps on a given constraint with

n variables. In this algorithm, sorting a constraint (line 1) is $O(n \log n)$, detecting an initial clique (line 5) is $O(\log n)$, storing an initial clique (6 to 7) is $O(n)$, and detecting and storing additional cliques (lines 8 to 14) is $O(n \log n)$. Thus, a conflict graph for a MILP model with m constraints and n variables is constructed in $O(mn \log n)$ steps, since we run Algorithm 3.1 for each constraint.

It is important to note that detecting and storing additional cliques would spend $O(n^2)$ steps if we explicitly store all the contents of these cliques. With this approach, we would have to iterate over all elements of a detected clique to store it. Consequently, the worst-case complexity of Algorithm 3.1 would be $O(n^2)$. However, any additional clique A that can be found by this algorithm is always formed by a subset C' of the first clique C and one variable outside C. For this reason, we implemented data structures that store only C completely. For each additional clique A, we store a tuple containing the variable outside C and the first position of C where the subset C' starts. Therefore, storing an additional clique is $O(1)$ and, consequently, the loop that extracts additional cliques (lines 8 to 14 of Algorithm 3.1) is $O(n \log n)$. Details about the data structures used to store conflict graphs are given in the next subsections.

Discarding the existence of cliques on a constraint at the first steps of the algorithm (lines 2 and 3) does not change its worst-case complexity. However, in practice, the execution time can be considerably reduced when the algorithm analyzes constraints with a large set of non-conflicting variables. The same effect occurs with the use of a binary search to detect the first clique (line 5), and the early termination of the algorithm when there are no additional cliques to be detected (lines 13 and 14). These simple mechanisms also represent a contribution to the algorithm proposed by Achterberg (2007).

The following example illustrates how our algorithm for detecting cliques on constraints works and compares the detected conflicts with the ones that could be found by the approach developed by Achterberg (2007).

Example Consider linear constraints:

$$\begin{aligned} -3x_1 + 4x_2 - 5x_3 + 6x_4 + 7x_5 + 8x_6 &\leq 2 \\ x_1 + x_2 + x_3 &\geq 1 \end{aligned}$$

where all variables are binary. The first step involves rewriting the constraints as knapsack constraints:

$$3\bar{x}_1 + 4x_2 + 5\bar{x}_3 + 6x_4 + 7x_5 + 8x_6 \leq 10 \tag{3.2}$$

$$\bar{x}_1 + \bar{x}_2 + \bar{x}_3 \leq 2 \tag{3.3}$$

Both constraints are already ordered according to their coefficients. We begin by analyzing constraint (3.2). First, we check for the existence of cliques in this constraint. When we activate the two variables with the largest coefficients ($x_5 = 1$ and $x_6 = 1$), we obtain $a_5 + a_6 = 7 + 8 = 15 > 10$. For this reason, we cannot discard the existence of cliques in this constraint. As such, we must now determine the smallest k such that $a_{j_k} + a_{j_{k+1}} > 10$. In this case, for $k = 3$ we have $a_3 + a_4 = 5 + 6 = 11$. Consequently, clique $C = \{\bar{x}_3, x_4, x_5, x_6\}$ is detected and stored. The next step consists of finding cliques involving variables $\bar{x}_1$ and x_2 outside C. For variable x_2, we perform a binary search that returns $f = 5$ since $a_2 + a_5 = 4 + 7 = 11 > 10$. Therefore, clique $\{x_2, x_5, x_6\}$ is detected. Finally, for $\bar{x}_1$ the binary search finds that $a_1 + a_6 = 3 + 8 = 11 > 10$, returning $f = 6$. Thus, clique $\{\bar{x}_1, x_6\}$ is also discovered. It is important to note that if we used the algorithm proposed by Achterberg (2007), cliques $\{x_2, x_5, x_6\}$ and $\{\bar{x}_1, x_6\}$ would not have been detected. As the last step, we analyze constraint (3.3). For this constraint we can discard the existence of cliques, since $a_2 + a_3 = 1 + 1 = 2 \leq 2$.

3.2.1 Space Efficient Data Structures

Data structures that efficiently store the cliques extracted from constraints are crucial in our algorithm. As mentioned before, explicitly storing all elements of all cliques extracted from a constraint increases the computational effort to construct CGs.

In Algorithm 3.1, after an initial clique C is found, any additional clique A is always a subset $C' \subset C$ plus a variable outside C. Moreover, given a clique $C = \{x_{j_k}, x_{j_{k+1}}, ..., x_{j_n}\}$, any subset of C that composes an additional clique A always has the form $C' = \{x_{j_l}, x_{j_{l+1}}, ..., x_{j_n}\}$, where $l > k$. Thus, an additional clique $A = \{x_{j_o} \cup C'\}$ can be represented by a tuple containing variable x_{j_o} and the first variable x_{j_l} of C where subset C' starts.

We use three arrays to store the extracted cliques. A two-dimensional array, referred to here as `first`, stores the initial cliques extracted from the constraints. Each row in `first` stores the elements of a clique. The array entry `size`$[c]$ contains the size of the c-th clique of `first`. The last array, denoted as `addtl`, stores the additional cliques. In this array, a clique is represented by a tuple of the form (x_o, c, l), which means that it is composed by variable x_o and all variables at positions l to `size`$[c]$ of the c-th clique of `first`.

Additionally, auxiliary arrays are used to store, for each variable, the indexes of `first` and `addtl` that contain cliques involving this variable. These structures are used to implement queries on the CG. The array entry `adjfirst`$[j]$ contains the indexes of cliques stored in `first` that involve variable x_j. The number of cliques in `first` that contain x_j is stored in `sizeaf`$[j]$. Arrays `adjaddtl` and `sizeaa` work in a similar way, but considering the cliques stored in `addtl`.

Figure 3.1 illustrates how our data structures work. This figure considers the cliques presented in Table 3.1, which contains an example of cliques that can be found by Algorithm 3.1. We start storing the first clique $\{x_3, x_4, x_5, x_6\}$ of constraint 1 at the first row of `first`. Then, the additional cliques of constraint 1 are converted in tuples and inserted in `addtl`:

- clique $\{x_2, x_5, x_6\}$ is converted on tuple $(x_2, 1, 3)$, since it is composed by variable x_2 and subset $\{x_5, x_6\}$, which starts at index 3 of the first clique of `first`;
- clique $\{x_1, x_6\}$ is converted on tuple $(x_1, 1, 4)$, since it is composed by variable x_1 and subset $\{x_6\}$, which starts at index 4 of the first clique of `first`.

Table 3.1: An example of cliques obtained after running Algorithm 3.1 in three constraints.

constraint	first clique	additional cliques
1	$\{x_3, x_4, x_5, x_6\}$	$\{x_2, x_5, x_6\}$
		$\{x_1, x_6\}$
2	$\{x_2, x_6, x_8\}$	
3	$\{x_4, x_6, x_8, x_9, x_{10}\}$	$\{x_3, x_6, x_8, x_9, x_{10}\}$
		$\{x_2, x_6, x_8, x_9, x_{10}\}$
		$\{x_1, x_9, x_{10}\}$

`first`

	1	2	3	4	5
1	x_3	x_4	x_5	x_6	
2	x_2	x_6	x_8		
3	x_4	x_6	x_8	x_9	x_{10}

`size`

1	4
2	3
3	5

`addtl`

1	(x_1, 1, 3)
2	(x_1, 1, 4)
3	(x_3, 3, 2)
4	(x_2, 3, 2)
5	(x_1, 3, 4)

`adjfirst`

	1	2	3
1			
2	2		
3	1		
4	1	3	
5	1		
6	1	2	3
7			
8	2	3	
9	3		
10	3		

`sizeaf`

1	0
2	1
3	1
4	2
5	1
6	3
7	0
8	2
9	1
10	1

`adjaddtl`

	1	2	3	4
1	2	5		
2	1	4		
3	3			
4				
5	1			
6	1	2	3	4
7				
8	3	4		
9	3	4	5	
10	3	4	5	

`sizeaa`

1	2
2	2
3	1
4	0
5	1
6	4
7	0
8	2
9	3
10	3

Figure 3.1: Filled data structures for the cliques of Table 3.1.

Following, clique $\{x_2, x_6, x_8\}$ of constraint 2 is stored at the second row of `first`. This constraint does not have additional cliques. Finally, clique $\{x_4, x_6, x_8, x_9, x_{10}\}$ of constraint 3 is stored at the third row of `first` and the three additional cliques of this constraint are stored in `addtl`:

- $\{x_3, x_6, x_8, x_9, x_{10}\}$ is stored as $(x_3, 3, 2)$;
- $\{x_2, x_6, x_8, x_9, x_{10}\}$ is stored as $(x_2, 3, 2)$;
- $\{x_1, x_9, x_{10}\}$ is stored as $(x_1, 3, 4)$.

3.2.2 Query Efficient Data Structures

The use of the data structures previously presented allows reducing the computational effort required to build CGs. It not only accelerates the construction process but also decreases the memory required to store a graph. However, the cost of making queries in these data structures increases according to the number of cliques stored by it. For example, the worst case of a query that returns all variables conflicting with a given variable occurs when we iterate over all cliques of a graph. The execution of this query in a graph with a large number of cliques could spend considerable time. This query would be faster if we use adjacency lists for each variable, explicitly storing all conflicting variables. In contrast, the use of adjacency lists increases the memory consumption and the computational effort required to build CGs, especially for the dense ones. Hence, there is a tradeoff between the computational effort and memory requirements to build and store a conflict graph, and the performance of querying it.

We implemented a hybrid solution that tries to limit the memory consumption of the conflict graph without significantly affecting the time spent to construct and query this structure. This solution uses the data structures presented before and maintains an adjacency list for each vertex. The array of adjacency lists is referred to here as `adjlist` and each array entry `adjlist`$[j]$ contains a set of variables conflicting with variable x_j. The adjacency list of each vertex is kept sorted so that queries in it can be performed in $O(\log n)$. A parameter *minClqSize* controls how the cliques are stored. After creating a conflict graph, we iterate over the cliques in `first` and `addtl` and remove those whose sizes are less than or equal to *minClqSize*. These small cliques are now stored as multiple pairs of conflicts in the adjacency lists of the vertices involved. Thus, large cliques are explicitly stored in `first` and `addtl`, while the small cliques are stored as multiple pairs of conflicts in `adjlist`.

Checking if two variables are conflicting is a query that frequently appears in conflict graph-based routines. Algorithm 3.2 implements this query method. Two variables x_j and x_k are conflicting if x_k appears in the adjacency list of x_j (or vice-versa) or if they appear together in at least one clique. First, we perform a binary search to test if x_k exists in `adjlist`$[j]$(lines 1 and 2). If `adjlist`$[j]$ contains x_k, variables x_j and x_j are conflicting and the algorithm finishes. Otherwise, the algorithm iterates over the cliques of `first` (lines 3 to 6) and `addtl` (lines 7 to 15) that contain x_j. At each iteration, if the current clique contains variable x_k, then x_j and x_k are conflicting and the algorithm finishes returning TRUE. Otherwise, the algorithm continues iterating over the cliques that contain x_j. After iterating over all these cliques and finding none of them that also contains x_k, the algorithm return FALSE, indicating that x_j and x_k are not conflicting. When iterating over array `addtl`, the algorithm has to convert each tuple (x_o, c, l) in a clique (lines 10 to 13). This clique is formed by variable x_o and the variables in array entries $\{first[c][l], first[c][l+1], ..., first[c][size[c]]\}$.

Algorithm 3.2: Checking if two variables are conflicting.

```
Input: Variables $x_j$ and $x_k$.
Output: TRUE if $x_j$ and $x_k$ are conflicting, or FALSE otherwise.
1  if $adjlist[j]$ contains $x_k$ then
2      return TRUE;
3  for $i = 1$ to $sizeaf[j]$ do
4      $p \leftarrow adjfirst[j][i]$;
5      if $first[p]$ contains $x_k$ then
6          return TRUE;
7  for $i = 1$ to $sizeaa[j]$ do
8      $p \leftarrow adjaddtl[j][i]$;
9      $(x_o, c, l) \leftarrow addtl[p]$;
10     $A \leftarrow \{x_o\}$;
11     while $l \leq size[c]$ do
12         $A \leftarrow A \cup \{first[c][l]\}$;
13         $l \leftarrow l + 1$;
14     if $A$ contains $x_k$ then
15         return TRUE;
16 return FALSE;
```

Another query method that is frequently employed in routines based on CGs is the one that returns all variables conflicting with a given variable x_j. Algorithm 3.3 presents an implementation of this query method considering our data structures. First, it gets the conflicting variables stored in the adjacency list of x_j (line 1). Then, it uses the auxiliary arrays to iterate over the cliques of `first` (lines 2 to 6) and `addtl` (lines 7 to 15) that contain x_j. At each iteration, the elements of the current clique, except x_j, are inserted in Q. The process of decoding a tuple (x_o, c, l) is the same as previously presented for Algorithm 3.2.

Algorithm 3.3: Getting the variables conflicting with x_j.

```
Input: Variable $x_j$.
Output: A set $Q$ of variables conflicting with $x_j$.
1  $Q \leftarrow adjlist[j]$;
2  for $i = 1$ to $sizeaf[j]$ do
3      $c \leftarrow adjfirst[j][i]$;
4      for $l = 1$ to $size[c]$ do
5          if $first[c][l] \neq x_j$ then
6              $Q \leftarrow Q \cup \{first[c][l]\}$;
7  for $i = 1$ to $sizeaa[j]$ do
8      $p \leftarrow adjaddtl[j][i]$;
9      $(x_o, c, l) \leftarrow addtl[p]$;
10     if $x_o \neq x_j$ then
11         $Q \leftarrow Q \cup \{x_o\}$;
12     while $l \leq size[c]$ do
13         if $first[c][l] \neq x_j$ then
14             $Q \leftarrow Q \cup \{first[c][l]\}$;
15         $l \leftarrow l + 1$;
16 return $Q$;
```

The queries in our conflict graph infrastructure are efficient when most of the cliques are stored as multiple pairs of conflicts in the adjacency list of the vertices. In fact, for typical instance problems of MIPLIB (Gleixner et al., 2018), the queries in the conflict graphs are faster when we set $minClqSize = 512$. In these conflict graphs, just a small set of conflicts are explicitly stored as cliques.

3.3 Computational Results

A computational experiment was conducted to compare the performance of our algorithm for building CGs, named as *ICE*, against the pairwise inspection scheme of Atamtürk et al. (2000), referred to here as *PI*, and the clique extraction algorithm of Achterberg (2007), denoted as *CE*. This experiment was carried out on four computers with Intel Core i7-4790 3.60 GHz processors and 32 GB of RAM running Ubuntu Linux version 18.04 64-bit. The source code was developed in C++ programming language and compiled with g++ version 7.4.0.

The algorithms for building conflict graphs were evaluated concerning the execution times and memory usages. *PI* and *CE* were implemented according to the descriptions given in their respective works. Since *PI* only detects pairs of conflicts, we use an adjacency list for each vertex of the graph. The conflict storage of *CE* uses the same data structures employed in our algorithm.

In this experiment, *PI* failed to construct graphs for eight instances: *eilA101-2*, *eilB101.2*, *eilD76.2*, *nw04*, *s100*, *square41*, *square47* and *supportcase6*. *PI* needed more than 32 GB of memory to construct and store CGs for these instances. They have some set packing and set partitioning constraints formed by a large number of variables, whose pairwise storage of conflicts results in excessive memory consumption. Since the other algorithms can explicitly store cliques, they did not face memory issues with respect to these instances. We penalize the cases where *PI* cannot construct CGs due to memory limitation, assigning for each affected instance a memory usage of 32 GB and an execution time of $1,800$ seconds.

Figure 3.2 shows the memory usage and time spent in constructing CGs for each algorithm and each instance set. The algorithms presented similar memory usages in instances of *bmc* and *timetabling*. Although the execution times are less than 1 second for these instances, *CE* and *ICE* obtained values that are smaller than *PI*. Complete results of this experiment are available for download at `http://professor.ufop.br/samuelbrito/thesis`.

Memory usages and execution times are similar in instance set *bpwc*, except for instance *uELGN_BPWC_3_2_18*. In this instance, *ICE* found about 8.7 million additional conflicts. The higher number of conflicts detected implied a greater consumption of time and memory.

The greatest performance gain with the use of the clique extraction approach and our data structures was obtained in *miplib* set. For some instances of this set, several cliques were detected and explicitly stored, contributing to the significative reduction of execution time and memory consumption on the construction of CGs. For example, *PI* needs more than 32 GB of memory to construct a graph for instance *eilD76.2*, while *CE* and *ICE* were able to build graphs for the same instance using only 52.10 MB. The results obtained in instance set *rostering* also demonstrates that *CE* and *ICE* are more efficient than *PI* in terms of memory consumption and time spent to create CGs.

In general, the combination of the strategy to avoid analyzing constraints that would not lead to the discovery of conflicts with the efficient clique extraction approach and the use of our optimized data structures contributed to decrease the amount of time

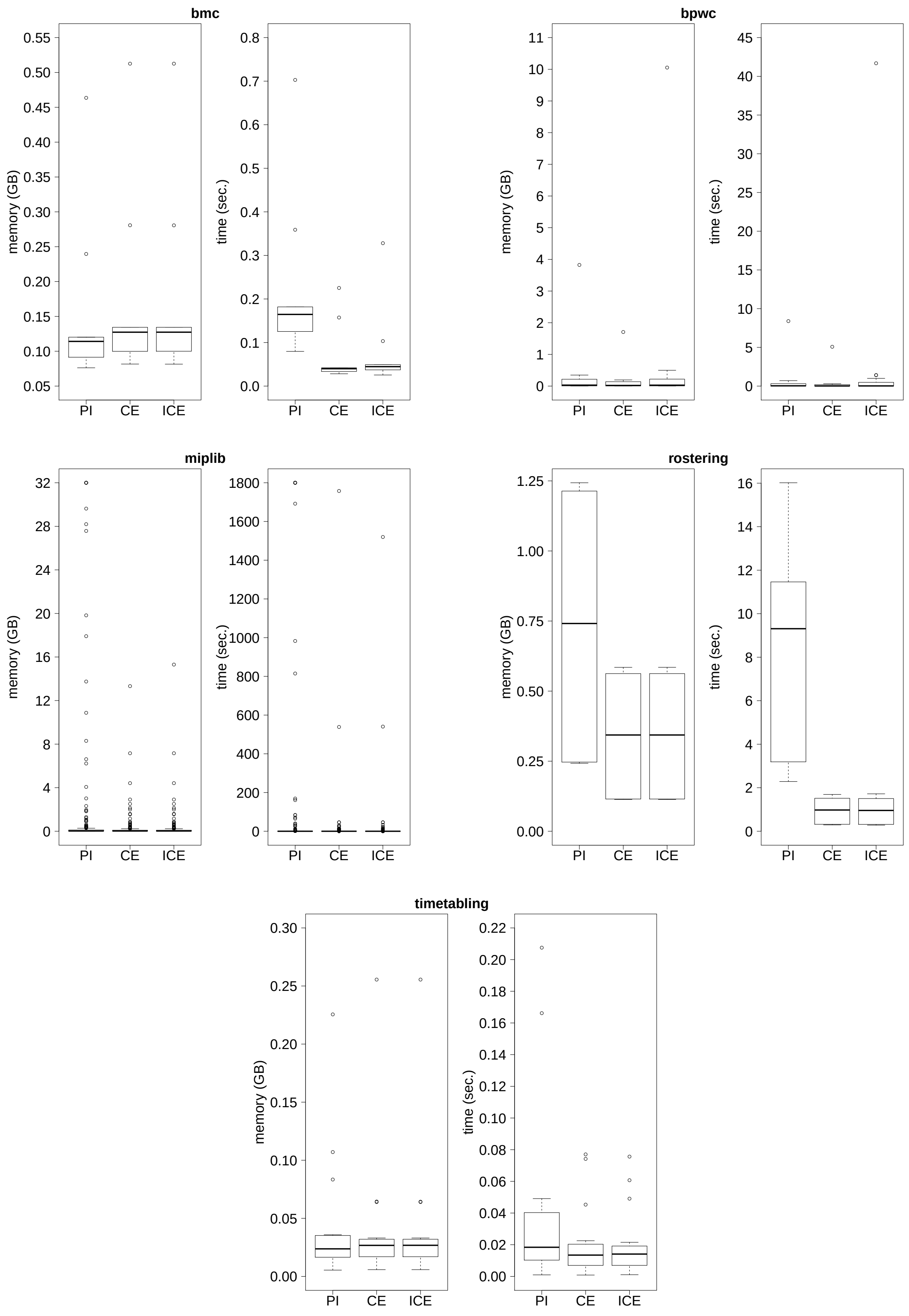


Figure 3.2: Execution times and memory usages in the construction of CGs.

and memory required to construct CGs. *ICE* built and stored CGs for all considered instance problems spending, on average, 252.47 MB of memory and 7.60 seconds. In comparison with *PI*, these results represent a decrease of 85.66% in memory consumption and 87.20% in the execution time.

The execution times and memory consumption of *CE* and *ICE* were similar, except in instances where *ICE* found more conflicts. Table 3.2 presents 27 instances in which *ICE* found more conflicts than *CE*, quantifying the improvement obtained. The number of new conflicts refers to the number of new edges that were included in the conflict graphs.

Our algorithm detected more conflicts than *CE* in constraints whose minimum and maximum coefficients of the variables are different and the highest coefficients are close to the RHS of the constraints. Instances of Bin Packing Problem with Conflicts use several constraints with this characteristic to model the capacity of the bins. It is the case of instance *uELGN_BPWC_3_2_18*, where more than 8.7 million of new edges were inserted in the associated CG. The highest increase of conflicts, in percentage, was obtained in instance *istanbul-no-cutoff*, in which *ICE* is responsible for detecting 127.78% more conflicts.

3.4 Conclusion

In this chapter, we presented our infrastructure for the efficient construction, storage and handling of conflict graphs. The routine for building these graphs is an improved version of the state-of-the-art conflict extraction algorithm. It is capable of detecting additional maximal cliques without increasing the computational complexity of the algorithm. Our optimized data structures selectively store conflicts pairwise or grouped in cliques, allowing to handle dense conflict graphs without incurring excessive memory usage. In these data structures, the sequence in which similar cliques are discovered is exploited to store them compactly.

Table 3.2: Instances where new conflicts were discovered.

instance	group	CE	ICE	new conflicts	% increase
n2seq36q	miplib	20,653,320	20,653,344	24	< 0.01
p0548	miplib	920	980	60	6.52
istanbul-no-cutoff	miplib	72	164	92	127.78
p2756	miplib	5,640	5,732	92	1.63
ua_BPWC_1_9_2	bpwc	13,506	13,718	212	1.57
ta_BPWC_6_9_8	bpwc	13,420	13,868	448	3.34
ta_BPWC_5_7_4	bpwc	12,876	13,474	598	4.64
ta_BPWC_5_7_1	bpwc	12,736	13,440	704	5.53
ua_BPWC_1_8_10	bpwc	48,532	49,798	1,266	2.61
ta_BPWC_5_5_5	bpwc	31,198	33,826	2,628	8.42
tELGN_BPWC_6_8_9	bpwc	28,542	31,598	3,056	10.71
uMIMT_BPPC_2_9_1	bpwc	58,716	64,978	6,262	10.66
neos-631694	miplib	377,746	385,368	7,622	2.02
tELGN_BPWC_6_6_20	bpwc	78,628	87,642	9,014	11.46
uELGN_BPWC_3_9_18	bpwc	253,020	281,838	28,818	11.39
neos-631784	miplib	7,918,996	7,966,118	47,122	0.60
neos-662469	miplib	2,401,182	2,460,118	58,936	2.45
tMIMT_BPPC_6_3_4	bpwc	542,740	629,266	86,526	15.94
supportcase18	miplib	1,913,864	2,023,850	109,986	5.75
tELGN_BPWC_7_6_16	bpwc	1,106,790	1,241,972	135,182	12.21
neos-631709	miplib	18,519,540	18,777,268	257,728	1.39
uMIMT_BPPC_2_5_2	bpwc	2,636,516	2,978,096	341,580	12.96
tMIMT_BPPC_8_7_5	bpwc	3,960,040	4,469,682	509,642	12.87
uMIMT_BPPC_3_7_6	bpwc	4,177,466	4,713,606	536,140	12.83
ta_BPWC_7_1_8	bpwc	5,488,238	6,404,428	916,190	16.69
neos-631710	miplib	129,069,680	130,707,306	1,637,626	1.27
uELGN_BPWC_3_2_18	bpwc	53,896,940	62,638,998	8,742,058	16.22

Chapter 4

Clique Strengthening

Preprocessing is an essential component in MILP solvers that can modify the structure of a MILP model to produce a stronger formulation. Stronger formulations usually have tighter dual bounds, which makes the branch-and-bound process more effective. Thus, a preprocessing component may accelerate the solution process and enable the early detection of infeasible problems.

There are several preprocessing strategies proposed in the literature. One of the precursors of these strategies was the work of Brearley et al. (1975), which describes techniques for mathematical programming systems that reduce the problem dimension by fixing variables, removing redundant rows, replacing constraints by simple bounds and more. Savelsbergh (1994) presented a framework for describing preprocessing and probing techniques, providing an overview of simple and advanced techniques to improve the representation of MILP models. More recently, Gamrath et al. (2015) developed three preprocessing techniques that were included in the non-commercial solver SCIP, and Achterberg et al. (2016) described the preprocessing strategies implemented in the commercial solver Gurobi.

One of the preprocessing strategies developed by Achterberg et al. (2016) and included in Gurobi is called *clique merging*. It consists of combining several set packing constraints into a single inequality. We based on this algorithm to develop a preprocessing routine that extends set packing constraints instead of combining them. We consider the whole conflict graph to extend each one of these constraints. Thus, variables that do not appear in other set packing constraints can be included in an extended constraint.

First, we create a set $\mathcal{C}$ containing all cliques formed by the set packing constraints of a given MILP model. Then, we try to extend each clique C in $\mathcal{C}$ using Algorithm 4.1.

Algorithm 4.1: Clique Extension

```
Input: Conflict graph $G = (V, E)$, clique $C$ and score function $S$.
Output: Extended clique $C'$.
1 Let $d$ be the vertex in $C$ with the smallest degree;
2 $L \leftarrow \{k \in N_G(d) \mid k \notin C\}$;
3 $C' \leftarrow C$;
4 while $L \neq \emptyset$ do
5     Let $l$ be the vertex in $L$ with the largest score $S(l)$;
6     Remove $l$ from $L$;
7     if $\exists l \in N_G(k)\ \forall k \in C'$ then
8         $C' \leftarrow C' \cup \{l\}$;
9 return $C'$;
```

Algorithm 4.1 is based on a greedy strategy that uses the information from a conflict graph $G = (V, E)$ and a score function S to add variables in clique C. Initially, a set L of candidate vertices for inclusion in clique C is constructed by selecting all neighbors of a vertex d that are not contained in C. Vertex d is the one with the smallest degree between the vertices of C. Notation $N_G(d)$ is used to represent the vertices in graph G that are adjacent to vertex d. Next, we create a set C' that initially contains all vertices of C. Then, we try to insert additional vertices in C' by iteratively selecting the vertex $l \in L$ with the largest score $S(l)$. Vertex l is inserted in C' only if it is adjacent to all vertices in C' (lines 7 and 8). The algorithm finishes when L is empty, returning the extended clique C'.

The score function S that is used to select vertices in Algorithm 4.1 can be defined in several ways. We use information from the MILP model or from the conflict graph associated with it to implement three variations of this function:

deg: returns the degree of a vertex.

mdg: returns the modified degree of a vertex, which is the sum of its degree and the degrees of all vertices adjacent to it.

rc: returns the reduced cost of the corresponding variable, which is the amount of penalty that would be generated if one unit of this variable was introduced into the solution. This version requires previously solving the LP relaxation of the considered problem.

The choice of the score function is made before the execution of the clique extension algorithm. Since the variables with the largest score are the best candidates to enter the clique, version rc of the score function has to be modified to return a value that is inversely proportional to the reduced cost. Thus, we consider that the variables with the smallest reduced costs have the largest scores.

After obtaining clique C', we generate the corresponding extended set packing constraint and insert it into the MILP model. Finally, a dominance checking procedure is performed to remove all constraints that are dominated by this extended constraint (Achterberg et al., 2016). In this context, a constraint i' dominates another constraint i if the corresponding clique of i is a subset of the clique formed by i'.

Our clique strengthening routine is especially effective when applied to MILP models that have several constraints expressed by pairs of conflicting variables. However, it can be computationally expensive, especially for problems with dense CGs and constraints with a large number of variables. For this reason, we limit the execution of the preprocessing routine to constraints with at most α_{max} variables, where α_{max} is an input parameter of the algorithm.

The following example illustrates the execution of the clique strengthening process.

Example Consider the following linear constraints:

$$-4x_1 + 4x_2 + 5x_3 + 6x_4 + 7x_5 + 10x_6 \leq 6 \tag{4.1}$$

$$x_2 + x_3 + x_4 \leq 1 \tag{4.2}$$

$$x_2 + x_5 \leq 1 \tag{4.3}$$

The first step is to rewrite constraint (4.1) as a knapsack constraint:

$$4\bar{x}_1 + 4x_2 + 5x_3 + 6x_4 + 7x_5 + 10x_6 \leq 10$$

Now, all the constraints are in the knapsack constraint format and we can run our algorithm for building the CG. Figure 4.1 shows the graph associated with constraints (4.1) to (4.3).

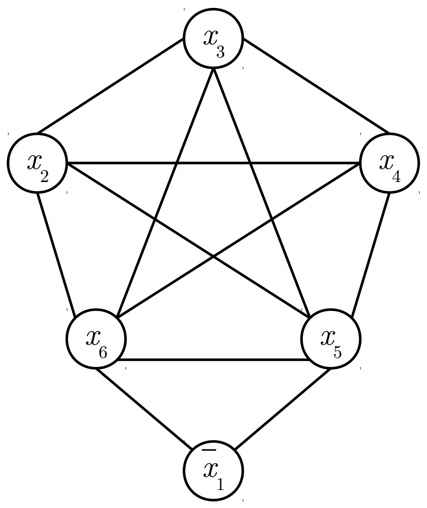


Figure 4.1: Conflict graph for constraints (4.1) to (4.3). For practical purposes, vertices that have only trivial conflicts were omitted.

Set $\mathcal{C}$ is then created, containing cliques of the constraints (4.2) and (4.3). The clique strengthening procedure is first applied to constraint (4.2), producing the extended constraint:

$$x_2 + x_3 + x_4 + x_5 + x_6 \leq 1 \tag{4.4}$$

Then, we remove constraints (4.2) and (4.3) since they are dominated by the extended constraint (4.4). There are no more constraints in $\mathcal{C}$ to be extended. Thus, the execution of clique strengthening in constraints (4.1) to (4.3) results in the following constraints:

$$\begin{aligned} -4x_1 + 4x_2 + 5x_3 + 6x_4 + 7x_5 + 10x_6 &\leq 6 \\ x_2 + x_3 + x_4 + x_5 + x_6 &\leq 1 \end{aligned}$$

4.1 Computational Results

An experiment was conducted for evaluating the ability of our preprocessing routine to produce strengthened formulations and to reduce the size of MILP models with respect to the number of constraints. This experiment was carried out on four computers with Intel Core i7-4790 3.60 GHz processors and 32 GB of RAM running Ubuntu Linux version 18.04 64-bit. The source code was developed in C++ programming language and compiled with g++ version 7.4.0.

We ran clique strengthening for all MILP models of the instance sets presented in Section 2.5, limiting the execution of the algorithm to constraints with at most 128 variables ($\alpha_{max} = 128$). This value was defined on a preliminary experiment that investigated the impact of setting different values to α_{max}, considering the execution times and the improvements in the LP relaxation of the MILP models.

After performing clique strengthening, we used the *COIN-OR Linear Program Solver* (CLP)[1] to solve the LP relaxation of the preprocessed MILP models, and then we calculated the gap closed. The percentage of the integrality gap closed is computed as follows:

$$gapClosed = 100 - 100 \times \frac{bestSol - currentLP}{bestSol - firstLP}$$

where *bestSol* is the best-known solution of the MILP model, $firstLP$ is the objective value of the root node LP relaxation and *currentLP* represents the objective value of the LP relaxation after applying clique strengthening. Therefore, as the percentage of gap closed increases, the difference between the objective value of the best-known solution and the objective value of the current LP relaxation decreases.

We considered four versions of clique strengthening, whose difference is in the clique extension procedure:

rnd: the clique extension procedure randomly selects the candidate variables.

deg: the clique extension procedure selects, at each iteration, the candidate variable with the highest degree.

[1]`https://github.com/coin-or/Clp`

mdg: the clique extension procedure selects, at each iteration, the candidate variable with the highest modified degree.

rc: the clique extension procedure selects, at each iteration, the candidate variable with the lowest reduced cost.

Figure 4.2 provides the results regarding the percentage of rows eliminated, execution times and the percentage of gap closed by executing clique strengthening in the considered instance sets. The execution times were measured considering the time spent in performing clique strengthening and solving the LP relaxation of the preprocessed MILP models. Complete results of this experiment are available for download at `http://professor.ufop.br/samuelbrito/thesis`.

The four versions presented a similar performance. A more significant difference is seen in some execution times of these approaches, where version *rc* presented the highest values. This version requires solving the LP relaxation of the MILP model to compute the reduced costs of the variables before performing the clique extension procedure, which increased its execution times.

Solving the LP relaxation of the preprocessed MILP models taken a considerable part of the execution times presented in Figure 4.2. Disregarding the time spent in this process, the maximum execution time of clique strengthening was 14.84 seconds. Furthermore, this routine ran in less than one second for 289 of 320 instances.

Despite the low impact on improving the values of the LP relaxation, there was a significant reduction in the number of constraints of the MILP models from instance sets *bmc* and *bpwc*. The percentage of constraints eliminated in instances from set *bmc* was more than 85% and above 71% in instances from *bpwc*.

Regardless of the version, the execution of the preprocessing routine did not improve the linear relaxation neither reduced the number of constraints for 205 of 253 instances from *miplib*. These instances were not affected by our preprocessing routine because they have no set packing constraints. For the other 48 instances of this group, our preprocessing routine was able to close the gap by up to 98.94% and reduce the number of constraints by up to 99.91%. Instance *sorrel3*, for example, had its number of constraints reduced by 95.39% and its integrality gap was closed by 98.94% after performing version *deg* of clique strengthening. This instance belongs to the Maximum Independent Set Problem and contains $169,162$ set packing constraints of size two that are used to model the edges of a graph.

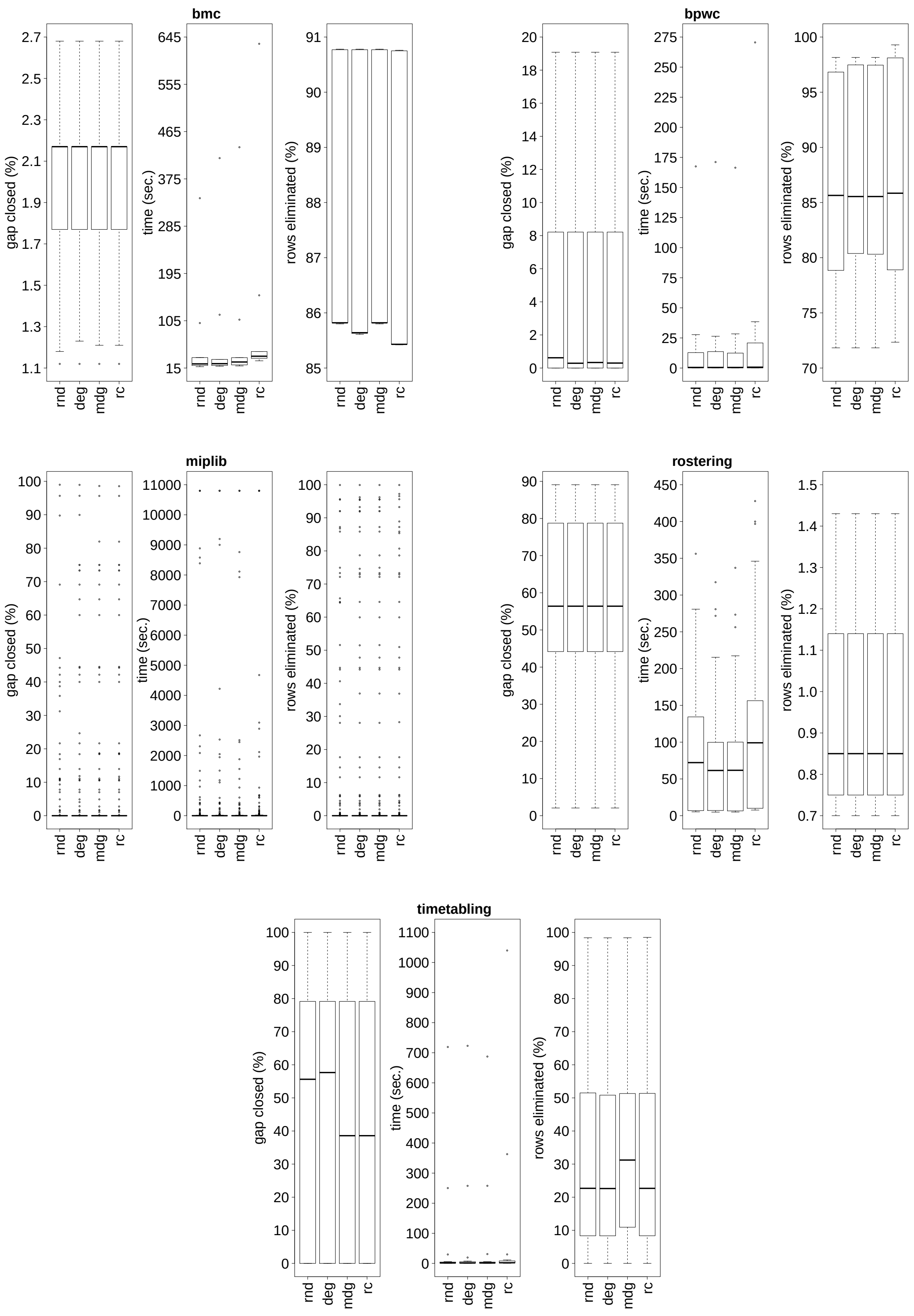


Figure 4.2: Results of the execution of clique strengthening.

Several constraints of the instances from set *rostering* were extended, but few became dominated. Thus, there was a small reduction in the number of constraints. However, the execution of clique strengthening in these instances allowed a significant improvement in the values of LP relaxations, closing the gap by up to 89.09%.

Significant results were also observed in instances of set *timetabling*. The four versions of our preprocessing routine were able to reduce the number of constraints of instances *trd445c* and *trdta0010* in 98.38% and 80.35%, respectively. Moreover, the integrality gaps in these instances were completely closed. These instances belong to a real Educational Timetabling problem of a Brazilian university (Gonçalves and Santos, 2008), containing a large number of set packing constraints that are used to model pairs of conflicting assignments based on student enrollments.

As the results show, clique strengthening can both reduce the number of constraints and also produce stronger formulations. It is more effective when applied to MILP models that have several constraints expressed by pairs of conflicting variables. Since we limited the size of constraints to apply this routine, its execution cost was relatively low, making possible its integration into MILP solvers. We chose *deg* as the default version of clique strengthening since it presented some execution times smaller than the other versions without decreasing the integrality gap closed.

4.2 Conclusion

This chapter presented a preprocessing routine that extends set packing constraints by the inclusion of additional conflicting variables. A greedy algorithm uses the information from a conflict graph to augment the cliques formed by the set packing constraints of a MILP model. After extending a constraint, a dominance checking procedure is performed to remove all constraints that become dominated. Computational results show that this preprocessing routine can both reduce the number of constraints and also produce stronger formulations.

Chapter 5

Cutting Planes

A primary application for conflict graphs is the generation of valid inequalities, also known as *cuts*. Cuts are linear constraints $a^T x \leq b$ that are violated by the current LP solution of a MILP model but do not remove any integer feasible solution. The addition of such inequalities enables tightening the LP relaxation of a MILP model, approximating it to the convex hull of integer feasible points. Cutting planes are often combined with a branch-and-bound scheme, resulting in the branch-and-cut or cut-and-branch algorithms that are present in modern MILP solvers.

Any feasible solution of a MILP model defines a vertex packing in its associated conflict graph. A vertex packing in a graph $G = (V, E)$ is a subset $P \subseteq V$ for which all $v_j, v_k \in P$ satisfy $(v_j, v_k) \notin E$. Based on this concept, one can conclude that any valid inequality for the vertex packing polytope is also valid for the convex hull of feasible solutions for this MILP model (Atamtürk et al., 2000). Thus, conflict graphs can be used to find inequalities that cut off the current LP solution.

Cliques and odd cycles are some of the most common classes of inequalities derived from the vertex packing polytope. Generally, the improvement in the value of the LP relaxation obtained by the inclusion of odd-cycle inequalities is small (Borndorfer, 1998; Méndez-Díaz and Zabala, 2008). However, the execution of a routine to separate these inequalities is computationally inexpensive in comparison with other cut separators, since they can be separated in polynomial time using shortest path algorithms (Grotschel et al., 1993; Rebennack, 2009).

The following sections present our routines for separating these conflict-based cuts. Furthermore, a cut pool structure used to filter and store cuts is presented. Preliminary versions of our cut separators were successfully applied to three classical combinatorial optimization problems: Capacitated Vehicle Routing (Pecin et al., 2017), Project Scheduling (Araujo et al., 2020) and Nurse Rostering (Santos et al., 2016). The use of these routines contributed to the solution of several hard instances for the first time in the literature.

5.1 Clique Inequalities

A clique inequality for a set C of conflicting variables is defined as:

$$\sum_{j \in C} x_j \leq 1$$

where C is a subset of the binary variables and their complements. As mentioned earlier, a clique represents a constraint in which at most one of the involved variables can be equal to one.

The main goal of the clique separation routine developed in this work is not to find *a most* violated inequality, but *a set of* violated inequalities. Previous work has proven this to be the best strategy. For example, Burke et al. (2012) used an algorithm to discover a most violated clique, but their computational results motivated the inclusion of additional cuts found during the separation process. This result is consistent with reports of the utilization of other cuts applied to different models, such as Chvátal-Gomory cuts (Fischetti and Lodi, 2007). The option of inserting a large number of valid inequalities at the same time is also responsible for increasing the importance of Gomory cuts (Cornuéjols, 2007).

Our clique separation routine is presented in Algorithm 5.1. It consists of using an LP solution $\check{x}$ and a conflict graph $G = (V, E)$ of a given MILP model to separate and return a set $\mathcal{S}$ of cliques violated by solution $\check{x}$. Parameter $minViol$ is used to control the minimum violation that a clique must have to enter in $\mathcal{S}$. Our clique separator executes two main steps: it separates violated cliques in the first step (lines 1 to 4) and extend these cliques in the second step (lines 5 to 12).

We begin generating a subgraph $G' = (V', E')$ induced by all variables (and their

Algorithm 5.1: Clique Cut Separator

```
   Input: LP solution $\breve{x}$, conflict graph $G = (V, E)$, $minViol$ and $maxCalls$.
   Output: Set $\mathcal{S}$ of violated cliques.
1  Let $G' = (V', E')$ be the subgraph of $G$ induced by all variables with fractional values
     in $\breve{x}$;
2  $w_j \leftarrow \breve{x}_j,\ \forall j \in V'$;
3  $minW \leftarrow 1 + minViol$;
4  $\mathcal{S} \leftarrow FindCliques(G', w, minW, maxCalls)$;
5  for $C \in \mathcal{S}$ do
6      Let $d$ be the vertex in $C$ with the smallest degree;
7      $L \leftarrow \{k \in N_G(d) \mid k \notin C\}$;
8      while $L \neq \emptyset$ do
9          Let $l$ be the vertex in $L$ with the smallest reduced cost in the current LP
             relaxation;
10         Remove $l$ from $L$;
11         if $\exists l \in N_G(k)\ \forall k \in C$ then
12             $C \leftarrow C \cup \{l\}$;
13 return $\mathcal{S}$;
```

respective complements) with fractional values at LP solution $\breve{x}$. Then, for each vertex j in subgraph G', we define the weight w_j as the value of its corresponding variable x_j in $\breve{x}$. The weight of a vertex $\bar{j}$ that represents the complement of a variable x_j is $w_{\bar{j}} = 1.0 - \breve{x}_j$. Now we have to search for cliques in G' whose sum of weights of its vertices is greater than or equal to $1 + minViol$ (line 4). These are the violated cliques.

The separation of violated cliques uses a modified version of the Bron-Kerbosch (BK) algorithm (Bron and Kerbosch, 1973). Although BK has exponential computational complexity in the worst case, the use of pivoting and pruning strategies enables efficient exploration of the search space. In practice, even for harder instances, maximal cliques with high weights are found during the first stages of the search. To avoid spending too much time in the clique separation step, we limit the number of recursive calls of BK by including a parameter called $maxCalls$. Details about this algorithm are discussed in Subsection 5.1.1.

After executing the BK algorithm, we have a set of violated cliques stored in $\mathcal{S}$. The clique extension module (lines 5 to 12) is then performed to extend each clique $C \in \mathcal{S}$ by inserting the variables (or their complements) with integer values at the current LP solution $\breve{x}$. For this, we use a greedy strategy and conflict graph G.

First, we create a set L of candidates to enter the clique C. It is built with the neighbors of the vertex in C with the smallest degree, excluding those that are already in C (lines 6 and 7). $N_G(d)$ indicates the vertices in G that are adjacent to vertex d. Then, we try to insert additional vertices in C by iteratively selecting the vertex $l \in L$ with the smallest reduced cost in the current LP relaxation. At each iteration, vertex l is inserted in C only if it is adjacent to all vertices in C. This process repeats until L is empty.

Figure 5.1 illustrates the importance of extending clique inequalities. Vertices within the gray area indicate variables with nonzero values in the solution of the current LP relaxation. Only vertices x_2, x_3 and x_4 could contribute toward defining a most violated clique inequality. Despite this, subsequent LP relaxations would include three different K_3 cliques, alternating the variable whose value is equal to zero. Reoptimizations of the LP could be avoided if the inequality of the K_4 clique was inserted immediately after the first LP relaxation of the problem. Moreover, a less dense constraint matrix may be obtained with the insertion of this dominant constraint.

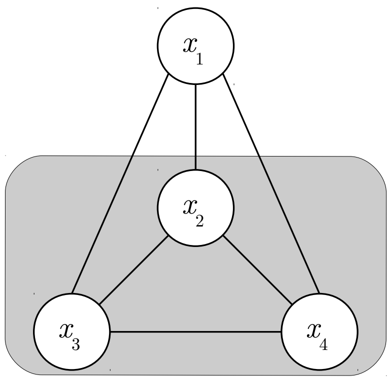


Figure 5.1: Example of a K_3 in which the extension module could be applied, transforming it into a K_4.

5.1.1 Bron-Kerbosch Algorithm

The main component of our clique separator is based on the *Bron-Kerbosch* algorithm, which is responsible for finding cliques with weights greater than a certain threshold. BK is a backtracking-based algorithm that enumerates all maximal cliques in undirected graphs (Bron and Kerbosch, 1973).

Some strategies to improve this algorithm are present in the literature. For example, in the same work as they presented the algorithm, Bron and Kerbosch (1973) introduced a variation that employs a pivoting strategy to decrease the number of recursive calls. Tomita et al. (2006) proposed a pivoted version of BK where all maximal cliques are enumerated in $O(3^{\frac{|V|}{3}})$ steps. This strategy makes the pivot a vertex with the highest number of neighbors in the candidate set.

Following the idea of Tomita et al. (2006), we implemented a pivoted version of BK. Additionally, a pruning strategy was added to accelerate the discovery of maximal cliques with weights greater than a threshold. Algorithm 5.2 details our implementation.

Algorithm 5.2: Bron-Kerbosch algorithm for detecting maximal cliques with weights above a threshold.

```
1  Function $FindCliques(G, w, minW, maxCalls)$:
2      $R \leftarrow \emptyset$; $P \leftarrow V$; $X \leftarrow \emptyset$; $\mathcal{S} \leftarrow \emptyset$;
3      $BronKerbosch(G, w, minW, maxCalls, \mathcal{S}, R, P, X, 0)$;
4      return $\mathcal{S}$;
5  Function $BronKerbosch(G, w, minW, maxCalls, \mathcal{S}, R, P, X, numCalls)$:
6      $numCalls \leftarrow numCalls + 1$;
7      if $numCalls > maxCalls$ then
8          return;
9      if $P \cup X = \emptyset$ then
10         if $\omega(R) \geq minW$ then
11             $\mathcal{S} \leftarrow \mathcal{S} \cup \{R\}$;
12         return;
13     if $\omega(R) + \omega(P) \geq minW$ then
14         choose a pivot vertex $u \in P \cup X$;
15         foreach $v \in P \setminus N_G(u)$ do
16             $BronKerbosch(G, w, minW, \mathcal{S}, R \cup \{v\}, P \cap N_G(v), X \cap N_G(v))$;
17             $P \leftarrow P \setminus \{v\}$;
18             $X \leftarrow X \cup \{v\}$;
```

Our algorithm works with three disjoint vertex sets: R, P and X. Set R is the set of vertices that are part of the current clique. Meanwhile, sets P and X are the candidate vertices to enter in R and all the vertices that have already been considered in earlier steps, respectively.

The algorithm begins with R and X empty, while P contains all the vertices of the graph. Within each recursive call, if the sets P and X are empty (line 8), then R is a maximal clique. This clique is stored in the clique set $\mathcal{S}$ if its weight $\omega(R) = \sum_{j \in R} w_j$ is greater than or equal to the minimum weight $minW$ (lines 10 to 11).

If R is not yet a maximal clique, the algorithm proceeds and calculates an upper bound to the weight that can be achieved by extending this set. This is done by adding the current weight of R to the weight of candidate vertices P. The upper bound to the weight of R is computed to avoid exploring sub-trees which would lead to cliques that do not satisfy the minimum weight $minW$ (line 13).

Then, a pivot vertex u is selected from $P \cup X$. It is well known that the selection of the pivot vertex is very influential on the overall performance of the method. Thus, we developed five different pivoting rules:

rnd: randomly selects a vertex.

deg: selects the vertex with the highest degree.

wgt: selects the vertex with the highest weight.

mdg: selects the vertex with the highest modified degree.

mwt: selects the vertex with the highest modified weight.

The modified weight of a vertex is computed as the sum of its weight and the weights of the vertices adjacent to it. To the best of our knowledge, this is the first time in the literature that different pivoting rules are evaluated for the BK algorithm in the context of clique cut separation.

Next, for each candidate vertex v which is not a neighbor of pivot u (line 15) a recursive call is made, adding v into clique R and updating sets P and X (line 16). At this point, sets P and X contain the neighbors of vertex v which are also neighbors of the other vertices contained in clique R. Using this configuration, the algorithm finds all extensions of R containing v. Once vertex v has been analyzed, it is removed from P and inserted into X (lines 17 and 18). The algorithm finishes after finding all maximal cliques with weights greater than $minW$ or when a maximum number of recursive calls $maxCalls$ is reached.

Since the most critical bottlenecks of Algorithm 5.2 are the set operations, we employ bit strings that exploit bit-level parallelism in hardware for optimizing the calculation of intersection, union and removal of sets. A bit string is an array that maps elements from some domain to values in the set $\{0, 1\}$. It is frequently used to represent a subset of a given population set. Each bit maps an element, where a 1-bit indicates the presence and a 0-bit the absence of an element in the subset. For example, in a population set of five elements $\{1, 2, 3, 4, 5\}$ bit string $B = 01101$ encodes the subset $\{1, 3, 4\}$, considering that the least significant bit is the right-most one.

We encode graph G as an array of bit strings, where each array entry corresponds to a row of the adjacency matrix of G. The complement graph $\bar{G}$ of G is also encoded as an array of bit strings to allow the implementation of efficient bitmasks operations concerning non-neighbor relations. Finally, sets P and X of the BK algorithm are also encoded as bit strings. With these representations, we can implement the critical set operations in Algorithm 5.2 as bitmask *AND* operations (Segundo et al., 2018):

- $P \cap N_G(v)$ in line 16: *AND* operation between bit string P and the v-th row of G.
- $X \cap N_G(v)$ in line 16: *AND* operation between bit string X and the v-th row of G.
- $P \setminus N_G(u)$ in line 15: *AND* operation between bit string P and the u-th row of $\bar{G}$.

5.2 Odd-Cycle Inequalities

Odd-cycle inequalities are also derived from the set packing polytope. Given a graph $G = (V, E)$, a subset $O \subseteq V$ is an odd cycle if the subgraph induced by O is a simple cycle with an odd number of vertices. In this case, the subgraph must have $|O|$ adjacent edges such that each vertex is incident to exactly two vertices. Thus, an odd cycle O formed by a set of binary variables (or their complements) defines the odd-cycle inequality:

$$\sum_{j \in O} x_j \leq \frac{|O| - 1}{2}$$

This inequality ensures that at most half of the variables can be activated.

Our odd-cycle separation routine is described in Algorithm 5.3. It is based on the concepts presented by Rebennack (2009) and returns a set $\mathcal{W}$ of tuples containing the violated odd cycles and their respective wheel centers. First, an auxiliary bipartite graph $G' = (V', E')$ is created from the original conflict graph $G = (V, E)$. Lines 2 to 4 present the creation of G'. The vertex set V' is formed by two subsets V_1 and V_2. In this case, for each vertex $j \in V$, two vertices j_1 and j_2 are created in V', where $j_1 \in V_1$

and $j_2 \in V_2$. Additionally, for each edge $(j,k) \in E$, two edges (j_1,k_2) and (j_2,k_1) are inserted into E', where $j_1, k_1 \in V_1$ and $j_2, k_2 \in V_2$. The auxiliary graph G' is bipartite since there is no edges connecting two vertices of V_1 or two vertices of V_2.

Algorithm 5.3: Odd-Cycle Cut Separator

```
Input: LP solution $\check{x}$ and conflict graph $G = (V, E)$.
Output: Set $\mathcal{W}$ of tuples containing violated odd cycles and their respective wheel
        centers.
1  $\mathcal{W} \leftarrow \emptyset$;
2  $V' \leftarrow \{j_1, j_2 \mid j \in V\}$;
3  $E' \leftarrow \{(j_1,k_2),(j_2,k_1) \mid (j,k) \in E\}$;
4  $G' \leftarrow (V', E')$;
5  for $(j,k) \in E$ do
6      $w(j_1,k_2) = (1 - \check{x}_j - \check{x}_k)/2$;
7      $w(j_2,k_1) = (1 - \check{x}_j - \check{x}_k)/2$ ;
8  for $j \in V$ do
9      $P \leftarrow ShortestPath(j_1, j_2, G', w)$;
10     Convert path $P$ to an odd cycle $O$ in the original graph $G$;
11     Let $\tilde{E}$ be the edge set of the subgraph of $G$ induced by $O$;
12     $cost \leftarrow 0$;
13     for $(j,k) \in \tilde{E}$ do
14         $cost \leftarrow cost + w(j,k)$;
15     if $cost < 0.5$ then
16         $C \leftarrow \emptyset$;
17         Let $d$ be the vertex in $O$ with the smallest degree;
18         $L \leftarrow \{k \in N_G(d) \mid k \notin O,\ \exists k \in N_G(j)\ \forall j \in O\}$;
19         while $L \neq \emptyset$ do
20             Let $l$ be the vertex that corresponds to the variable with the smallest
                 reduced cost in $L$;
21             Remove $l$ from $L$;
22             if $\exists l \in N_G(k)\ \forall k \in C$ then
23                 $C \leftarrow C \cup \{l\}$;
24         $\mathcal{W} \leftarrow \mathcal{W} \cup \{(O,C)\}$;
25 return $\mathcal{W}$;
```

The next step is to compute the weight of each edge of G' (lines 5 to 7), since our cut separator works with an edge-weighted graph. The weight of each edge in the auxiliary graph G' is computed according to the corresponding edge (j,k) of the original graph G, which is defined as:

$$w(j,k) = \frac{1 - \check{x}_j - \check{x}_k}{2}$$

where $\check{x}_j$ and $\check{x}_k$ are the values of variables x_j and x_k at LP solution $\check{x}$. Here, variables x_j and x_k are conflicting, which implies that $\check{x}_j + \check{x}_k \leq 1$ and, consequently, $w(j,k) \geq 0$.

After creating the auxiliary bipartite graph and computing its edge weights, the search for violated odd cycles begins. For each vertex $j \in V$ we run Dijkstra's algorithm in G' to find the shortest path P from j_1 to j_2. The shortest path has an odd number of edges since vertices j_1 and j_2 are in two different sets of the bipartition. Then, the corresponding odd-cycle O is constructed from the shortest path P (line 10). The resulting odd-cycle inequality is violated by the current LP solution $\check{x}$ if and only if:

$$\sum_{(j,k) \in \tilde{E}} w(j,k) < 0.5$$

where $\tilde{E}$ is the set of edges of the subgraph of G induced by the variables in odd cycle O. Thus, Algorithm 5.3 tries to find one odd-cycle inequality (namely, a most violated one) for each variable. Odd-cycles of size three are discarded since they correspond to cliques and could be found by the clique cut separation procedure.

When a violated odd cycle is found, a lifting step is performed (lines 16 to 23). This step tries to transform the violated odd cycle into an odd wheel. In graph theory, an odd wheel is an odd cycle that contains an additional vertex that is adjacent to all other vertices. Thus, an odd wheel can be obtained by inserting a variable into the center of the odd cycle. An odd-wheel inequality has the following format:

$$\sum_{j \in O} x_j + \frac{|O| - 1}{2} x_c \leq \frac{|O| - 1}{2}$$

where O is an odd cycle and x_c is a variable that has conflict with all of those in O.

Our lifting step consists of a new approach: we use a greedy strategy that finds and inserts a clique C in the center of the odd cycle. In the literature, it is common to consider the insertion of only one variable into the center of the odd cycle, such as the lifting strategy presented by Rebennack (2009). Initially, we select a vertex d with the smallest degree between the vertices of O. Then, we create a set L of candidates to compose clique C, containing the neighbors of d that are conflicting with all vertices in O but are not included in O (lines 17 and 18). Following, we construct clique C by iteratively selecting the vertex $l \in L$ whose corresponding variable has the smallest reduced cost. A vertex l is inserted in C only if it is adjacent to all vertices in C. This process repeats until L is empty. Finally, a tuple formed by violated odd-cycle O and its corresponding wheel center C is inserted in $\mathcal{W}$ (line 24).

Figure 5.2 illustrates an odd wheel formed by the inclusion of the clique involving variables $\{x_6, x_7, x_8\}$ into the center of the odd cycle formed by variables $\{x_1, x_2, x_3, x_4, x_5\}$. In this example, each variable in the clique is conflicting with all variables that compose the odd cycle. The odd-wheel inequality associated with the odd cycle of this figure is:

$$x_1 + x_2 + x_3 + x_4 + x_5 + 2x_6 + 2x_7 + 2x_8 \leq 2$$

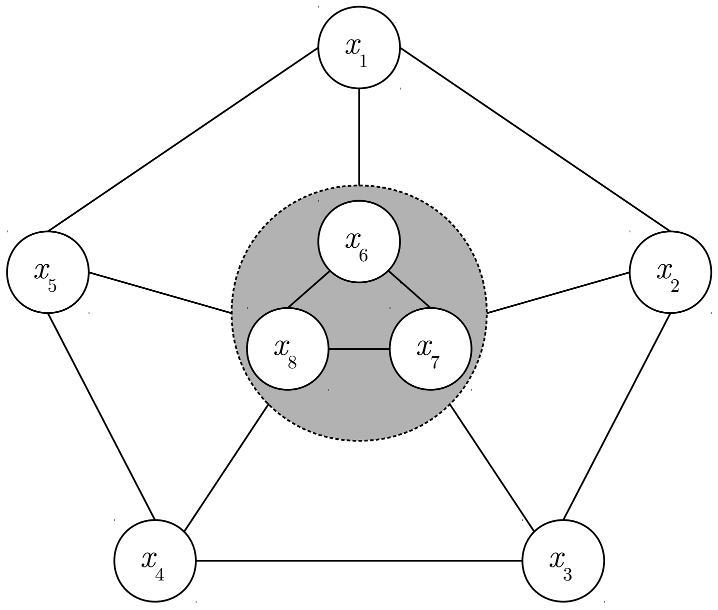


Figure 5.2: Example of an odd cycle with the inclusion of a wheel center. Vertices x_6, x_7 and x_8 are connected to all the vertices of the odd cycle formed by $\{x_1, x_2, x_3, x_4, x_5\}$.

5.3 Cut Pool

The execution of our cut generation routines can produce a large number of valid inequalities, and the insertion of several valid inequalities can deteriorate the solution process of the LP relaxation and generate numerical issues. On the other hand, the insertion of few cuts can deteriorate the dual bound that is used to prove optimality. We try to solve this tradeoff by using a data structure called *cut pool*. This structure is responsible for storing the cuts, maintaining only those that are most promising.

Our cut pool implements methods for removing repeated and dominated cuts and filtering the cuts that are stored by this structure. Before inserting a cut into the cut pool, we use a hash table to verify if it has already been inserted. Thus, repeated cuts are quickly discarded. The dominance checking method first normalizes the right-hand side of the cuts. Then, a cut with coefficients a and right-hand side b dominates another constraint with coefficients a' and right-hand side b' if and only if $b \leq b'$ and $a_j \geq a'_j$ for each $j \in \{1, ..., n\}$. Since the dominance checking is time-consuming, we only perform this step after generating all cuts of the current cut loop iteration.

A major challenge in developing a cut pool structure is to decide how to filter cuts, maintaining a set of the most promising ones. Our filtering strategy consists of computing a score $S(C)$ for each cut C that we try to insert into the cut pool and using this score to decide whether the cut will be stored or discarded. Given a cut C and an LP solution $\breve{x}$, the score of this cut is computed as:

$$S(C) = \frac{viol(C)}{actv(C)}$$

where $viol(C)$ is the violation of the cut with respect to $\breve{x}$ and $actv(C)$ is the number of variables in C whose values in $\breve{x}$ are greater than zero.

We use an auxiliary array to identify, for each variable, the cut in the pool with the best score that contains this variable. The auxiliary array is updated at each successful insertion of a cut in the cut pool. A cut is only inserted into the cut pool if it has the best score for at least one variable. Therefore, we limit the maximum number of cuts that can be stored by our cut pool to the number of the variables of the MILP model under consideration.

5.4 Computational Results

We conducted three computational experiments to evaluate the conflict-based cut separators proposed in this thesis. The first experiment evaluated the pivoting rules implemented in the BK algorithm. The second experiment analyzed the improvements in the dual bounds obtained by the execution of our clique cut separator and compared this routine against the clique cut separators available in some MILP solvers. The last computational experiment evaluated the contribution of our odd-cycle cut separator in improving the dual bounds and compared the performance of two lifting strategies. All of these experiments were carried out on four computers with Intel Core i7-4790 3.60 GHz processors and 32 GB of RAM running Ubuntu Linux version 18.04 64-bit. The source code was developed in C++ programming language and compiled with g++ version 7.4.0.

The metrics used for comparison purposes were the execution times and the gap closed by the cut separators. The percentage of gap closed is computed as:

$$gapClosed = 100 - 100 \times \frac{bestSol - currentLP}{bestSol - firstLP}$$

where *bestSol* is the best-known solution of the MILP model, $firstLP$ is the objective value of the root node LP relaxation and *currentLP* represents the objective value of the LP relaxation after including the separated cuts into the MILP model. The metric *average gap closed* is also employed and is computed as the arithmetic mean of the gap closed over the problem instances. Complete results of the experiments presented in this chapter are available for download at `http://professor.ufop.br/samuelbrito/thesis`.

5.4.1 Pivoting Rules of Bron-Kerbosch Algorithm

An essential part of our clique cut separator is the BK algorithm. It is responsible for finding cliques with weights greater than a certain threshold in a vertex-weighted graph. The pivoting rule plays an important role in this algorithm, allowing the reduction of the number of recursive calls made by it.

In this sense, we conducted a computational experiment to evaluate the performance of the pivoting rules that we implemented in the BK algorithm. First, we generated an instance set containing several vertex-weighted graphs. We ran our clique cut separator in the MILP models presented in Section 2.5, stopping this routine when no cut was separated or after performing three iterations of the cut generation loop. The pivoting rule of BK was randomly selected at each execution of the algorithm. In each iteration of the cut generation loop, we used CLP to solve the LP relaxation of the MILP models and saved the subgraphs induced by the variables with fractional values. After performing these steps, 438 conflict graphs were generated.

Following, we ran the BK algorithm to detect the violated cliques in these graphs, limiting the maximum number of recursive calls to $100,000$ ($maxCalls = 100,000$). We investigated the performance of five versions of the BK algorithm that differ only from the pivoting rule. These versions are referred to here according to their pivoting rules, which were presented in Section 5.1.1.

Table 5.1 presents the summarized results of the execution of each version of the BK algorithm. In this table, column "exact" indicates the number of graphs for which the algorithm ran completely, without stopping for the maximum number of recursive calls. Column "avg calls" presents the average number of recursive calls made by the algorithm. The average and the maximum number of violated cliques found are presented in columns "avg clqs" and "max clqs". Finally, columns "avg time" and "max time" present the average and the maximum time spent, in seconds, by each version of the BK algorithm.

Table 5.1: Summarized results of the execution of BK algorithm with different pivoting rules.

version	exact	avg calls	avg clqs	max clqs	avg time	max time
rnd	416	12,515.72	370.90	8,940	0.57	35.24
deg	414	13,506.78	356.14	11,559	0.56	26.95
wgt	424	9,759.37	381.66	11,115	0.28	14.80
mdg	413	13,393.91	355.09	11,660	0.49	26.64
mwt	410	13,790.83	357.18	9,076	0.53	34.09

The execution of the BK algorithm was very fast. Regardless of the pivoting rule, the time spent by this algorithm was less than one second for 94% of the instances. The instances in which the algorithm spent more than one second have dense conflict graphs with many cliques explicitly stored. Consequently, the process of iterating over the conflicts to encode the graphs as arrays of bit strings took the largest portion of the execution times in these instances.

According to the results, the pivoting rule that defines the vertex with the highest weight as the pivot obtained the best results. The number of recursive calls made by this version is up to 29% less than those made by other versions. The reduction in the number of recursive calls implied a decrease in the execution time, making *wgt* the fastest version among those tested. In addition, *wgt* ran completely for a greater number of instances and found more cliques than the other versions. Based on these results, we defined *wgt* as the default pivoting rule of our implementation of the BK algorithm.

5.4.2 Clique Cut Separator Experiments

After choosing the pivoting rule to be used in the BK algorithm, we evaluated the ability of our clique cut separator in tightening the LP relaxations. In this experiment, we considered two versions of our clique cut separator: one version with the lifting module disabled and other with this module activated. These versions are referred to here as *bkclq* and *bkclqext*, respectively.

We compared the performance of our clique cut separator against the clique cut separators implemented in three MILP solvers. The first clique cut separator that we compared, referred to here as *cglclq*, is used by the COIN-OR CBC solver and provided by the COIN-OR Cut Generation Library (CGL)[1]. We used the C++ API of CGL to develop a routine that calls the clique cut separator at each iteration of the cut generation loop.

The second clique cut separator compared in this experiment, named here as *glpclq*, is provided by the open-source solver of the GNU Linear Programming Kit (GLPK)[2] version 4.65. We ran GLPK with all presolving, preprocessing, heuristics and other cut separators turned off. Thus, we capture only the effect of the inclusion of the clique inequalities. Furthermore, we implemented a callback procedure that computes and stores the number of cuts separated, the current objective value of the LP relaxation, the gap closed and the time elapsed at each iteration of the cut generation loop.

The last clique cut separator that we compared was the one included in the commer-

[1]`https://github.com/coin-or/Cgl`
[2]`https://www.gnu.org/software/glpk/`

cial solver IBM ILOG CPLEX[3] version 12.8, referred to here as *cpxclq*. We considered only the "very aggressively" strategy of this cut separator since in preliminary experiments it performed slightly better than the other strategies. We ran CPLEX with all presolving, preprocessing, heuristics and other cut separators turned off, according to the CPLEX User's Manual[4]. We also implemented a callback procedure that computes and stores the current objective value of the LP relaxation, the gap closed and the time elapsed at each iteration of the cut generation loop.

All cut separators were executed at the root node LP relaxation of the instance problems presented in Section 2.5, considering at most 50 iterations of the cut generation loop and a time limit of $10,800$ seconds. CLP was employed to solve the LP relaxation at each iteration of *bkclq*, *bkclqext* and *cglclq*, while *glpclq* and *cpxclq* used their own linear program solvers.

Figure 5.3 presents the execution times and the gap closed by each clique cut separator. Comparing the two versions of our cut separator, one can observe that the inclusion of the lifting module contributed to improve the execution times. The insertion of lifted cliques avoided some reoptimizations of the LP relaxations, which saved some iterations of the cut generation loop and, consequently, the execution time of the cut separator. Moreover, the version of our clique cut separator that includes the lifting module produced better dual bounds.

Our clique cut separator improved the LP relaxation for 136 instances. Most instances in which the objective value of the LP relaxation was not changed belong to instance set *miplib*. In fact, we analyzed these instances and noted that their associated CGs have only trivial conflicts or a small set of non-trivial conflicts.

The dual bounds obtained by *bkclqext* are significantly better than those attained by *cglclq* and *glpclq* in all instance sets. Even in instances with denser conflict graphs, such as those in sets *bmc* and *bpwc*, *cglclq* and *glpclq* had difficulty finding violated cliques.

Similar results in terms of the percentage of gap closed were obtained by *bkclqext* and *cpxclq*, except in set *bpwc*. On the instances of this set, *cpxclq* did not find violated any violated clique, while the cliques separated by *bkclqext* contributed to close the gap by up 97.71%. It is worth mentioning that, for some instances, the initial bounds of the root node LP relaxations computed by CPLEX were already better than the values calculated by CLP and the linear program solver of GLPK, even with the preprocessing

[3]`https://www.ibm.com/analytics/cplex-optimizer`
[4]`https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/`

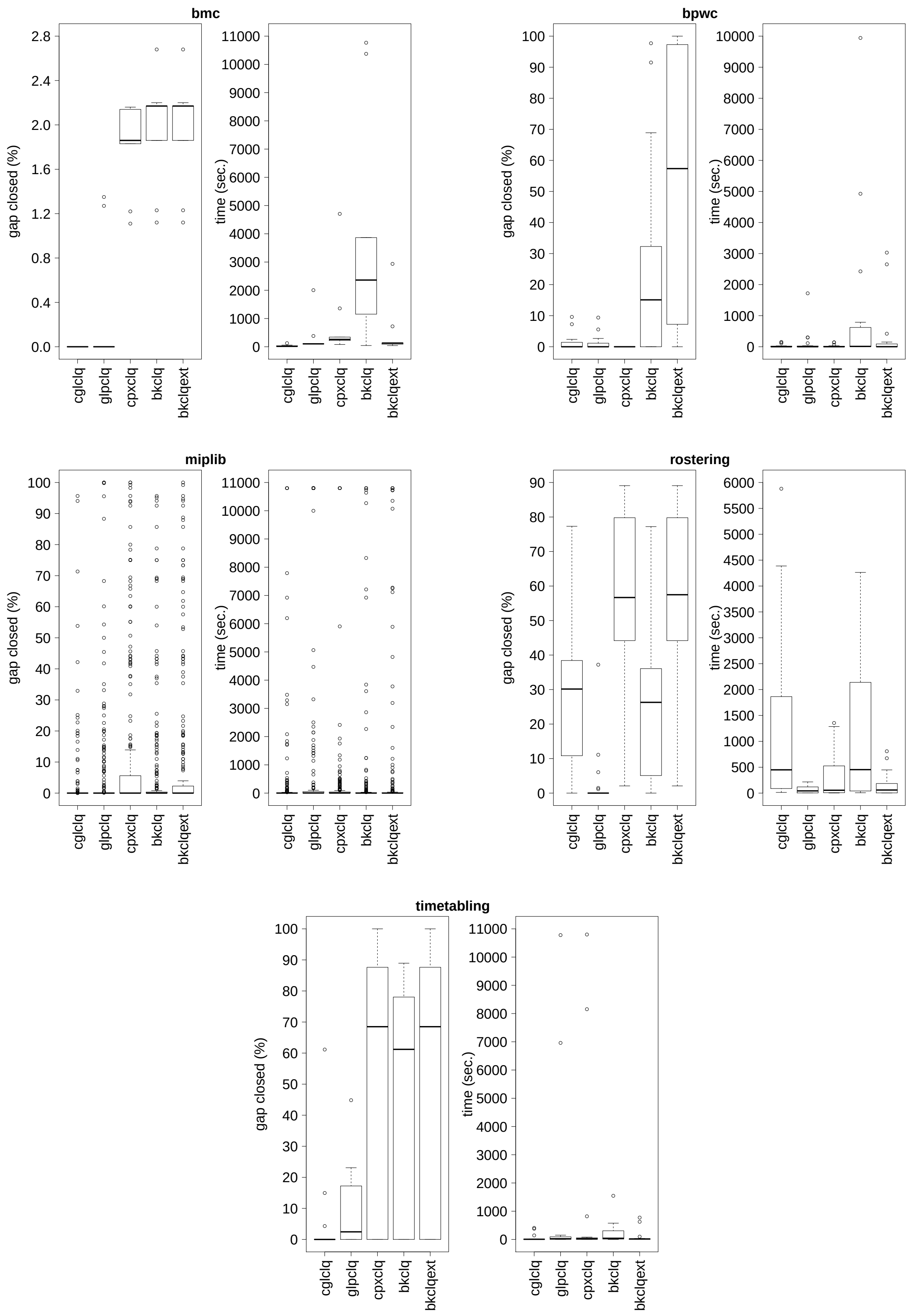


Figure 5.3: Execution times and gap closed by the clique cut separators.

routines disabled.

In addition to obtain better dual bounds, the execution of *bkclqext* was responsible for completely closing the gap for a greater number of instances. Cut separators *bkclqext*, *cpxclq* and *glpclq* completely closed the gap for 8, 5 and 2 instances, respectively. Cut separators *bkclq* and *cglclq* were not able to completely close the gap for any instance.

The average gap closed by the clique cut separators at each iteration of the cut generation loop is presented in Figure 5.4. Differences in the performances of the cut separators can be seen in the earlier steps of the separation process.

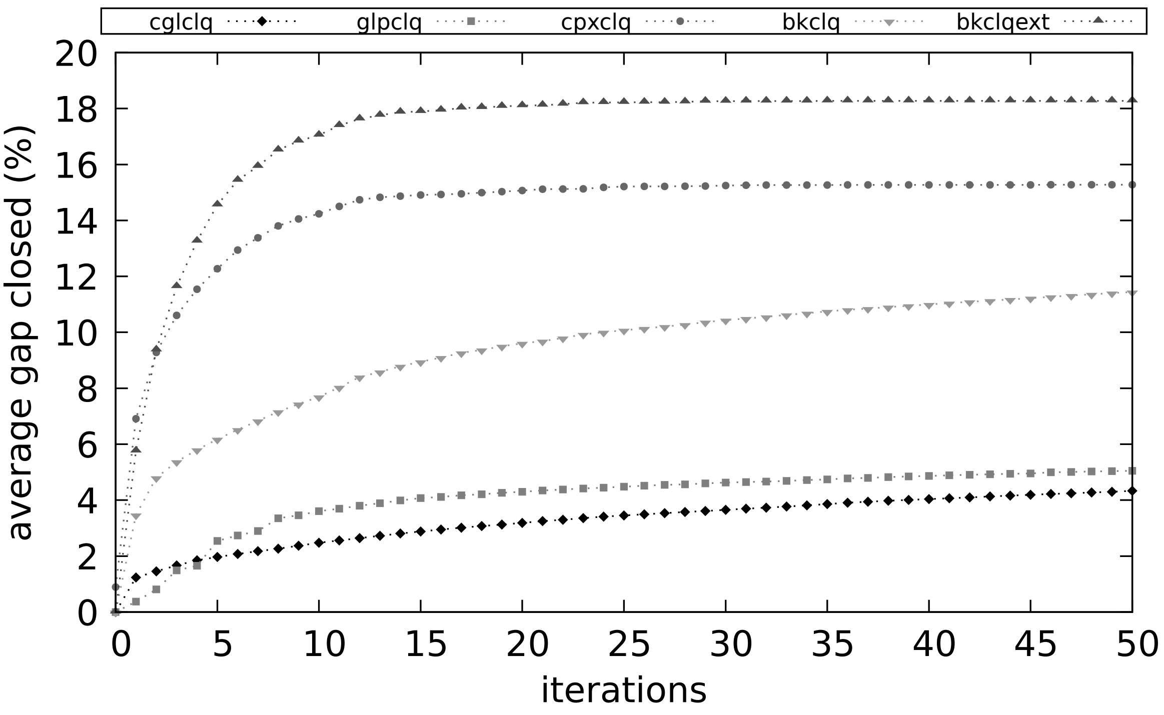


Figure 5.4: Average gap closed by each clique cut separator.

After 50 iterations, the average gap closed by the cut separators was 4.33% for *cglclq*, 5.05% for *glpclq*, 15.27% for *cpxclq* and 18.27% for *bkclqext*. Thus, the average gap closed by our clique cut separator was 4.22 times better than the clique cut separator of CGL, 3.62 times better than the clique cut separator provided by GLPK and 19.65% better than the clique cut separator of CPLEX.

5.4.3 Odd-Cycle Cut Separator Experiments

Following the evaluation of our conflict-based cut separation routines, we analyzed the ability of our odd-cycle cut separator in improving the bounds given by the LP relaxation of a MILP. We also investigated the impact of performing our proposed lifting module, which consists of inserting a clique into the center of an odd cycle. For this, we ran our odd-cycle cut separation at the root node LP relaxation of the instances, considering at most 50 iterations of the cut generation loop and a time limit of $10,800$ seconds. CLP was employed to solve the LP relaxation at each iteration.

We ran three versions of our cut separator. In the first version, named here as *off*, we did not execute the lifting module. In the second version, referred to here as *var*, we executed a lifting module that tries to insert one variable into the center of an odd cycle. Our proposed lifting module was performed in the third version of the cut separator and is referred to here as *clq*. The results of this experiment are presented in Figure 5.5.

Regardless of the version of the cut separator, the inclusion of odd-cycle inequalities had no significant impact on the dual bound improvement. For most instances, no odd cycles of size greater than three were found. As explained before, odd cycles of size three are not separated by the odd-cycle cut separator, since they correspond to cliques and can be separated by the clique cut separator. Even in instances where a considerable set of odd-cycle cuts were separated, the improvement in the LP relaxation was small. For example, *clq* separated $1,764$ odd-cycle cuts in instance *br2* of set *timetabling*, but the gap closed was 5.03%.

Odd-cycle cuts were separated in 82 of 320 instances, but the improvement in the objective value of the LP relaxation occurred only in 31 of them. The maximum percentage of gap closed by our cut separator was 88.00%, obtained by all separators in instance *neos8*. Several odd cycles of size 5, 7 and 9 were separated in this instance.

In general, the time spent in separating odd-cycle cuts was small. Considering all problem instances, the maximum time spent in this step was 7.42 seconds. The execution of CLP to solve the LP relaxation of the problems in each iteration of the cut generation loop was responsible for increasing the execution times.

It was not possible to detect differences between the performances of the three versions of our odd-cycle separator. In fact, there are few cases where the lifting module was able to insert wheels into the centers of the odd cycles. Regardless of the version of the lifting module, some odd cycles were transformed into odd wheels only in 11 instances.

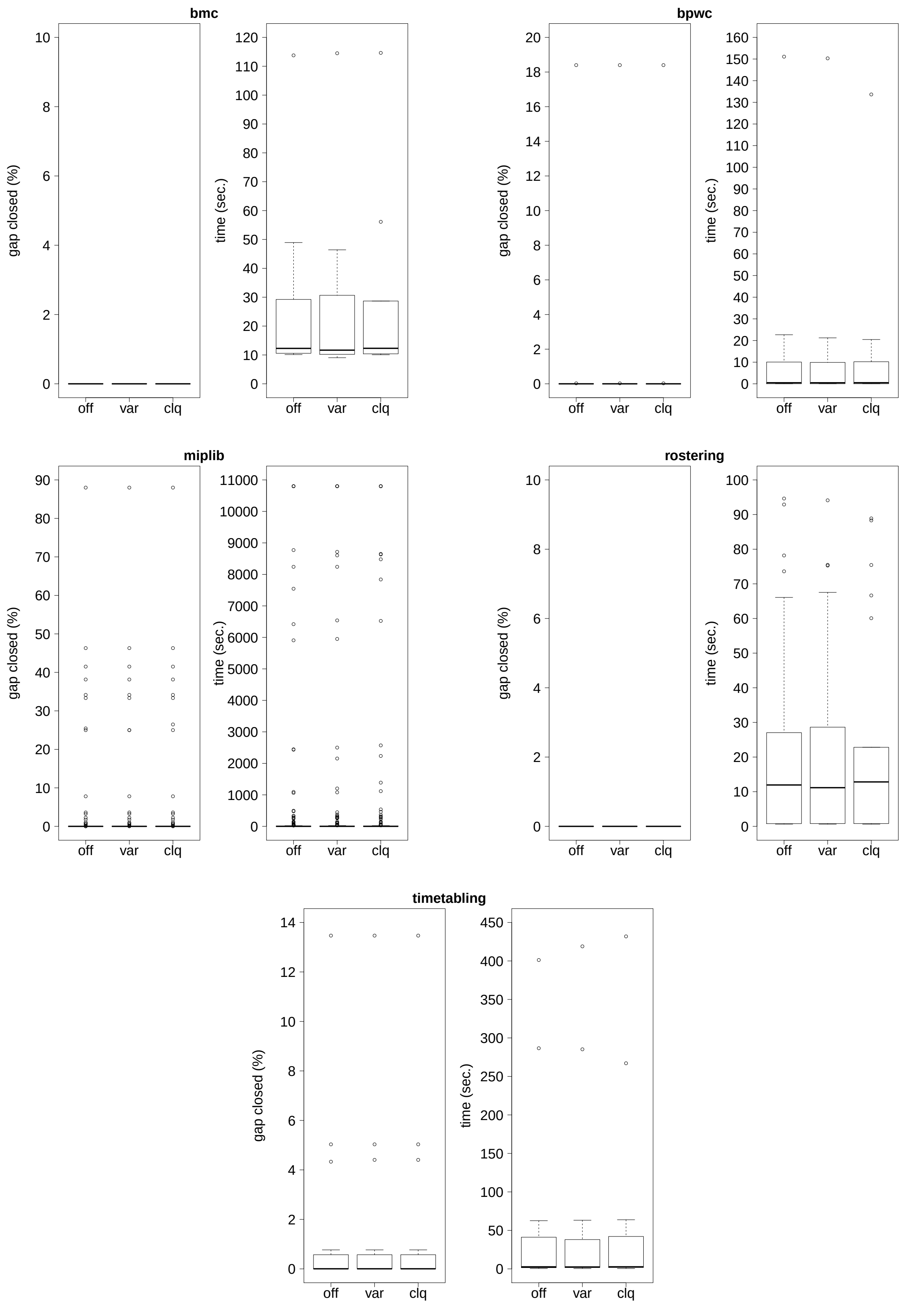


Figure 5.5: Execution times and gap closed by the odd-cycle cut separator.

For these instances, the average gap closed by *clq* was 3.84% better than *var*.

5.5 Conclusion

This chapter presented two cut separation routines and a data structure for storing the cuts. The proposed cut pool is responsible for removing repeated and dominated cuts as well as filtering the cuts in order to maintain only those that are most promising. Our clique cut separator uses the BK algorithm for separating a set of violated clique inequalities. It was capable of obtaining dual bounds at the root node LP relaxation which are even stronger than the ones provided by the clique cut separator of CPLEX solver. Our odd-cycle cut separator has a new lifting module that tries to insert cliques in the centers of the odd cycles. The improvements in the dual bounds obtained by including only odd-cycle cuts were relatively small.

Chapter 6

Improving the COIN-OR Branch-and-Cut Solver

The conflict graph-based algorithms and data structures proposed in this thesis were included in the source code of the COIN-OR Branch-and-Cut (CBC) solver. CBC is a MILP solver written in C++, and it is one of the fastest open-source alternatives nowadays. It is also a fundamental component used by Mixed-Integer Nonlinear solvers, such as Bonmin (Belotti et al., 2009) and Couenne (Bonami et al., 2008). The new version of CBC containing our contributions can be downloaded from the GitHub repository[1]. This version will be released as CBC 3.0.

In this new version, a conflict graph is constructed after the execution of the preprocessing routines of CBC, followed by the execution of the clique strengthening routine. Our conflict-based cut separators are performed during the execution of the branch-and-cut algorithm. The clique and odd-cycle cut separators of CGL, included in the previous version of CBC, were replaced by our cut separators. The following section investigates the performance of the new version of CBC.

6.1 Computational Results

Two experiments were performed to evaluate the new version of CBC that includes our conflict graph-based algorithms and data structures. The first experiment analyzed the individual contribution of our preprocessing and cut separation routines in the solving

[1]`https://github.com/coin-or/Cbc`

process of CBC. The second computational experiment compared the new version of CBC against its previous version. Both experiments were carried out on four computers with Intel Core i7-4790 3.60 GHz processors and 32 GB of RAM running Ubuntu Linux version 18.04 64-bit, considering the instances described in Section 2.5. The source code was developed in C++ programming language and compiled with g++ version 7.4.0.

For comparison purposes, we used the execution times and the gap closed by CBC in solving the MILP models. The percentage of gap closed is computed as:

$$gapClosed = 100 - 100 \times \frac{bestSol - lastLP}{bestSol - firstLP}$$

where *bestSol* is the best-known solution of the MILP model, $firstLP$ is the objective value of the root node LP relaxation and *lastLP* represents the lower bound obtained by CBC at the end of its execution. The metric *average gap closed* is also employed and is computed as the arithmetic mean of the percentage of gap closed over the problem instances.

6.1.1 Individual Impact of Each Routine

The first experiment investigated the individual performance impact of our preprocessing and cut separation routines in CBC solver. To this end, we first executed the new version of CBC and then individually removed each routine, generating additional configurations. The default parameters of CBC for heuristics, preprocessing, branching rules and cuts separators were used in all executions. We only turned off the clique and odd-cycle cut separators of CBC, since we are using our cut separators. Each configuration was executed for all instances of the sets described in Section 2.5 with a time limit of $10,800$ seconds.

The box plots of Figure 6.1 show the results of this experiment. In this figure, configuration "cbc+cg" refers to the new version of CBC including all of our routines. Configuration "-{clqstr}" represents the new version of CBC without performing our clique strengthening routine. Finally, configurations "-{bkclqext}" and "-{oddw}" contain the results of the execution of the new version of CBC without including our clique and odd-cycle cut separators, respectively. Detailed results of this experiment is presented in Table A.2 in the appendix.

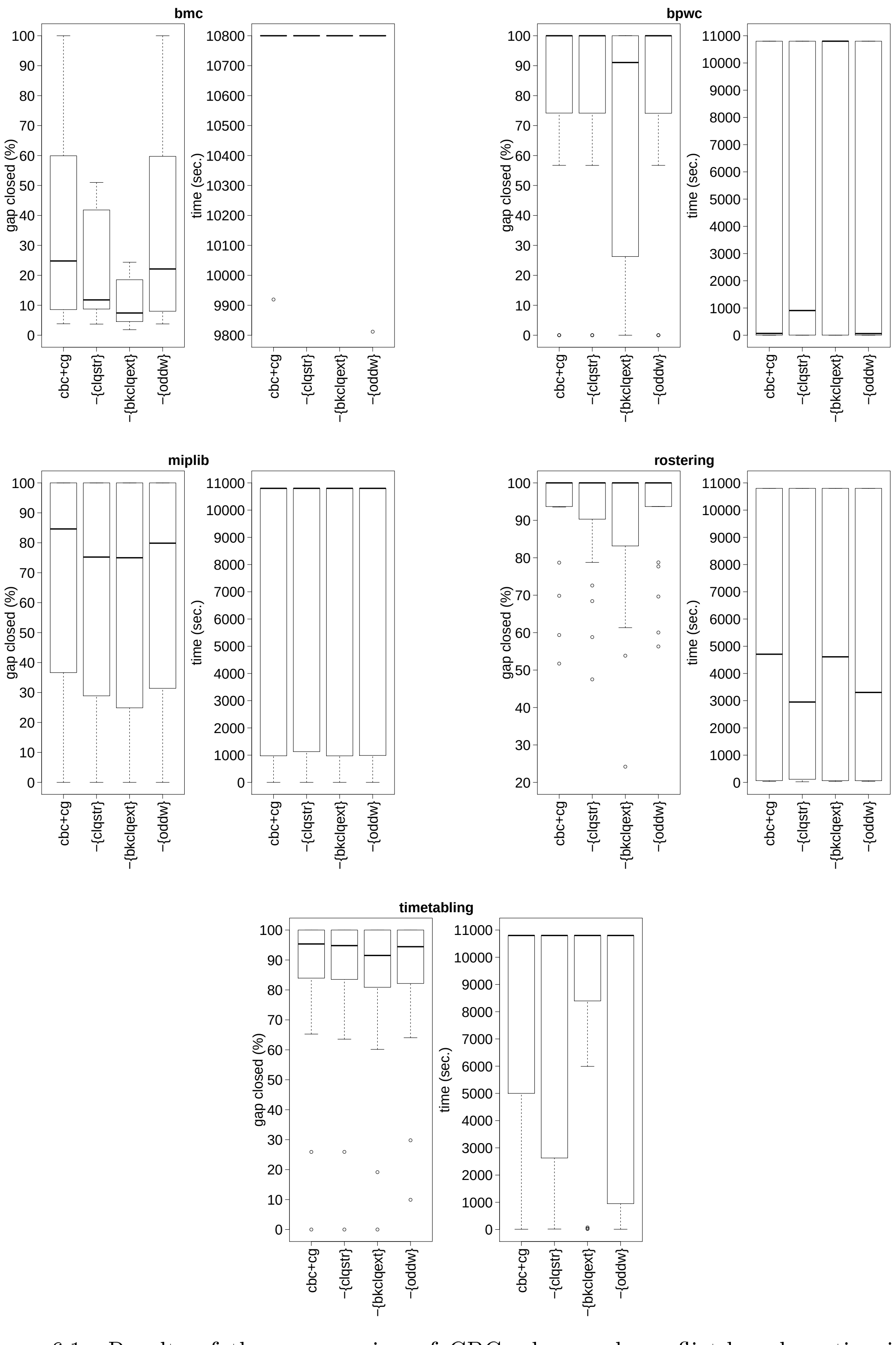


Figure 6.1: Results of the new version of CBC when each conflict-based routine is individually removed.

The most significant impact occurred when our clique cut separator was removed. In this case, the average gap closed by CBC decreased by 69.54% in instance set *bmc*, 14.51% in *bpwc*, 5.59% in *miplib*, 4.79% in *rostering* and 3.38% in *timetabling*. The removal of this routine also resulted in an increase in the average execution times. For example, the average execution time for instance set *bpwc* increased in 53.78%. Moreover, the total number of instances whose optimality was proven decreased in 12.40%.

6.1.2 Results of the New Version of CBC Solver

As the second experiment, we investigated the performance improvement of the new version of CBC against its previous version. The default parameters of CBC for heuristics, preprocessing, branching rules and cuts separators were used in both versions. We ran each version of CBC on all MILP models of the sets described in Section 2.5, considering a time limit of $10,800$ seconds. The new version of CBC is denoted here as *cbc+cg*, while the previous version of this solver is referred to as *cbc*. Figure 6.2 shows the results of this experiment. Detailed results are presented in Table A.2 in the appendix.

The inclusion of our conflict graph-based algorithms and data structures in CBC contributed significantly to improve the dual bounds obtained by this solver. As observed in Figure 6.2, the median gap closed by the new version of CBC is greater than the values obtained by the previous version of this solver in all instance sets. The percentage of gap closed by *cbc+cg* was greater than or equal to that obtained by *cbc* in all 320 instances. Consequently, the average gap closed by CBC increased from 58.86% to 68.76%, representing an improvement of 16.82%. Reductions in the execution times were observed in several instances.

In order to better visualize the results, we computed the evolution of the average gap closed by each version of CBC over the execution time for each instance set. The results are presented in Figure 6.3. The most significative improvements were obtained in instance sets *bmc*, *bpwc* and *rostering*. In instances of these three sets, the clique strengthening routine considerably reduced the number of rows of these instances, while the clique cut separator inserted strong valid inequalities.

At the end of the executions, the average gap closed by *cbc+cg* was four times better than the one obtained by *cbc* in instance set *bmc*. Furthermore, the average gap closed by the new version of CBC was 54.01% better in instance set *bpwc*, and 66.45% better in *rostering*.

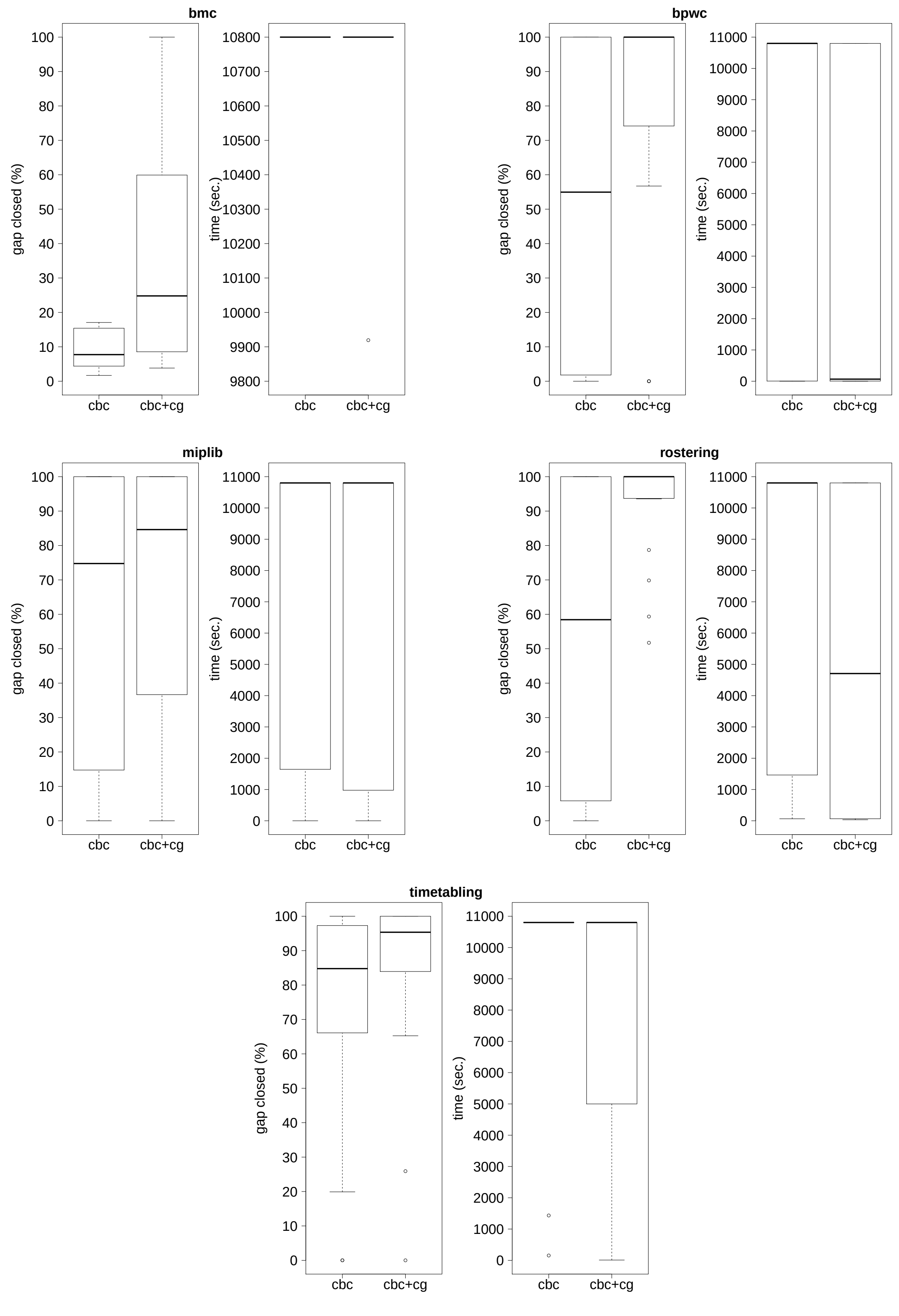


Figure 6.2: Execution times and gap closed by the two versions of CBC solver.

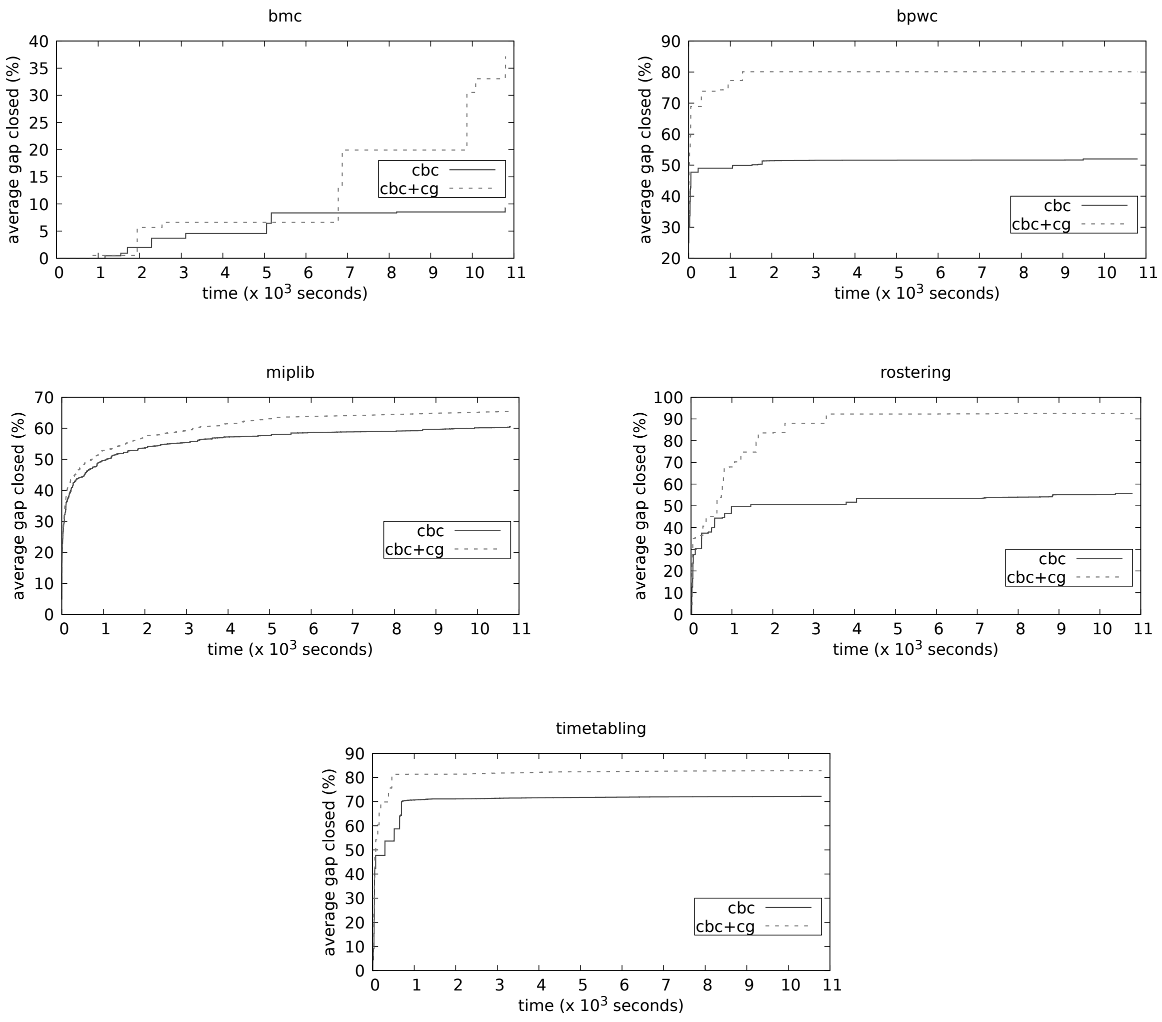


Figure 6.3: Evolution of the average gap closed over time for each instance set.

Improvements in the average gap closed of instance sets *miplib* and *timetabling* were slightly smaller. In *timetabling*, the average gap closed by CBC increased in 14.74%. The smallest improvement in the average gap closed was 8.95%, obtained in the instances of *miplib*. As observed in the previous experiments, the conflict graphs of several instances of this set have only trivial conflicts or a small set of non-trivial conflicts. Thus, conflict-based routines have difficulties to improve the dual bounds.

We also investigated the evolution of the number of instances solved by each version of CBC over time. These results are provided in Figure 6.4. The previous version of CBC was able to prove the optimality for 102 instances, while the new version proved the optimality for 126 instances. This result represents an increase of 23.53% in the number of instances solved. Moreover, the use of our conflict graph-based algorithms and data structures not only increased the number of instances solved but also decreased the execution time necessary for doing so. Considering the 24 instances in which *cbc+cg* proved the optimality and *cbc* stopped by time limit, almost half of them were solved in less than 1,000 seconds.

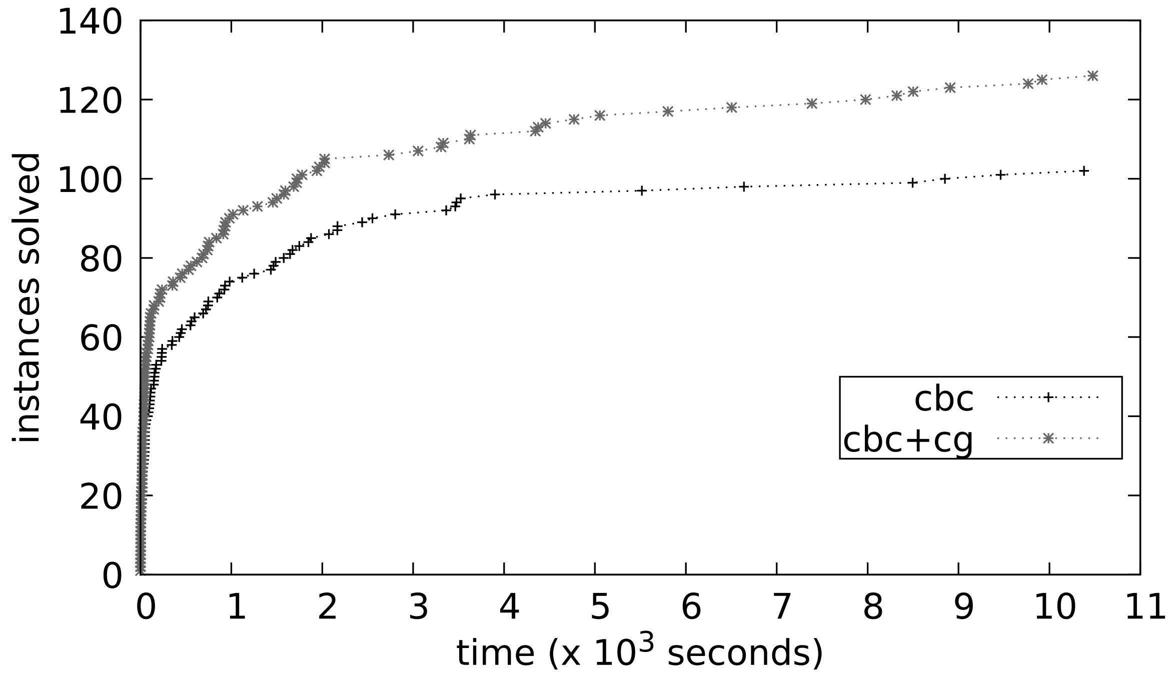


Figure 6.4: Number of instances solved over three hours.

6.2 Conclusion

The integration of our conflict graph-based algorithms and data structures in the CBC solver was presented in this chapter. The average gap closed by the new version of CBC containing our contributions was up to four times better than its previous version. Moreover, the number of MILP models solved by CBC in a time limit of three hours was increased by 23.53%. Considering the individual impact of each routine on the solution of MILP models by CBC, the most significant contribution was given by our clique cut separator.

Chapter 7

Diving Heuristics

In effective branch-and-bound algorithms, subproblems are frequently discarded for infeasibility or bounding. Considering an objective function of minimization, a subproblem whose lower bound exceeds or equals the global upper bound is discarded because it cannot lead to a better solution. Therefore, branch-and-bound algorithms benefit directly from obtaining an integer feasible solution as early as possible. An integer solution can be obtained from primal heuristics (Fischetti and Lodi, 2011) or directly from the LP relaxation of a subproblem when the integrality conditions of the variables are satisfied.

A special type of primal heuristics is called *diving heuristics* (Berthold, 2006). A diving heuristic can be seen as a depth-first-search in the branch-and-bound tree whose main goal is to construct integer feasible solutions from fractional solutions. Algorithm 7.1 illustrates a generic diving procedure. It starts constructing a set $\mathcal{D}$ containing all integer variables whose values in LP solution $\check{x}$ are fractional. While the stopping criteria are not satisfied, a variable x_j with the best score s_j is chosen from $\mathcal{D}$, and one of its bounds is updated depending on the rounding direction d_j. The process of replacing a lower bound l_j by $\lceil \check{x}_j \rceil$ is called *bounding up* the variable x_j, while replacing an upper bound u_j by $\lfloor \check{x}_j \rfloor$ is referred to as *bounding down* the variable x_j. The LP relaxation of the modified problem is then solved, and if it is infeasible, the algorithm finishes without obtaining an integer solution. Otherwise, the set $\mathcal{D}$ is updated, and if it is empty, an integer feasible solution is found and returned.

Algorithm 7.1: Generic Diving Heuristic

```
Input: LP solution $\check{x}$, rounding function $\rho$ and score function $\phi$.
Output: An integer feasible solution or $NULL$.
1  $\mathcal{D} \leftarrow \{j \in \mathcal{I} \mid \check{x}_j \notin \mathbb{Z}\}$;
2  while stopping criteria are not satisfied do
3      for $j \in \mathcal{D}$ do
4          $d_j \leftarrow \rho(j)$;
5          $s_j \leftarrow \phi(j)$;
6      Let $j$ be the index of the candidate variable with the best score $s_j$;
7      Let $l_j$ and $u_j$ be the lower and upper bounds of $x_j$, respectively;
8      if $d_j = up$ then
9          $l_j \leftarrow \lceil \check{x}_j \rceil$;
10     else
11         $u_j \leftarrow \lfloor \check{x}_j \rfloor$;
12     if LP relaxation of the modified problem is infeasible then
13         return $NULL$;
14     else
15         $\check{x} \leftarrow$ new LP solution;
16         $\mathcal{D} \leftarrow \{j \in \mathcal{I} \mid \check{x}_j \notin \mathbb{Z}\}$;
17         if $\mathcal{D} = \emptyset$ then
18             return $\check{x}$;
19 return $NULL$;
```

The diving process terminates when one of the following conditions holds:

- an integer feasible solution is found;
- the LP relaxation is infeasible after some fixations;
- some stopping criterion (e.g. time limit or iteration limit) is reached.

It is noteworthy that the algorithm can be modified to continue the search when an integer solution is found, aiming to find solutions with better quality. It is also possible to use a backtracking scheme to try different paths in the tree, thus preventing the search from stopping at the first infeasibility found.

There are several ways to define the score function ϕ. Generally, the rounding function ρ consists of bounding a variable to the direction where the best score is obtained. The combination of a rounding function and score function defines the variable selection strategy and, consequently, the diving heuristic. Some of the most common strategies are:

Coefficient Diving: selects the variable with the smallest number of potentially violated rows;

Fractional Diving: selects the variable with the lowest fractionality;

Linesearch Diving: selects the variable with the greatest difference between the first LP solution and the current LP solution;

Vectorlength Diving: selects the variable with the smallest ratio of potential objective change and number of affected constraints.

More details about diving heuristics are presented in the work of Berthold (2006).

The following section presents two diving heuristics that we proposed and implemented. These heuristics consider the information extracted from conflict graphs to define their variable selection strategies.

7.1 Conflict-Based Diving Heuristics

Some diving heuristics use the concept of *down-locks* and *up-locks* to define their variable selection strategy. The number of down-locks of a variable corresponds to the number of constraints that this variable appears with negative coefficients (Berthold, 2006). On the other hand, the number of constraints in which a variable appears with positive coefficients defines its up-locks.

Given these definitions, a variable with zero up-locks (down-locks) can always be bounded up (bounded down) without increasing the current violation of any constraint. However, bounding up (bounding down) a variable whose number of up-locks (down-locks) is greater than zero might increase the current violation of the constraints. Therefore, a variable lock is a value that indicates the risk of increasing the constraint violations when bounding a variable in a certain direction.

Coefficient diving is an example of a diving heuristic that employs the concept of locks. It selects, at each iteration, the variable with the smallest number of locks and defines the rounding direction as the direction that this minimum value is obtained.

Contrary to this idea, we proposed and implemented two conflict-based diving heuristics that prefer selecting the variables with the highest risk of generating infeasibilities. A strategy that considers taking the most critical decisions first is called *fail fast strategy* (Berthold, 2014). In the context of primal heuristics, fail fast strategies have two advantages. Firstly, it is probably easier to repair infeasibility when there are a small number of fixed variables. Secondly, the failure of a heuristic caused by an early decision may avoid spending much running time.

An example of a diving heuristic that employs the fail fast strategy is proposed by Witzig and Gleixner (2019). Whenever an infeasibility is detected during the diving process, it is analyzed, the corresponding conflict constraint is stored and the algorithm performs 1-level backtracking. With this mechanism, several conflict constraints are dynamically discovered during the diving iterations. This information is then used to develop a conflict diving heuristic, which consists of selecting the variable that most appears in the conflict constraints. Unlike this approach, our diving heuristics considers the conflicts provided from the conflict graph.

7.1.1 Conflict Diving

The first diving heuristic that we proposed analyzes the number of conflicts (i.e., the degree) of the variables with respect to the conflict graph. For this, we consider the *conflict locks* of the variables. The number of *conflict up-locks* of a variable corresponds to the degree of its corresponding vertex in the conflict graph. On the other hand, the number of *conflict down-locks* of a variable is related to the degree of the vertex that corresponds to the complement of this variable. In this case, conflict locks estimate the risk of generating infeasibility when bounding a variable in a certain direction.

The rounding function ρ_{cnf} of Conflict Diving prefers the direction that is more likely to lead to infeasibilities. Thus, the rounding direction of a given variable x_j is defined as:

$$\rho(j)_{cnf} = \begin{cases} \textit{down} & \text{if } \underline{\zeta}_j > \overline{\zeta}_j, \\ \textit{up} & \text{if } \overline{\zeta}_j > \underline{\zeta}_j, \\ \textit{down} & \text{if } \overline{\zeta}_j = \underline{\zeta}_j \text{ and } f_j < 0.5, \\ \textit{up} & \text{if } \overline{\zeta}_j = \underline{\zeta}_j \text{ and } f_j \geq 0.5, \end{cases}$$

where $\underline{\zeta}_j$ and $\overline{\zeta}_j$ are the number of conflict down-locks and conflict up-locks of x_j, respectively. When the number of conflict up-locks and conflict down-locks are equal, we consider the fractional part $f_j = \check{x}_j - \lfloor \check{x}_j \rfloor$ of variable x_j in the current LP solution $\check{x}$ to choose the rounding direction.

In Conflict Diving, the score function ϕ_{cnf} prefers variables that have a large number of locks on the chosen rounding direction. The score $\phi_{cnf}(j)$ for a variable x_j is given by:

$$\phi(j)_{cnf} = \begin{cases} \underline{\zeta}_j & \text{if } \rho(j) = \textit{down}, \\ \overline{\zeta}_j & \text{if } \rho(j) = \textit{up}. \end{cases}$$

Thus, the diving candidates are explored in the non-increasing order of their conflict locks.

It is important to note that the score function ϕ_{cnf} always returns zero for general integer variables since conflict graphs are composed only by binary variables. Furthermore, ϕ_{cnf} always returns one for variables that have only trivial conflicts. As a consequence, in some cases, the best score of the diving candidates can be less than or equal to one, indicating that Conflict Diving does not have sufficient information to choose the variable that will be bounded. In this situation, we use the variable selection strategy of Linesearch Diving (Berthold, 2008).

The rounding direction of a variable x_j considering Linesearch Diving is defined as:

$$\rho(j)_{lns} = \begin{cases} down & \text{if } \breve{x}_j < \breve{x}_j^R, \\ up & \text{if } \breve{x}_j > \breve{x}_j^R, \\ down & \text{if } \breve{x}_j = \breve{x}_j^R \text{ and } f_j < 0.5, \\ up & \text{if } \breve{x}_j = \breve{x}_j^R \text{ and } f_j \geq 0.5, \end{cases}$$

were $\breve{x}^R$ is the solution of the root node LP relaxation and $\breve{x}$ is the LP solution of the current node. When $\breve{x}_j$ are equal to $\breve{x}_j^R$, we consider the fractional part $f_j = \breve{x}_j - \lfloor \breve{x}_j \rfloor$ of variable x_j in the current LP solution $\breve{x}$ to choose the rounding direction. In Linesearch Diving, the score function ϕ_{lns} prefers the variables that have the greatest difference between the solution of the root node LP relaxation and the current LP solution on the chosen rounding direction. Therefore, the score $\phi_{lns}(j)$ for a variable x_j is given by:

$$\phi(j)_{lns} = \begin{cases} \frac{\breve{x}_j - \lfloor \breve{x}_j \rfloor}{\breve{x}_j^R - \breve{x}_j} & \text{if } \rho(j) = down, \\ \frac{\lceil \breve{x}_j \rceil - \breve{x}_j}{\breve{x}_j - \breve{x}_j^R} & \text{if } \rho(j) = up. \end{cases}$$

Algorithm 7.2 presents the proposed Conflict Diving Heuristic. It starts constructing a set $\mathcal{D}$ containing all integer variables whose values in LP solution $\breve{x}$ are fractional. In each iteration of the algorithm, the best score s_{best} of the variables in $\mathcal{D}$ is computed, considering the score function ϕ_{cnf} (lines 4 to 9). If s_{best} is less than or equal to one, the algorithm uses rounding function ρ_{lns} and score function ϕ_{lns} of Linesearch Diving (lines 10 to 13). Then, the variable with the best score is chosen and one of its bounds is updated according to the rounding direction (lines 14 to 19). After changing the bound of a binary variable (i.e., fixing it), it is possible to propagate this change by using the information from the conflict graph (line 20). The propagation of the bound change reduces the number of diving iterations and, consequently, decreases the running time of the heuristic. The LP relaxation of the modified problem is then solved. If it is infeasible, the algorithm finishes without obtaining an integer solution. Otherwise, solution $\breve{x}$ and set $\mathcal{D}$ are updated. If $\mathcal{D}$ is empty, an integer feasible solution is found and returned. Otherwise, another iteration of the algorithm is performed.

Algorithm 7.2: Conflict Diving Heuristic

```
Input: Conflict graph $G$, LP solution $\check{x}^R$ of the root node, rounding function
       $\rho_{cnf}$, rounding function $\rho_{lns}$, score function $\phi_{cnf}$ and score function $\phi_{lns}$.
Output: An integer feasible solution or $NULL$.
1  $\check{x} \leftarrow \check{x}^R$;
2  $\mathcal{D} \leftarrow \{j \in \mathcal{I} \mid \check{x}_j \notin \mathbb{Z}\}$;
3  while stopping criteria are not satisfied do
4      $s_{best} \leftarrow 0$;
5      for $j \in \mathcal{D}$ do
6          $d_j \leftarrow \rho_{cnf}(j)$;
7          $s_j \leftarrow \phi_{cnf}(j)$;
8          if $s_j > s_{best}$ then
9              $s_{best} \leftarrow s_j$;
10     if $s_{best} \leq 1$ then
11         for $j \in \mathcal{D}$ do
12             $d_j \leftarrow \rho_{lns}(j)$;
13             $s_j \leftarrow \phi_{lns}(j)$;
14     Let $j$ be the index of the candidate variable with the best score $s_j$;
15     Let $l_j$ and $u_j$ be the lower and upper bounds of $x_j$, respectively;
16     if $d_j = up$ then
17         $l_j \leftarrow \lceil \check{x}_j \rceil$;
18     else
19         $u_j \leftarrow \lfloor \check{x}_j \rfloor$;
20     Propagate this bound change using conflict graph $G$;
21     if LP relaxation of the modified problem is infeasible then
22         return $NULL$;
23     else
24         $\check{x} \leftarrow$ new LP solution;
25         $\mathcal{D} \leftarrow \{j \in \mathcal{I} \mid \check{x}_j \notin \mathbb{Z}\}$;
26         if $\mathcal{D} = \emptyset$ then
27             return $\check{x}$;
28 return $NULL$;
```

7.1.2 Modified Degree Diving

The second conflict-based diving heuristic that we proposed and implemented is called *Modified Degree Diving*. In this heuristic, the definition of conflict locks is related to the modified degree of the variables. The number of *conflict up-locks* of a variable is the modified degree of its corresponding vertex in the conflict graph. On the other hand, the number of *conflict down-locks* of a variable is related to the modified degree of the vertex that corresponds to the complement of this variable.

The rounding function ρ_{mdg} of Modified Degree Diving prefers the direction that has the highest modified degree. Thus, the rounding direction of a given variable x_j is defined as:

$$\rho(j)_{mdg} = \begin{cases} \mathit{down} & \text{if } \underline{\xi}_j > \overline{\xi}_j, \\ \mathit{up} & \text{if } \overline{\xi}_j > \underline{\xi}_j, \\ \mathit{down} & \text{if } \overline{\xi}_j = \underline{\xi}_j \text{ and } f_j < 0.5, \\ \mathit{up} & \text{if } \overline{\xi}_j = \underline{\xi}_j \text{ and } f_j \geq 0.5, \end{cases}$$

where $\underline{\xi}_j$ and $\overline{\xi}_j$ are the number of conflict down-locks and conflict up-locks of x_j, respectively. When the number of conflict up-locks and conflict down-locks are equal, we consider the fractional part $f_j = \check{x}_j - \lfloor \check{x}_j \rfloor$ of variable x_j in the current LP solution $\check{x}$ to choose the rounding direction.

The score function ϕ_{mdg} of Modified Degree Diving prioritizes the variables that have a large number of locks on the chosen rounding direction. The score $\phi_{mdg}(j)$ for a variable x_j is given by:

$$\phi(j)_{mdg} = \begin{cases} \underline{\xi}_j & \text{if } \rho(j) = \mathit{down}, \\ \overline{\xi}_j & \text{if } \rho(j) = \mathit{up}. \end{cases}$$

Similar to Conflict Diving, the diving candidates in Modified Degree Diving are explored in the non-increasing order of their conflict locks.

The score function ϕ_{mdg} always returns zero for general integer variables and two for variables that have only trivial conflicts (both variable and its complement have their degrees equal to one). When the best score of the diving candidates is less than or equal to two, we use the variable selection strategy of Linesearch Diving in the same way as used in Conflict Diving. The algorithm of Modified Degree Diving can be obtained by replacing the rounding function ρ_{cnf} by ρ_{mdg} and the score function ϕ_{cnf} by ϕ_{mdg} in Algorithm 7.2.

7.2 Computational Results

We evaluated our conflict-based diving heuristics concerning the ability to generate feasible integer solutions. For this, we ran Conflict Diving and Modified Degree Diving for each instance problem presented in Section 2.5, stopping the executions when one of the following conditions holds:

- an integer feasible solution is found;
- the LP relaxation is infeasible;
- execution time reached $10,800$ seconds.

For comparison purposes, we implemented four of the most common diving heuristics: Fractional Diving, Coefficient Diving, Linesearch Diving and Vectorlength Diving (Berthold, 2006). These heuristics only require an initial LP relaxation and the general structure of the MILP model itself to be executed.

All the diving heuristics were implemented in C++ programming language and compiled with g++ version 7.4.0. The experiment was carried out on four computers with Intel Core i7-4790 3.60 GHz processors and 32 GB of RAM running Ubuntu Linux version 18.04 64-bit. Table 7.1 presents the results obtained in this experiment. Detailed results are available for download at `http://professor.ufop.br/samuelbrito/thesis`.

Table 7.1: Summarized results of the conflict-based diving heuristics.

strategy	solution found	best solution	only solution	avg time
coefficient	112	69	**16**	823.26
fractional	87	53	2	728.87
linesearch	140	81	14	383.03
vectorlength	117	52	1	508.20
conflict	137	71	3	**352.79**
modified degree	**150**	**83**	14	370.75

There were 68 instances for which all the diving heuristics found a feasible solution. The diving heuristic that found the greatest number of feasible solutions was the Modified Degree Diving, which was able to produce by up 72.41% more solutions than the other heuristics. In each iteration of this heuristic, when the fractional variables have only trivial conflicts, the variable selection strategy of Linesearch was performed and contributed to generate feasible solutions for several instances. On the other hand, when the fractional variables also have non-trivial conflicts, the variable selection strategy that considers the modified degree was executed and could guide the heuristic in the process of finding feasible solutions. Even though this heuristic does not consider the objective coefficient of the variables, it found the best solutions for 83 instances.

The number of feasible solutions obtained by Conflict Diving is slightly less than Linesearch Diving. For some instances, selecting the variables with the highest degrees generated infeasibilities in the early diving iterations.

Conflict Diving and Modified Degree Diving obtained the best execution times. In instances with dense conflict graphs, performing a conflict propagation after changing the bound of a variable can considerably reduce the number of diving iterations, decreasing the running time of the heuristics. In addition, choosing the most conflicting variable can generate infeasibilities in the first diving iterations. These results corroborate with the idea that taking the most critical decisions first is a good strategy for solving MILP models (Berthold, 2014).

The relation between the number instances where a feasible solution was found and the execution time of the heuristics is presented in Figure 7.1. In this figure it is possible to observe that Modified Degree Diving generated a greater number of feasible solutions, spending smaller execution times when compared with the other diving heuristics.

Most of the feasible solutions found by the diving heuristics were discovered in the earlier execution times. For example, the feasible solutions for 125 of 150 instances found by Modified Degree diving were generated in less than 100 seconds.

As the results show, the conflict-based diving heuristics proved to be reasonable algorithms to find feasible solutions. Hence, they can be inserted into a MILP solver to provide initial solutions after running the branch-and-cut algorithm.

7.3 Conclusion

We have shown in this chapter two diving heuristics that use information from conflict graphs to generate feasible solutions for MILP models. These heuristics employ the concept of fail fast strategy, first adjusting the bounds of the variables that are more

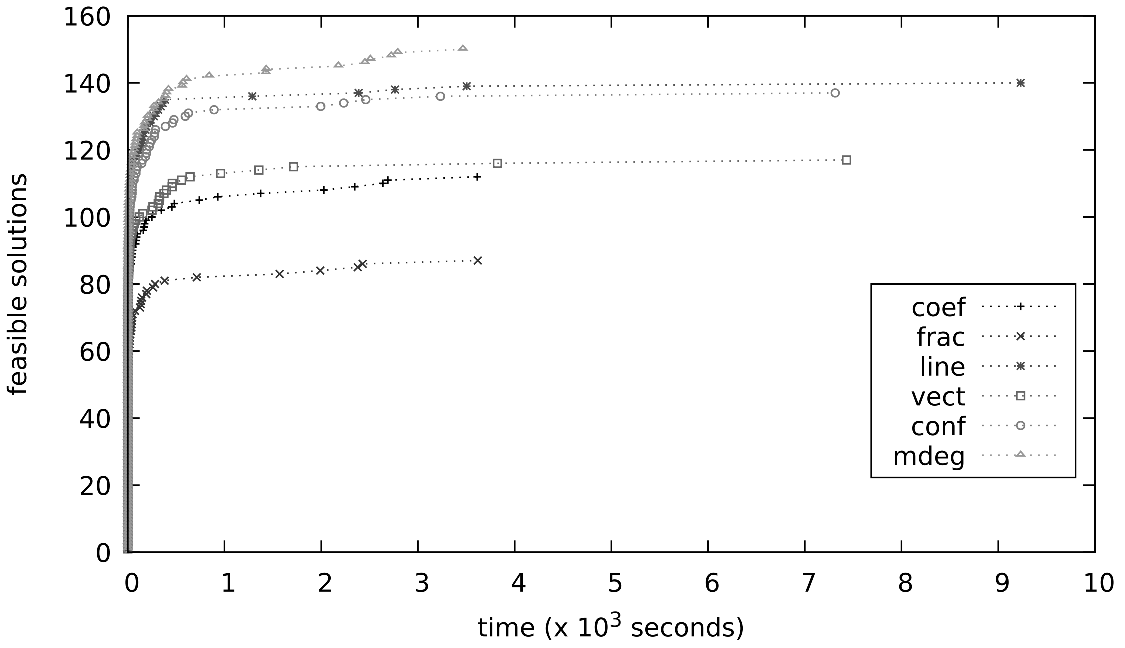


Figure 7.1: Number of feasible solutions found by each diving heuristic over the time.

likely to cause infeasibilities. Both proposed diving heuristics presented execution times smaller than the classical diving heuristics that we evaluated in our experiments. Moreover, the heuristic that uses the modified degree in its variable selection strategy found the greatest number of feasible solutions among the diving heuristics evaluated.

Chapter 8

Final Considerations

This thesis presented conflict graph-based algorithms for Mixed-Integer Linear Programming problems. We developed a conflict graph infrastructure, characterized by the efficient construction and storage of such graphs. Our algorithm for building conflict graphs is an improved version of a state-of-the-art conflict detection algorithm that extracts cliques from MILP model constraints. We included an additional step in this algorithm that detects additional maximal cliques without changing the worst-case complexity. Optimized data structures that selectively store conflicts pairwise or grouped in cliques were also developed. Our conflict graph infrastructure was able to construct graphs even for instances with a large number of conflicts.

Conflict graphs were then used in the implementation of a preprocessing algorithm and two cut separators. The preprocessing algorithm is based on a clique merging procedure that combines several set packing constraints into a single constraint. Significant improvements with respect to the dual bounds of the problems were obtained, especially for MILP models with several constraints expressed by a small number of conflicting variables. Our preprocessing routine was responsible for reducing the number of constraints, strengthening the initial dual bounds and for accelerating the process of proving optimality for a great number of instances.

The two conflict-based cut separators that we developed are responsible for separating cliques and odd cycles. Our clique cut separator obtained better dual bounds than those provided by the equivalent cut generators of CBC, GLPK and CPLEX solvers. As previous works shows, the inclusion of odd cycle cuts had no significant impact on the dual bound improvement. However, the cost for separating these cuts is low, which means that it can be included in a cutting plane strategy without a significant increase in execution time.

A new version of the CBC solver was generated, including our conflict graph infrastructure, preprocessing routine and cut separators. Experiments with this new version revealed an improvement in the average gap closed of 16.88% in comparison to the previous version of the CBC solver. Furthermore, time spent to prove optimality for the instances decreased, while the number of instances solved increased from 102 to 126. For instance sets containing MILP models of Bin Packing with Conflicts, Nurse Rostering, Bandwidth Multicoloring and Educational Timetabling, the average gap closed by the new version of CBC was up to four times better than its previous version.

Two conflict-based diving heuristics were also proposed and developed in this thesis. These heuristics tend to select first the variables that are most likely to produce infeasibilities. One heuristic considers the degree and the other uses the modified degree of the variables at the conflict graph. Both diving heuristics presented execution times smaller than the classical diving heuristics that we compared in our experiments. Moreover, the heuristic that uses the modified degree in its variable selection strategy found the greatest number of feasible solutions among those considered in the experiments.

8.1 Further Research

Regarding the construction and use of conflict graphs explored in this thesis, some aspects may be further investigated:

- the number of conflicts in the graphs could be augmented by using constraint propagation techniques;
- other exact algorithms and heuristics could be employed to develop new clique cut separation routines, in order to compare them with the performance of the BK algorithm;
- conflict graphs could be used to strengthen other families of cuts, such as knapsack inequalities;
- machine learning techniques could be used to decide when to activate or deactivate the preprocessing and cut separators since for some cases they cannot improve dual bounds;

- node selection strategies based on the conflict graphs could be designed, in order to accelerate the process of solving MILP models;
- machine learning techniques could be used to choose the best diving heuristic for a given problem;
- logical relations of conflict graphs could be used to develop improvement heuristics; these heuristics could be performed during some steps of branch-and-cut in order to improve the incumbent solution.

Bibliography

Achterberg, T.: 2007, *Constraint Integer Programming*, PhD thesis, Technische Universitat Berlin, Berlin, Germany.

Achterberg, T., Bixby, R. E., Gu, Z., Rothberg, E. and Weninger, D.: 2016, Presolve reductions in mixed integer programming, *Technical Report 16-44*, Zuse Institute Berlin, Berlin.

Achterberg, T. and Wunderling, R.: 2013, Mixed integer programming: Analyzing 12 years of progress, *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 449–481.

Ahuja, R. K., Özlem Ergun, Orlin, J. B. and Punnen, A. P.: 2002, A survey of very large-scale neighborhood search techniques, *Discrete Applied Mathematics* **123**(1), 75 – 102.

Applegate, D. L., Bixby, R. E., Chvatal, V. and Cook, W. J.: 2006, *The traveling salesman problem: a computational study*, Princeton university press.

Araujo, J. A., Santos, H. G., Gendron, B., Jena, S. D., Brito, S. S. and Souza, D. S.: 2020, Strong bounds for resource constrained project scheduling: Preprocessing and cutting planes, *Computers & Operations Research* **113**, 104782.

Atamtürk, A., Nemhauser, G. L. and Savelsbergh, M. W.: 2000, Conflict graphs in solving integer programming problems, *European Journal of Operational Research* **121**(1), 40 – 55.

Bäck, T., Fogel, D. B. and Michalewicz, Z.: 1997, *Handbook of evolutionary computation*, CRC Press.

Balas, E., Ceria, S., Dawande, M., Margot, F. and Pataki, G.: 2001, Octane: A new heuristic for pure 0–1 programs, *Operations Research* **49**(2), 207–225.

Belotti, P., Lee, J., Liberti, L., Margot, F. and Wächter, A.: 2009, Branching and bounds tightening techniques for non-convex minlp, *Optimization Methods and Software* **24**(4-5), 597–634.

Berthold, T.: 2006, *Primal heuristics for mixed integer programs*, diploma thesis, Technische Universitat Berlin, Berlin, Germany.

Berthold, T.: 2008, Heuristics of the branch-cut-and-price-framework scip, *Operations Research Proceedings 2007*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 31–36.

Berthold, T.: 2014, *Heuristic algorithms in global MINLP solvers*, PhD thesis, Technische Universitat Berlin, Berlin, Germany.

Bixby, R. E. and Lee, E. K.: 1998, Solving a truck dispatching scheduling problem using branch-and-cut, *Operations Research* **46**(3), 355–367.

Bixby, R. and Rothberg, E.: 2007, Progress in computational mixed integer programming—a look back from the other side of the tipping point, *Annals of Operations Research* **149**(1), 37–41.

Bonami, P., Biegler, L. T., Conn, A. R., Cornuéjols, G., Grossmann, I. E., Laird, C. D., Lee, J., Lodi, A., Margot, F., Sawaya, N. and Wächter, A.: 2008, An algorithmic framework for convex mixed integer nonlinear programs, *Discrete Optimization* **5**(2), 186–204.

Borndorfer, R.: 1998, *Aspects of Set Packing, Partitioning, and Covering*, PhD thesis, Technische Universitat Berlin, Berlin, Germany.

Boschetti, M. A., Maniezzo, V., Roffilli, M. and Bolufé Röhler, A.: 2009, Matheuristics: Optimization, simulation and control, *in* M. J. Blesa, C. Blum, L. Di Gaspero, A. Roli, M. Sampels and A. Schaerf (eds), *Hybrid Metaheuristics*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 171–177.

Brearley, A. L., Mitra, G. and Williams, H. P.: 1975, Analysis of mathematical programming problems prior to applying the simplex algorithm, *Mathematical Programming* **8**(1), 54–83.

Brito, S. S.: 2015, *Conflict graphs: Construction and applications in integer programming problems (in portuguese)*, Master's thesis, Universidade Federal de Ouro Preto, Ouro Preto, Brazil.

Bron, C. and Kerbosch, J.: 1973, Algorithm 457: finding all cliques of an undirected graph, *Commun. ACM* **16**(9), 575–577.

Burke, E. K., Mareček, J., Parkes, A. J. and Rudová, H.: 2012, A branch-and-cut procedure for the udine course timetabling problem, *Annals of Operations Research* **194**(1), 71–87.

Cornuéjols, G.: 2007, Revival of the gomory cuts in the 1990's, *Annals of Operations Research* **149**(1), 63–66.

Danna, E., Rothberg, E. and Pape, C. L.: 2005, Exploring relaxation induced neighborhoods to improve mip solutions, *Mathematical Programming* **102**(1), 71–90.

Dias, B., de Freitas, R., Maculan, N. and Michelon, P.: 2016, Constraint and integer programming models for bandwidth coloring and multicoloring in graphs, *Proceedings of the XLVIII Brazilian Symposium on Operations Research*, pp. 4116–4127.

Fischetti, M., Glover, F. and Lodi, A.: 2005, The feasibility pump, *Mathematical Programming* **104**(1), 91–104.

Fischetti, M. and Lodi, A.: 2003, Local branching, *Mathematical Programming* **98**(1), 23–47.

Fischetti, M. and Lodi, A.: 2007, Optimizing over the first chvátal closure, *Mathematical Programming B* **110**(1), 3–20.

Fischetti, M. and Lodi, A.: 2011, Heuristics in mixed integer programming, *Wiley Encyclopedia of Operations Research and Management Science*, Vol. 8, John Wiley & Sons, pp. 738–747.

Fonseca, G. H., Santos, H. G., Carrano, E. G. and Stidsen, T. J.: 2017, Integer programming techniques for educational timetabling, *European Journal of Operational Research* **262**(1), 28 – 39.

Gamrath, G., Koch, T., Martin, A., Miltenberger, M. and Weninger, D.: 2015, Progress in presolving for mixed integer programming, *Mathematical Programming Computation* **7**(4), 367–398.

Garey, M. R. and Johnson, D. S.: 1979, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York.

Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P., Jarck, K., Koch, T., Linderoth, J., Lübbecke, M., Mittelmann, H. D., Ozyurt, D., Ralphs, T. K., Salvagnin, D. and Shinano, Y.: 2018, MIPLIB 2017. `http://miplib.zib.de`.

Glover, F. and Laguna, M.: 1997a, General purpose heuristics for integer programming—part i, *Journal of Heuristics* **2**(4), 343–358.

Glover, F. and Laguna, M.: 1997b, General purpose heuristics for integer programming—part ii, *Journal of Heuristics* **3**(2), 161–179.

Glover, F. and Laguna, M.: 1998, *Tabu Search*, Springer US, Boston, MA, pp. 2093–2229.

Gonçalves, L. C. N. I. and Santos, H. G.: 2008, Optimization in mass higher education institutions: a tactical approach using aps concepts (in portuguese), *Anais do XLIII Simpósio Brasileiro de Pesquisa Operacional*, pp. 692–703.

Grotschel, M., Lovasz, L. and Schrijver, A.: 1993, *Geometric Algorithms and Combinatorial Optimization*, Springer.

Hansen, P. and Mladenović, N.: 1999, *An Introduction to Variable Neighborhood Search*, Springer US, Boston, MA, pp. 433–458.

Hansen, P., Mladenović, N. and Urošević, D.: 2006, Variable neighborhood search and local branching, *Computers & Operations Research* **33**(10), 3034 – 3045. Part Special Issue: Constraint Programming.

Haspeslagh, S., De Causmaecker, P., Schaerf, A. and Stølevik, M.: 2014, The first international nurse rostering competition 2010, *Annals of Operations Research* **218**(1), 221–236.

Hoffman, K. and Padberg, M.: 1993, Solving airline crew scheduling problems by branch-and-cut, *Management Science* **39**(6), 657–682.

Johnson, E. L. and Nemhauser, G. L.: 1992, Recent developments and future directions in mathematical programming, *IBM Systems Journal* **31**(1), 79–93.

Jünger, M., Liebling, T. M., Naddef, D., Nemhauser, G. L., Pulleyblank, W. R., Reinelt, G., Rinaldi, G. and Wolsey, L. A.: 2009, *50 Years of integer programming 1958-2008: From the early years to the state-of-the-art*, Springer Science & Business Media.

Kirkpatrick, S., Gelatt, C. D. and Vecchi, M. P.: 1983, Optimization by simulated annealing, *Science* **220**(4598), 671–680.

Kolliopoulos, S. G. and Young, N. E.: 2005, Approximation algorithms for covering/packing integer programs, *Journal of Computer and System Sciences* **71**(4), 495 – 505.

Land, A. H. and Doig, A. G.: 1960, An automatic method of solving discrete programming problems, *Econometrica: Journal of the Econometric Society* pp. 497–520.

Lawler, E. L. and Wood, D. E.: 1966, Branch-and-bound methods: A survey, *Operations Research* **14**(4), 699–719.

Lenstra, J. K., Shmoys, D. B. and Tardos, E.: 1990, Approximation algorithms for scheduling unrelated parallel machines, *Mathematical programming* **46**(1-3), 259–271.

Lokketangen, A. and Glover, F.: 1998, Solving zero-one mixed integer programming problems using tabu search, *European Journal of Operational Research* **106**(2), 624 – 658.

Méndez-Díaz, I. and Zabala, P.: 2008, A cutting plane algorithm for graph coloring, *Discrete Applied Mathematics* **156**, 159–179.

Mészáros, C. and Suhl, U. H.: 2003, Advanced preprocessing techniques for linear and quadratic programming, *OR Spectrum* **25**(4), 575–595.

Orlowski, S., Wessäly, R., Pióro, M. and Tomaszewski, A.: 2010, Sndlib 1.0 survivable network design library, *Networks* **55**(3), 276–286.

Padberg, M.: 1973, On the facial structure of set packing polyhedra, *Mathematical Programming* **5**(1), 199–215.

Pecin, D., Pessoa, A., Poggi, M. and Uchoa, E.: 2017, Improved branch-cut-and-price for capacitated vehicle routing, *Mathematical Programming Computation* **9**(1), 61–100.

Pochet, Y. and Wolsey, L. A.: 2006, *Production planning by mixed integer programming*, Springer Science & Business Media.

Rebennack, S.: 2009, Stable set problem: Branch & cut algorithms, *in* C. A. Floudas and P. M. Pardalos (eds), *Encyclopedia of Optimization*, Springer US, pp. 3676–3688.

Rossi, F., Van Beek, P. and Walsh, T.: 2006, *Handbook of constraint programming*, Elsevier.

Rossi, R. A. and Zhou, R.: 2018, Graphzip: a clique-based sparse graph compression method, *Journal of Big Data* **5**(1), 10.

Sadykov, R. and Vanderbeck, F.: 2013, Bin packing with conflicts: A generic branch-and-price algorithm, *INFORMS Journal on Computing* **25**(2), 244–255.

Santos, H. G., Toffolo, T. A. M., Gomes, R. A. M. and Ribas, S.: 2016, Integer programming techniques for the nurse rostering problem, *Annals of Operations Research* **239**(1), 225–251.

Savelsbergh, M. W. P.: 1994, Preprocessing and probing techniques for mixed integer programming problems, *ORSA Journal on Computing* **6**(4), 445–454.

Segundo, P. S., Artieda, J. and Strash, D.: 2018, Efficiently enumerating all maximal cliques with bit-parallelism, *Computers & Operations Research* **92**, 37–46.

Tomita, E., Tanaka, A. and Takahashi, H.: 2006, The worst-case time complexity for generating all maximal cliques and computational experiments, *Theoretical Computer Science* **363**(1), 28 – 42. Computing and Combinatorics.

Toth, P. and Vigo, D.: 2002, *An Overview of Vehicle Routing Problems*, Society for Industrial and Applied Mathematics, pp. 1–26.

Van Roy, T. J. and Wolsey, L. A.: 1987, Solving mixed integer programming problems using automatic reformulation, *Operations Research* **35**(1), 45–57.

Witzig, J. and Gleixner, A.: 2019, Conflict-driven heuristics for mixed integer programming, *Technical Report 19-08*, Zuse Institute Berlin, Berlin, Germany.

Xu, J., Li, M., Kim, D. and Xu, Y.: 2003, Raptor: Optimal protein threading by linear programming, *Journal of Bioinformatics and Computational Biology* **01**(01), 95–117.

Appendix A

Detailed Results of the Computational Experiments

Table A.1 presents the characteristics of the mixed-integer linear programs used in the computational experiments. Columns “set” and “cols” present the instance set and the number of variables of each instance, respectively. Columns “int”, “bin” and “con” contain the number of integer, binary and continuous variables of each problem instance. Columns “rows” and “nz” detail information with respect to the number of constraints and nonzeros coefficients of each instance. Finally, column “cg_ρ” presents the density of the conflict graph constructed by our algorithm for each instance.

Table A.1: Instance set characteristics.

name	**set**	**cols**	**int**	**bin**	**con**	**rows**	**nz**	cg_ρ **(×100)**
30n20b8	miplib	18,380	62	18,318	0	576	109,706	0.85
50v-10	miplib	2,013	183	1,464	366	233	2,745	0.03
air03	miplib	10,757	0	10,757	0	124	91,028	13.87
air04	miplib	8,904	0	8,904	0	823	72,965	1.34
air05	miplib	7,195	0	7,195	0	426	52,121	2.44
app1-1	miplib	2,480	0	1,225	1,255	4,926	18,275	0.04
app1-2	miplib	26,871	0	13,300	13,571	53,467	199,175	0.00
assign1-5-8	miplib	156	0	130	26	161	3,720	1.16
atlanta-ip	miplib	48,738	106	46,667	1,965	21,732	257,532	0.00
b1c1s1	miplib	3,872	0	288	3,584	3,904	11,408	0.17
bab1	miplib	61,152	0	61,152	0	60,680	854,392	0.08
bab2	miplib	147,912	0	147,912	0	17,245	2,027,726	0.00
bab3	miplib	393,800	0	393,800	0	23,069	3,301,838	0.01
bab5	miplib	21,600	0	21,600	0	4,964	155,520	0.03
bab6	miplib	114,240	0	114,240	0	29,904	1,283,181	0.00
beasleyC3	miplib	2,500	0	1,250	1,250	1,750	5,000	0.04
binkar10_1	miplib	2,298	0	170	2,128	1,026	4,496	0.29
blp-ar98	miplib	16,021	0	15,806	215	1,128	200,601	0.03

Table A.1: Instance set characteristics (continued).

name	set	cols	int	bin	con	rows	nz	cg_ρ (×100)
blp-ic98	miplib	13,640	0	13,550	90	717	191,947	0.04
bnatt400	miplib	3,600	0	3,600	0	5,614	21,698	0.04
bppc4-08	miplib	1,456	0	1,454	2	111	23,964	1.29
br1	timetabling	1,344	0	1,344	0	3,243	12,904	0.17
br2	timetabling	4,284	0	4,284	0	9,248	42,584	0.10
br3	timetabling	6,368	0	6,368	0	12,056	61,408	0.08
br5	timetabling	19,468	0	19,468	0	28,981	221,123	0.11
brazil3	timetabling	23,968	94	23,874	0	14,646	133,184	0.06
chromaticindex1024-7	miplib	73,728	0	73,728	0	67,583	270,324	0.00
chromaticindex512-7	miplib	36,864	0	36,864	0	33,791	135,156	0.01
cmflsp50-24-8-8	miplib	16,392	0	1,392	15,000	3,520	158,622	0.07
CMS750_4	miplib	11,697	0	7,196	4,501	16,381	44,903	0.01
co-100	miplib	48,417	0	48,417	0	2,187	1,995,817	6.26
cod105	miplib	1,024	0	1,024	0	1,024	57,344	9.45
comp07-2idx	timetabling	17,264	109	17,155	0	21,235	86,577	0.01
comp21-2idx	timetabling	10,863	71	10,792	0	14,038	57,301	0.01
cost266-UUE	miplib	4,161	0	171	3,990	1,446	12,312	0.59
csched007	miplib	1,758	0	1,457	301	351	6,379	0.64
csched008	miplib	1,536	0	1,284	252	351	5,687	0.61
cvs16r128-89	miplib	3,472	0	3,472	0	4,633	12,528	0.08
d_BPWC_2_4_10	bpwc	22,824	0	22,824	0	287,902	620,453	0.18
da_BPWC_2_8_6	bpwc	6,475	0	6,475	0	83,612	179,175	0.24
da_BPWC_2_8_9	bpwc	6,475	0	6,475	0	82,878	177,707	0.24
dano3_3	miplib	13,873	0	69	13,804	3,202	79,655	0.73
dano3_5	miplib	13,873	0	115	13,758	3,202	79,655	0.44
drayage-100-23	miplib	11,090	0	11,025	65	4,630	41,550	0.48
drayage-25-23	miplib	11,090	0	11,025	65	4,630	41,550	0.48
ds	miplib	67,732	0	67,732	0	656	1,024,059	3.44
dws008-01	miplib	11,096	0	6,608	4,488	6,064	56,400	0.26
eil33-2	miplib	4,516	0	4,516	0	32	44,243	23.62
eilA101-2	miplib	65,832	0	65,832	0	100	959,373	20.90
eilA76	miplib	1,422	0	1,422	0	75	10,967	11.60
eilB101	miplib	2,818	0	2,818	0	100	24,120	12.87
eilB101.2	miplib	53,444	0	53,444	0	100	577,946	17.39
eilB76	miplib	1,060	0	1,060	0	75	6,296	8.15
eilC76	miplib	1,644	0	1,644	0	75	14,673	13.40
eilC76.2	miplib	28,599	0	28,599	0	75	314,837	18.38
eilD76	miplib	1,898	0	1,898	0	75	19,111	15.02
eilD76.2	miplib	30,588	0	30,588	0	75	381,749	20.08
enlight_hard	miplib	200	100	100	0	100	560	0.50
exp-1-500-5-5	miplib	990	0	250	740	550	1,980	0.60
fast0507	miplib	63,009	0	63,009	0	507	409,349	0.00
fastxgemm-n2r6s0t2	miplib	784	0	48	736	5,998	19,376	1.05
fiball	miplib	34,219	258	33,960	1	3,707	104,792	0.02
germanrr	miplib	10,813	5,251	5,323	239	10,779	175,547	0.01
glass-sc	miplib	214	0	214	0	6,119	63,918	0.23
glass4	miplib	322	0	302	20	396	1,815	0.80
gmu-35-40	miplib	1,205	0	1,200	5	424	4,843	0.41
gmu-35-50	miplib	1,919	0	1,914	5	435	8,643	0.46
graph20-20-1rand	miplib	2,183	0	2,183	0	5,587	19,277	0.30
graphdraw-domain	miplib	254	20	180	54	865	2,600	0.70
h80x6320d	miplib	12,640	0	6,320	6,320	6,558	31,521	0.32
hypothyroid-k1	miplib	2,602	1	2,601	0	5,195	433,884	0.84
ic97_potential	miplib	728	73	450	205	1,046	3,138	0.11

Table A.1: Instance set characteristics (continued).

name	set	cols	int	bin	con	rows	nz	cg_ρ ($\times$100)
icir97_tension	miplib	2,494	573	262	1,659	1,203	22,333	0.19
irish-electricity	miplib	61,728	0	9,888	51,840	104,259	523,257	0.01
irp	miplib	20,315	0	20,315	0	39	98,254	14.05
istanbul-no-cutoff	miplib	5,282	0	30	5,252	20,346	71,477	6.33
k1mushroom	miplib	8,211	1	8,210	0	16,419	1,697,946	0.42
keller4cpart	miplib	9,606	0	9,606	0	327,198	663,660	0.30
keller4cpartpp	miplib	9,601	0	9,601	0	41,648	386,003	0.30
l152lav	miplib	1,989	0	1,989	0	97	9,922	3.24
lectsched-5-obj	miplib	21,805	416	21,389	0	38,884	239,608	0.00
leo1	miplib	6,731	0	6,730	1	593	131,218	0.08
leo2	miplib	11,100	0	11,099	1	593	219,959	0.08
long_early01	rostering	52,729	0	52,729	0	17,241	1,012,492	0.23
long_early02	rostering	52,803	0	52,803	0	17,241	1,012,566	0.22
long_hidden01	rostering	63,205	0	63,205	0	28,370	1,065,275	0.16
long_hidden02	rostering	63,205	0	63,205	0	28,370	1,065,275	0.16
long_late01	rostering	63,005	0	63,005	0	27,875	1,063,670	0.16
long_late02	rostering	63,005	0	63,005	0	27,875	1,063,670	0.16
long_late04	rostering	63,005	0	63,005	0	27,875	1,063,670	0.16
lotsize	miplib	2,985	0	1,195	1,790	1,920	6,565	0.08
mad	miplib	220	0	200	20	51	2,808	1.62
map10	miplib	164,547	0	146	164,401	328,818	549,920	0.34
map16715-04	miplib	164,547	0	146	164,401	328,818	549,920	0.34
markshare_4_0	miplib	34	0	30	4	4	123	1.69
markshare2	miplib	74	0	60	14	7	434	0.84
mas74	miplib	151	0	150	1	13	1,706	0.33
mas76	miplib	151	0	150	1	12	1,640	0.33
mc11	miplib	3,040	0	1,520	1,520	1,920	6,080	0.03
mcsched	miplib	1,747	0	1,745	2	2,107	8,088	0.09
medium_early02	rostering	30,309	0	30,309	0	8,668	622,471	0.42
medium_hidden01	rostering	37,415	0	37,415	0	16,070	635,725	0.27
medium_hidden02	rostering	37,415	0	37,415	0	16,070	635,725	0.27
medium_hidden05	rostering	37,415	0	37,415	0	16,070	635,725	0.27
medium_late01	rostering	34,850	0	34,850	0	14,062	623,360	0.31
medium_late02	rostering	34,814	0	34,814	0	14,062	623,352	0.31
medium_late03	rostering	29,486	0	29,486	0	8,872	603,434	0.43
mik-250-20-75-4	miplib	270	175	75	20	195	9,270	0.67
milo-v12-6-r2-40-1	miplib	2,688	0	840	1,848	5,628	14,604	0.10
momentum1	miplib	5,174	0	2,349	2,825	42,680	103,198	0.49
mushroom-best	miplib	8,468	118	8,237	113	8,580	188,735	0.01
mzzv11	miplib	10,240	251	9,989	0	9,499	134,603	0.13
mzzv42z	miplib	11,717	235	11,482	0	10,460	151,261	0.08
n2seq36q	miplib	22,480	0	22,480	0	2,565	183,292	1.02
n3div36	miplib	22,120	0	22,120	0	4,484	340,740	0.01
neos-1281048	miplib	739	0	739	0	522	8,808	1.07
neos-1354092	miplib	13,702	420	13,282	0	3,135	187,187	0.02
neos-1445765	miplib	20,617	0	2,150	18,467	2,147	40,230	0.02
neos-1456979	miplib	4,605	180	4,245	180	6,770	36,440	0.76
neos-1582420	miplib	10,100	100	10,000	0	10,180	24,814	0.01
neos-1595230	miplib	490	0	490	0	1,750	3,885	1.03
neos-1599274	miplib	4,500	0	4,500	0	1,237	46,800	0.58
neos-1620770	miplib	792	0	792	0	9,296	19,292	1.45
neos-1620807	miplib	231	0	231	0	1,340	2,860	2.49
neos-1622252	miplib	828	0	828	0	9,695	20,125	1.41
neos-2657525-crna	miplib	524	378	146	0	342	1,690	0.68

Table A.1: Instance set characteristics (continued).

name	set	cols	int	bin	con	rows	nz	cg_ρ ($\times 100$)
neos-2746589-doon	miplib	50,936	224	50,704	8	31,530	271,072	0.10
neos-2978193-inde	miplib	20,800	0	64	20,736	396	41,600	0.79
neos-2987310-joes	miplib	27,837	0	3,051	24,786	29,015	580,291	0.03
neos-3046615-murg	miplib	274	16	240	18	498	1,266	0.42
neos-3216931-puriri	miplib	3,555	0	3,268	287	5,989	91,691	0.50
neos-3381206-awhea	miplib	2,375	1,900	475	0	479	4,275	0.11
neos-3402294-bobin	miplib	2,904	0	2,616	288	591,076	2,034,888	0.14
neos-3555904-turama	miplib	37,461	0	37,461	0	146,493	793,605	0.73
neos-3627168-kasai	miplib	1,462	0	535	927	1,655	5,158	0.09
neos-3656078-kumeu	miplib	14,870	4,455	9,755	660	17,656	59,292	0.02
neos-3754480-nidda	miplib	253	0	50	203	402	1,488	1.01
neos-4300652-rahue	miplib	33,003	0	20,900	12,103	76,992	183,616	0.00
neos-4338804-snowy	miplib	1,344	42	1,260	42	1,701	6,342	0.04
neos-4387871-tavua	miplib	4,004	0	2,000	2,004	4,554	23,496	0.13
neos-4413714-turia	miplib	190,402	0	190,201	201	2,303	761,756	0.03
neos-4532248-waihi	miplib	86,842	0	86,841	1	167,322	525,339	0.16
neos-4647030-tutaki	miplib	12,600	0	7,000	5,600	8,382	3,953,388	0.02
neos-4722843-widden	miplib	77,723	20	73,349	4,354	113,555	311,529	0.00
neos-4738912-atrato	miplib	6,216	5,096	1,120	0	1,947	19,521	0.06
neos-4763324-toguru	miplib	53,593	0	53,592	1	106,954	266,805	0.11
neos-4954672-berkel	miplib	1,533	0	630	903	1,848	8,007	0.08
neos-5049753-cuanza	miplib	242,736	0	8,304	234,432	322,248	1,440,672	0.01
neos-5052403-cygnet	miplib	32,868	0	32,868	0	38,268	4,898,304	0.00
neos-5093327-huahum	miplib	40,640	0	64	40,576	51,840	784,768	0.79
neos-5104907-jarama	miplib	345,856	0	9,520	336,336	489,818	2,053,548	0.01
neos-5107597-kakapo	miplib	3,114	0	2,976	138	6,498	19,392	0.02
neos-5114902-kasavu	miplib	710,164	0	14,560	695,604	961,170	4,240,376	0.01
neos-5188808-nattai	miplib	14,544	0	288	14,256	29,452	133,686	0.55
neos-5195221-niemur	miplib	14,546	0	9,792	4,754	42,256	176,586	0.02
neos-565815	miplib	1,276	0	1,276	0	15,413	124,071	2.44
neos-611135	miplib	6,400	0	6,400	0	5,277	769,300	3.06
neos-631694	miplib	3,725	0	3,725	0	3,996	18,523	0.71
neos-631709	miplib	45,150	0	45,150	0	46,496	225,148	0.23
neos-631710	miplib	167,056	0	167,056	0	169,576	834,166	0.12
neos-631784	miplib	22,725	0	22,725	0	23,996	113,023	0.39
neos-662469	miplib	18,235	328	17,907	0	1,085	200,055	0.19
neos-785899	miplib	1,320	0	1,320	0	1,653	17,180	1.26
neos-787933	miplib	236,376	0	236,376	0	1,897	298,320	0.00
neos-791021	miplib	9,448	0	9,448	0	3,694	29,708	0.14
neos-799838	miplib	20,844	0	20,844	0	5,976	57,888	0.03
neos-808214	miplib	1,308	0	1,308	0	640	22,530	1.21
neos-825075	miplib	800	0	800	0	328	5,480	0.92
neos-848589	miplib	550,539	0	747	549,792	1,484	1,101,078	0.07
neos-860300	miplib	1,385	0	1,384	1	850	384,329	5.23
neos-873061	miplib	175,288	0	87,644	87,644	93,360	350,576	0.00
neos-905856	miplib	686	0	686	0	403	6,601	1.43
neos-911970	miplib	888	0	840	48	107	3,408	0.74
neos-912023	miplib	686	0	686	0	623	14,728	1.52
neos-931538	miplib	7,920	0	7,920	0	5,964	33,480	0.10
neos-934531	miplib	1,082	0	1,082	0	47,078	136,119	2.06
neos-948346	miplib	57,855	0	57,855	0	1,570	540,443	0.20
neos-950242	miplib	5,760	240	5,520	0	34,224	104,160	0.08
neos-957323	miplib	57,756	0	57,756	0	3,757	499,656	0.19
neos1	miplib	2,112	0	2,112	0	5,020	21,312	0.17

Table A.1: Instance set characteristics (continued).

name	set	cols	int	bin	con	rows	nz	cg_ρ (×100)
neos17	miplib	535	0	300	235	486	4,931	0.17
neos18	miplib	3,312	0	3,312	0	11,402	24,614	0.04
neos5	miplib	63	0	53	10	63	2,016	0.95
neos8	miplib	23,228	4	23,224	0	46,324	313,180	0.01
net12	miplib	14,115	0	1,603	12,512	14,021	80,384	0.08
netdiversion	miplib	129,180	0	129,180	0	119,589	615,282	0.00
nexp-150-20-8-5	miplib	20,115	0	17,880	2,235	4,620	42,465	0.01
ns1208400	miplib	2,883	0	2,880	3	4,289	81,746	0.61
ns1688347	miplib	2,685	0	2,685	0	4,191	66,908	3.23
ns1696083	miplib	7,982	0	7,982	0	11,063	384,129	2.47
ns1760995	miplib	17,956	0	17,822	134	615,388	1,854,012	0.38
ns1830653	miplib	1,629	0	1,458	171	2,932	100,933	0.95
ns894236	miplib	9,666	0	9,666	0	8,218	41,067	0.26
ns903616	miplib	21,582	0	21,582	0	18,052	91,641	0.21
nu25-pr12	miplib	5,868	36	5,832	0	2,313	17,712	0.01
nursesched-medium-hint03	rostering	34,248	78	34,170	0	14,062	622,800	0.32
nursesched-sprint02	rostering	10,250	20	10,230	0	3,522	204,000	1.20
nw04	miplib	87,482	0	87,482	0	36	636,666	22.12
opm2-z10-s4	miplib	6,250	0	6,250	0	160,633	371,240	0.21
p0033	miplib	33	0	33	0	16	98	2.52
p0201	miplib	201	0	201	0	133	1,923	1.18
p0282	miplib	282	0	282	0	241	1,966	0.44
p0548	miplib	548	0	548	0	176	1,711	0.17
P1	bmc	11,824	0	11,823	1	304,432	620,645	0.07
P2	bmc	11,656	0	11,655	1	288,525	588,663	0.07
p200x1188c	miplib	2,376	0	1,188	1,188	1,388	4,752	0.04
p2756	miplib	2,756	0	2,756	0	755	8,937	0.04
P3	bmc	8,842	0	8,841	1	227,468	463,735	0.10
P4	bmc	9,388	0	9,387	1	232,257	473,859	0.08
P5	bmc	8,317	0	8,316	1	213,918	436,110	0.10
P6	bmc	7,582	0	7,581	1	187,451	382,441	0.11
p6b	miplib	462	0	462	0	5,852	11,704	1.48
P7	bmc	23,668	0	23,667	1	610,120	1,243,865	0.04
P8	bmc	11,824	0	11,823	1	304,432	620,645	0.07
P9	bmc	47,356	0	47,355	1	1,221,496	2,490,305	0.02
pb-simp-nonunif	miplib	23,848	0	23,848	0	1,451,912	4,366,648	0.00
pdistuchoa	miplib	500	0	500	0	26,314	52,628	5.37
pg	miplib	2,700	0	100	2,600	125	5,200	0.50
pg5_34	miplib	2,600	0	100	2,500	225	7,700	0.50
physiciansched3-3	miplib	79,555	0	72,141	7,414	266,227	1,062,479	0.00
physiciansched6-2	miplib	111,827	0	109,346	2,481	168,336	480,259	0.00
piperout-08	miplib	10,399	130	10,245	24	14,589	44,959	0.33
piperout-27	miplib	11,659	121	11,514	24	18,442	54,662	0.26
pk1	miplib	86	0	55	31	45	915	0.92
proteindesign121hz512p9	miplib	159,145	91	159,054	0	301	629,449	0.38
proteindesign122trx11p8	miplib	127,326	78	127,248	0	254	503,427	0.45
qap10	miplib	4,150	0	4,150	0	1,820	18,200	0.23
radiationm18-12-05	miplib	40,623	11,247	14,688	14,688	40,935	96,149	0.01
radiationm40-10-02	miplib	172,013	47,213	62,400	62,400	173,603	406,825	0.00
rail01	miplib	117,527	0	117,527	0	46,843	392,086	0.00
rail02	miplib	270,869	0	270,869	0	95,791	756,228	0.00
rail507	miplib	63,019	0	63,009	10	509	468,878	0.00
ran14x18-disj-8	miplib	504	0	252	252	447	10,277	0.20
rd-rplusc-21	miplib	622	0	457	165	125,899	852,384	2.83

Table A.1: Instance set characteristics (continued).

name	set	cols	int	bin	con	rows	nz	cg_ρ (×100)
reblock115	miplib	1,150	0	1,150	0	4,735	13,724	0.22
reblock67	miplib	670	0	670	0	2,523	7,495	0.35
rmatr100-p10	miplib	7,359	0	100	7,259	7,260	21,877	0.50
rmatr200-p5	miplib	37,816	0	200	37,616	37,617	113,048	0.25
rocI-4-11	miplib	6,839	1,016	5,192	631	10,883	27,383	0.02
rocII-5-11	miplib	11,523	0	11,341	182	26,897	303,291	0.45
rococoB10-011000	miplib	4,456	136	4,320	0	1,667	16,517	0.02
rococoC10-001000	miplib	3,117	124	2,993	0	1,293	11,751	0.03
roi2alpha3n4	miplib	6,816	0	6,642	174	1,251	878,812	0.02
roi5alpha10n8	miplib	106,150	0	105,950	200	4,665	2,370,224	0.00
roll3000	miplib	1,166	492	246	428	2,295	29,386	0.52
s100	miplib	364,417	0	364,417	0	14,733	1,777,917	0.96
s250r10	miplib	273,142	0	273,139	3	10,962	1,318,607	0.14
satellites2-40	miplib	35,378	0	34,324	1,054	20,916	283,668	0.01
satellites2-60-fs	miplib	35,378	0	34,324	1,054	16,516	125,048	0.01
savsched1	miplib	328,575	0	252,731	75,844	295,989	1,770,507	0.00
sct2	miplib	5,885	0	2,872	3,013	2,151	23,643	0.03
seymour	miplib	1,372	0	1,372	0	4,944	33,549	0.04
seymour1	miplib	1,372	0	451	921	4,944	33,549	0.15
sing326	miplib	55,156	0	40,010	15,146	50,781	268,173	0.01
sing44	miplib	59,708	0	43,524	16,184	54,745	281,260	0.01
snp-02-004-104	miplib	228,350	167	167	228,016	126,512	463,941	0.30
sorrell3	miplib	1,024	0	1,024	0	169,162	338,324	8.12
sp150x300d	miplib	600	0	300	300	450	1,200	0.17
sp97ar	miplib	14,101	0	14,101	0	1,761	290,968	0.03
sp98ar	miplib	15,085	0	15,085	0	1,435	426,148	0.04
splice1k1	miplib	3,253	1	3,252	0	6,505	1,761,016	2.69
sprint_early01	rostering	10,460	0	10,460	0	3,522	204,210	1.15
sprint_early02	rostering	10,458	0	10,458	0	3,522	204,208	1.15
sprint_hidden01	rostering	10,421	0	10,421	0	3,814	202,591	1.14
sprint_hidden02	rostering	10,421	0	10,421	0	3,814	202,591	1.14
sprint_late01	rostering	11,863	0	11,863	0	5,032	208,583	0.89
sprint_late02	rostering	10,423	0	10,423	0	3,804	202,783	1.14
square41	miplib	62,234	37	62,197	0	40,160	13,566,426	11.03
square47	miplib	95,030	43	94,987	0	61,591	27,329,856	11.19
stdc6262p	timetabling	16,415	0	16,415	0	27,334	95,471	0.01
supportcase10	miplib	14,770	0	14,770	0	165,684	555,082	0.02
supportcase18	miplib	13,410	0	13,410	0	240	28,920	0.29
supportcase26	miplib	436	0	396	40	870	2,492	0.13
supportcase33	miplib	20,203	101	20,102	0	20,489	211,915	0.36
supportcase40	miplib	16,440	0	2,000	14,440	38,192	104,420	0.26
supportcase6	miplib	130,052	1	130,051	0	771	584,976	5.61
supportcase7	miplib	138,844	14	451	138,379	6,532	2,845,545	0.22
swath1	miplib	6,805	0	2,306	4,499	884	34,965	0.02
swath3	miplib	6,805	0	2,706	4,099	884	34,965	0.02
t1717	miplib	73,885	0	73,885	0	551	325,689	0.64
t1722	miplib	36,630	0	36,630	0	338	133,096	0.72
ta_BPWC_5_5_5	bpwc	945	0	945	0	4,664	10,979	1.00
ta_BPWC_5_7_1	bpwc	591	0	591	0	2,186	5,315	1.05
ta_BPWC_5_7_4	bpwc	591	0	591	0	2,290	5,523	1.05
ta_BPWC_6_9_8	bpwc	834	0	834	0	2,542	6,273	0.56
ta_BPWC_7_1_8	bpwc	28,038	0	28,038	0	205,356	465,793	0.21
tbfp-network	miplib	72,747	0	72,747	0	2,436	215,837	0.83
tELGN_BPWC_6_6_20	bpwc	1,645	0	1,645	0	4,980	12,771	0.84

Table A.1: Instance set characteristics (continued).

name	set	cols	int	bin	con	rows	nz	cg_ρ (×100)
tELGN_BPWC_6_8_9	bpwc	918	0	918	0	1,569	4,495	0.99
tELGN_BPWC_7_6_16	bpwc	9,142	0	9,142	0	108,815	234,919	0.38
thor50dday	miplib	106,261	0	53,131	53,130	53,360	212,060	0.00
timtab1	miplib	397	94	77	226	171	829	0.65
tMIMT_BPPC_6_3_4	bpwc	5,899	0	5,899	0	26,108	63,535	0.46
tMIMT_BPPC_8_7_5	bpwc	21,816	0	21,816	0	332,209	706,047	0.24
tr12-30	miplib	1,080	0	360	720	750	2,508	0.14
traininstance2	miplib	12,890	2,602	5,278	5,010	15,603	41,531	1.14
traininstance6	miplib	10,218	2,056	4,154	4,008	12,309	32,785	1.48
trd445c	timetabling	1,431	0	1,431	0	96,133	195,080	3.84
trdcrooms	timetabling	174,915	170,303	4,612	0	338,305	1,176,743	0.02
trdnc18	timetabling	10,930	828	10,102	0	3,850	48,103	0.09
trdta0010	timetabling	5,759	121	5,638	0	6,367	29,756	0.31
trdta449	timetabling	14,741	1,293	13,448	0	23,268	73,966	0.03
trdta8265	timetabling	9,151	89	9,062	0	19,484	126,610	0.11
trdta99	timetabling	9,613	175	9,438	0	26,881	157,258	0.12
trdtatl9220	timetabling	5,778	177	5,601	0	9,378	40,800	0.12
trento1	miplib	7,687	0	6,415	1,272	1,265	93,571	0.01
ua_BPWC_1_8_10	bpwc	1,548	0	1,548	0	9,257	21,131	0.55
ua_BPWC_1_9_2	bpwc	834	0	834	0	2,663	6,515	0.55
uccase12	miplib	62,529	0	9,072	53,457	121,161	419,447	0.01
uccase9	miplib	33,242	0	8,064	25,178	49,565	332,316	0.01
uct-subprob	miplib	2,256	0	379	1,877	1,973	10,147	0.15
uELGN_BPWC_3_2_18	bpwc	114,957	0	114,957	0	1,348,796	2,925,507	0.12
uELGN_BPWC_3_9_18	bpwc	3,638	0	3,638	0	9,337	23,951	0.55
uMIMT_BPPC_2_5_2	bpwc	15,185	0	15,185	0	192,758	414,887	0.33
uMIMT_BPPC_2_9_1	bpwc	1,388	0	1,388	0	2,696	7,169	0.88
uMIMT_BPPC_3_7_6	bpwc	20,725	0	20,725	0	260,763	560,977	0.28
unitcal_7	miplib	25,755	0	2,856	22,899	48,939	127,595	0.02
var-smallemery-m6j6	miplib	5,608	0	5,606	2	13,416	850,621	0.03
wachplan	miplib	3,361	1	3,360	0	1,553	89,361	0.35
wnq-n100-mw99-14	miplib	10,000	0	10,000	0	656,900	1,333,400	0.33

Table A.2 presents the detailed results of the experiments with the new version of the CBC solver. Configuration “cbc+cg” refers to the new version of CBC including all of our routines, while “cbc” represents the previous version of this solver. Configuration “-{clqstr}” represents the new version of CBC without performing our clique strengthening routine. Finally, configurations “-{bkclqext}” and “-{oddw}” contain the results of the execution of the new version of CBC without including our clique and odd-cycle cut separators, respectively.

Table A.2: Results of the execution of the COIN-OR Branch-and-Cut solver.

instance	group	cbc		cbc+cg		-{clqstr}		-{bkclqext}		-{oddw}	
		gap closed	time	gap closed	time	gap closed	time	gap closed	time	gap closed	time
30n20b8	miplib	83.690	10800.000	83.690	10800.000	83.690	10800.000	83.690	10800.000	83.690	10800.000
50v-10	miplib	63.811	10800.000	63.811	10800.000	63.811	10800.000	63.811	10800.000	63.811	10800.000
air03	miplib	100.000	1.120	100.000	2.680	100.000	2.540	100.000	2.740	100.000	2.580
air04	miplib	100.000	64.640	100.000	58.030	100.000	53.840	100.000	144.460	100.000	53.690
air05	miplib	100.000	33.710	100.000	34.780	100.000	32.590	100.000	23.920	100.000	34.070
app1-1	miplib	100.000	17.020	100.000	16.950	100.000	17.340	100.000	16.450	100.000	17.090
app1-2	miplib	100.000	3523.240	100.000	3631.780	100.000	3523.590	100.000	3541.500	100.000	3496.830
assign1-5-8	miplib	45.181	10800.000	45.181	10800.000	45.181	10800.000	45.181	10800.000	45.181	10800.000
atlanta-ip	miplib	1.244	10800.000	17.303	10800.000	0.936	10800.000	0.910	10800.000	0.959	10800.000
b1c1s1	miplib	60.288	10800.000	60.288	10800.000	60.288	10800.000	60.288	10800.000	60.288	10800.000
bab1	miplib	65.603	10800.000	65.603	10800.000	65.603	10800.000	65.603	10800.000	65.603	10800.000
bab2	miplib	81.275	10800.000	87.521	10800.000	81.275	10800.000	81.275	10800.000	87.521	10800.000
bab3	miplib	69.812	10800.000	93.784	10800.000	69.812	10800.000	69.812	10800.000	93.784	10800.000
bab5	miplib	69.653	10800.000	93.911	10800.000	69.653	10800.000	69.653	10800.000	93.911	10800.000
bab6	miplib	82.370	10800.000	86.724	10800.000	83.235	10800.000	81.771	10800.000	82.822	10800.000
beasleyC3	miplib	85.107	10800.000	85.107	10800.000	85.107	10800.000	85.107	10800.000	85.107	10800.000
binkar10_1	miplib	100.000	117.360	100.000	116.610	100.000	110.680	100.000	110.360	100.000	118.620
blp-ar98	miplib	87.114	10800.000	87.114	10800.000	87.114	10800.000	87.114	10800.000	87.114	10800.000
blp-ic98	miplib	76.272	10800.000	76.272	10800.000	76.272	10800.000	76.272	10800.000	76.272	10800.000
bnatt400	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
bppc4-08	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
br1	timetabling	75.016	10800.000	87.163	10800.000	87.135	10800.000	80.761	10800.000	87.469	10800.000
br2	timetabling	81.439	10800.000	82.298	10800.000	82.048	10800.000	81.008	10800.000	81.061	10800.000
br3	timetabling	91.361	10800.000	93.591	10800.000	94.140	10800.000	92.800	10800.000	93.939	10800.000
br5	timetabling	80.003	10800.000	85.582	10800.000	84.974	10800.000	82.273	10800.000	83.255	10800.000
brazil3	timetabling	86.364	10800.000	100.000	10800.000	95.455	10800.000	90.909	10800.000	95.455	10800.000
chromaticindex1024-7	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
chromaticindex512-7	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
cmflsp50-24-8-8	miplib	40.948	10800.000	40.948	10800.000	40.948	10800.000	40.948	10800.000	40.948	10800.000
CMS750_4	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
co-100	miplib	12.461	10800.000	46.156	10800.000	19.078	10800.000	19.078	10800.000	19.368	10800.000
cod105	miplib	37.239	10800.000	37.083	10800.000	37.191	10800.000	37.080	10800.000	37.207	10800.000
comp07-2idx	timetabling	100.000	10800.000	100.000	8501.040	100.000	8931.720	100.000	10800.000	100.000	2444.140
comp21-2idx	timetabling	57.191	10800.000	65.259	10800.000	63.554	10800.000	60.139	10800.000	64.049	10800.000
cost266-UUE	miplib	72.481	10800.000	72.585	10800.000	72.585	10800.000	72.481	10800.000	72.585	10800.000

Table A.2: Results for the execution of COIN-OR Branch-and-Cut solver (continued).

instance	group	cbc		cbc+cg		-{clqstr}		-{bkclqext}		-{oddw}	
		gap closed	time	gap closed	time	gap closed	time	gap closed	time	gap closed	time
csched007	miplib	46.207	10800.000	46.207	10800.000	46.207	10800.000	46.207	10800.000	46.207	10800.000
csched008	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
cvs16r128-89	miplib	28.868	10800.000	28.868	10800.000	28.868	10800.000	28.868	10800.000	28.868	10800.000
d_BPWC_2_4_10	binpacking	0.001	10800.000	100.000	80.890	100.000	1407.230	85.100	10800.000	100.000	80.310
da_BPWC_2_8_6	binpacking	2.521	10800.000	58.375	10800.000	58.339	10800.000	2.723	10800.000	58.238	10800.000
da_BPWC_2_8_9	binpacking	1.092	10800.000	56.723	10800.000	56.701	10800.000	0.458	10800.000	56.716	10800.000
dano3_3	miplib	100.000	350.490	100.000	358.520	100.000	380.820	100.000	351.720	100.000	351.050
dano3_5	miplib	100.000	1118.760	100.000	1017.510	100.000	1021.170	100.000	1028.640	100.000	1041.500
drayage-100-23	miplib	100.000	47.460	100.000	48.650	100.000	47.720	100.000	55.580	100.000	48.030
drayage-25-23	miplib	99.917	10800.000	99.917	10800.000	99.917	10800.000	99.917	10800.000	99.917	10800.000
ds	miplib	0.329	10800.000	1.321	10800.000	1.321	10800.000	1.600	10800.000	1.321	10800.000
dws008-01	miplib	40.472	10800.000	40.472	10800.000	40.472	10800.000	40.472	10800.000	40.472	10800.000
eil33-2	miplib	12.750	10800.000	100.000	37.650	12.750	10800.000	12.750	10800.000	100.000	134.610
eilA101-2	miplib	20.865	10800.000	100.000	10478.550	20.865	10800.000	20.865	10800.000	24.065	10800.000
eilA76	miplib	100.000	233.820	100.000	18.850	100.000	19.180	100.000	24.300	100.000	19.070
eilB101	miplib	100.000	2439.560	100.000	1967.930	100.000	1783.930	100.000	469.630	100.000	1739.890
eilB101.2	miplib	60.100	10800.000	70.316	10800.000	61.308	10800.000	61.308	10800.000	61.308	10800.000
eilB76	miplib	100.000	26.130	100.000	6.500	100.000	6.980	100.000	7.010	100.000	6.460
eilC76	miplib	100.000	689.520	100.000	94.920	100.000	83.350	100.000	86.600	100.000	26.990
eilC76.2	miplib	53.048	10800.000	100.000	4457.350	53.048	10800.000	53.048	10800.000	100.000	1442.080
eilD76	miplib	100.000	721.180	100.000	216.180	100.000	216.060	100.000	54.440	100.000	189.070
eilD76.2	miplib	23.828	10800.000	100.000	3306.520	23.828	10800.000	23.828	10800.000	51.078	10800.000
enlight_hard	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
exp-1-500-5-5	miplib	75.233	10800.000	75.233	10800.000	75.233	10800.000	75.233	10800.000	75.233	10800.000
fast0507	miplib	100.000	1486.370	100.000	1778.030	100.000	1670.900	100.000	1520.120	100.000	1609.940
fastxgemm-n2r6s0t2	miplib	2.387	10800.000	2.251	10800.000	2.451	10800.000	2.389	10800.000	2.198	10800.000
fiball	miplib	100.000	844.690	100.000	734.470	100.000	785.310	100.000	905.080	100.000	858.390
germanrr	miplib	27.109	10800.000	27.052	10800.000	27.097	10800.000	26.985	10800.000	27.048	10800.000
glass-sc	miplib	65.346	10800.000	65.346	10800.000	65.346	10800.000	65.346	10800.000	65.346	10800.000
glass4	miplib	50.000	10800.000	50.000	10800.000	50.000	10800.000	50.001	10800.000	50.001	10800.000
gmu-35-40	miplib	11.207	10800.000	11.207	10800.000	11.207	10800.000	11.207	10800.000	11.207	10800.000
gmu-35-50	miplib	0.282	10800.000	0.282	10800.000	0.282	10800.000	0.282	10800.000	0.282	10800.000
graph20-20-1rand	miplib	49.107	10800.000	49.107	10800.000	49.107	10800.000	49.107	10800.000	49.107	10800.000
graphdraw-domain	miplib	88.961	10800.000	91.635	10800.000	92.008	10800.000	89.980	10800.000	91.729	10800.000
h80x6320d	miplib	77.509	10800.000	77.509	10800.000	77.509	10800.000	77.509	10800.000	77.509	10800.000

Table A.2: Results for the execution of COIN-OR Branch-and-Cut solver (continued).

instance	group	cbc		cbc+cg		-{clqstr}		-{bkclqext}		-{oddw}	
		gap closed	time	gap closed	time	gap closed	time	gap closed	time	gap closed	time
hypothyroid-k1	miplib	100.000	1846.630	100.000	1940.640	100.000	1906.630	100.000	1916.250	100.000	1892.290
ic97_potential	miplib	65.441	10800.000	65.441	10800.000	65.441	10800.000	65.385	10800.000	65.441	10800.000
icir97_tension	miplib	70.681	10800.000	70.681	10800.000	70.681	10800.000	70.681	10800.000	70.681	10800.000
irish-electricity	miplib	84.609	10800.000	84.623	10800.000	84.598	10800.000	84.625	10800.000	84.619	10800.000
irp	miplib	100.000	9.370	100.000	16.490	100.000	15.640	100.000	6.850	100.000	16.770
istanbul-no-cutoff	miplib	100.000	2072.840	100.000	1722.170	100.000	1689.430	100.000	1728.130	100.000	1680.130
k1mushroom	miplib	2.192	10800.000	2.192	10800.000	2.192	10800.000	2.192	10800.000	2.192	10800.000
keller4cpart	miplib	28.164	10800.000	43.396	10800.000	41.055	10800.000	31.687	10800.000	43.396	10800.000
keller4cpartpp	miplib	0.106	10800.000	2.325	10800.000	2.325	10800.000	0.126	10800.000	2.325	10800.000
l152lav	miplib	100.000	1.420	100.000	1.350	100.000	1.390	100.000	1.310	100.000	1.990
lectsched-5-obj	miplib	62.500	10800.000	62.500	10800.000	62.500	10800.000	62.500	10800.000	62.500	10800.000
leo1	miplib	67.014	10800.000	67.014	10800.000	67.014	10800.000	67.014	10800.000	67.014	10800.000
leo2	miplib	53.840	10800.000	53.840	10800.000	53.840	10800.000	53.840	10800.000	53.840	10800.000
long_early01	rostering	100.000	866.430	100.000	677.400	100.000	534.640	100.000	758.150	100.000	704.640
long_early02	rostering	95.098	10800.000	100.000	2026.510	100.000	3849.930	100.000	1857.990	100.000	2361.700
long_hidden01	rostering	36.593	10800.000	93.934	10800.000	90.305	10800.000	84.834	10800.000	93.997	10800.000
long_hidden02	rostering	24.979	10800.000	93.684	10800.000	93.684	10800.000	83.158	10800.000	93.684	10800.000
long_late01	rostering	0.000	10800.000	100.000	8320.280	99.773	10800.000	100.000	8585.050	99.397	10800.000
long_late02	rostering	0.000	10800.000	99.998	10800.000	99.999	10800.000	96.468	10800.000	99.170	10800.000
long_late04	rostering	0.000	10800.000	94.546	10800.000	78.761	10800.000	78.761	10800.000	78.761	10800.000
lotsize	miplib	39.711	10800.000	39.711	10800.000	39.711	10800.000	39.711	10800.000	39.711	10800.000
mad	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
map10	miplib	100.000	3363.290	100.000	3053.760	100.000	3077.270	100.000	3153.970	100.000	3011.780
map16715-04	miplib	0.653	10800.000	0.653	10800.000	0.653	10800.000	0.653	10800.000	0.653	10800.000
markshare_4_0	miplib	100.000	13.660	100.000	13.720	100.000	13.630	100.000	14.050	100.000	13.480
markshare2	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
mas74	miplib	100.000	147.420	100.000	149.530	100.000	149.010	100.000	151.030	100.000	149.540
mas76	miplib	100.000	19.670	100.000	19.230	100.000	19.530	100.000	19.260	100.000	19.520
mc11	miplib	75.734	10800.000	75.734	10800.000	75.734	10800.000	75.734	10800.000	75.734	10800.000
mcsched	miplib	83.760	10800.000	100.000	9765.830	100.000	9691.970	85.265	10800.000	100.000	9631.910
medium_early02	rostering	60.340	10800.000	100.000	65.200	100.000	191.310	100.000	62.990	100.000	62.250
medium_hidden01	rostering	5.800	10800.000	59.355	10800.000	58.806	10800.000	53.841	10800.000	60.035	10800.000
medium_hidden02	rostering	0.000	10800.000	69.848	10800.000	68.420	10800.000	61.329	10800.000	69.644	10800.000
medium_hidden05	rostering	0.000	10800.000	51.731	10800.000	47.515	10800.000	24.194	10800.000	56.305	10800.000
medium_late01	rostering	31.792	10800.000	93.553	10800.000	92.404	10800.000	86.532	10800.000	94.123	10800.000

Table A.2: Results for the execution of COIN-OR Branch-and-Cut solver (continued).

instance	group	cbc		cbc+cg		-{clqstr}		-{bkclqext}		-{oddw}	
		gap closed	time	gap closed	time	gap closed	time	gap closed	time	gap closed	time
medium_late02	rostering	70.975	10800.000	100.000	936.110	100.000	1286.010	100.000	964.470	100.000	1467.750
medium_late03	rostering	56.539	10800.000	100.000	7387.810	100.000	2052.380	100.000	7363.820	100.000	4245.270
mik-250-20-75-4	miplib	100.000	21.110	100.000	21.710	100.000	21.090	100.000	21.330	100.000	21.270
milo-v12-6-r2-40-1	miplib	69.243	10800.000	69.243	10800.000	69.243	10800.000	69.243	10800.000	69.243	10800.000
momentum1	miplib	64.543	10800.000	67.618	10800.000	64.548	10800.000	64.645	10800.000	68.547	10800.000
mushroom-best	miplib	28.899	10800.000	28.899	10800.000	28.899	10800.000	28.899	10800.000	28.899	10800.000
mzzv11	miplib	100.000	455.460	100.000	527.250	100.000	230.280	100.000	246.270	100.000	208.330
mzzv42z	miplib	100.000	46.430	100.000	62.310	100.000	54.710	100.000	53.940	100.000	59.460
n2seq36q	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
n3div36	miplib	100.000	6638.880	100.000	5802.160	100.000	5749.800	100.000	5815.630	100.000	6049.010
neos-1281048	miplib	100.000	152.020	100.000	18.120	100.000	18.180	100.000	219.660	100.000	13.950
neos-1354092	miplib	100.000	10800.000	100.000	10800.000	100.000	10800.000	100.000	10800.000	100.000	10800.000
neos-1445765	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-1456979	miplib	65.818	10800.000	65.818	10800.000	65.818	10800.000	65.818	10800.000	65.818	10800.000
neos-1582420	miplib	100.000	22.250	100.000	22.350	100.000	22.350	100.000	22.510	100.000	22.370
neos-1595230	miplib	79.010	10800.000	79.108	10800.000	79.140	10800.000	78.915	10800.000	79.032	10800.000
neos-1599274	miplib	100.000	1.190	100.000	1.210	100.000	1.180	100.000	1.210	100.000	1.220
neos-1620770	miplib	75.000	10800.000	87.500	10800.000	75.000	10800.000	75.000	10800.000	75.000	10800.000
neos-1620807	miplib	100.000	1877.190	100.000	1130.360	100.000	1128.540	100.000	1152.590	100.000	1924.290
neos-1622252	miplib	75.000	10800.000	75.000	10800.000	75.000	10800.000	75.000	10800.000	75.000	10800.000
neos-2657525-crna	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-2746589-doon	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-2978193-inde	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-2987310-joes	miplib	100.000	6.140	100.000	6.460	100.000	6.610	100.000	6.330	100.000	6.570
neos-3046615-murg	miplib	43.311	10800.000	43.311	10800.000	43.311	10800.000	43.311	10800.000	43.311	10800.000
neos-3216931-puriri	miplib	5.702	10800.000	5.702	10800.000	5.702	10800.000	5.702	10800.000	5.702	10800.000
neos-3381206-awhea	miplib	100.000	23.010	100.000	23.870	100.000	23.520	100.000	23.490	100.000	23.200
neos-3402294-bobin	miplib	100.000	234.730	100.000	237.190	100.000	232.790	100.000	235.950	100.000	245.710
neos-3555904-turama	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-3627168-kasai	miplib	87.329	10800.000	87.329	10800.000	87.353	10800.000	87.329	10800.000	87.329	10800.000
neos-3656078-kumeu	miplib	23.154	10800.000	61.239	10800.000	40.207	10800.000	24.891	10800.000	30.206	10800.000
neos-3754480-nidda	miplib	81.593	10800.000	81.593	10800.000	81.593	10800.000	81.593	10800.000	81.593	10800.000
neos-4300652-rahue	miplib	1.178	10800.000	1.178	10800.000	1.178	10800.000	1.178	10800.000	1.178	10800.000
neos-4338804-snowy	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-4387871-tavua	miplib	46.608	10800.000	47.710	10800.000	46.608	10800.000	46.608	10800.000	47.710	10800.000

Table A.2: Results for the execution of COIN-OR Branch-and-Cut solver (continued).

instance	group	cbc		cbc+cg		-{clqstr}		-{bkclqext}		-{oddw}	
		gap closed	time	gap closed	time	gap closed	time	gap closed	time	gap closed	time
neos-4413714-turia	miplib	100.000	88.120	100.000	107.270	100.000	111.690	100.000	107.650	100.000	94.750
neos-4532248-waihi	miplib	0.963	10800.000	0.872	10800.000	0.872	10800.000	0.971	10800.000	0.908	10800.000
neos-4647030-tutaki	miplib	99.857	10800.000	99.857	10800.000	99.857	10800.000	99.857	10800.000	99.857	10800.000
neos-4722843-widden	miplib	48.290	10800.000	100.000	4374.910	49.388	10800.000	49.388	10800.000	100.000	6016.140
neos-4738912-atrato	miplib	100.000	922.510	100.000	929.950	100.000	929.890	100.000	917.130	100.000	932.350
neos-4763324-toguru	miplib	1.180	10800.000	1.180	10800.000	1.180	10800.000	1.180	10800.000	1.180	10800.000
neos-4954672-berkel	miplib	67.558	10800.000	67.558	10800.000	67.558	10800.000	67.558	10800.000	67.558	10800.000
neos-5049753-cuanza	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-5052403-cygnet	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-5093327-huahum	miplib	39.745	10800.000	39.745	10800.000	39.745	10800.000	39.745	10800.000	39.745	10800.000
neos-5104907-jarama	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-5107597-kakapo	miplib	100.000	47.990	100.000	49.010	100.000	47.020	100.000	49.210	100.000	47.960
neos-5114902-kasavu	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-5188808-nattai	miplib	100.000	9463.620	100.000	8909.360	100.000	9206.400	100.000	9446.750	100.000	9049.140
neos-5195221-niemur	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-565815	miplib	100.000	108.910	100.000	23.540	100.000	117.910	100.000	23.860	100.000	36.400
neos-611135	miplib	0.000	10800.000	6.883	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
neos-631694	miplib	20.000	10800.000	100.000	6.650	100.000	27.820	100.000	6.520	100.000	6.350
neos-631709	miplib	0.000	10800.000	100.000	215.440	64.706	10800.000	64.706	10800.000	64.706	10800.000
neos-631710	miplib	0.000	10800.000	100.000	1577.380	100.000	1317.550	0.000	10800.000	100.000	999.340
neos-631784	miplib	0.000	10800.000	73.333	10800.000	73.333	10800.000	73.333	10800.000	100.000	82.850
neos-662469	miplib	57.201	10800.000	57.176	10800.000	57.176	10800.000	57.201	10800.000	57.176	10800.000
neos-785899	miplib	100.000	46.510	100.000	46.950	100.000	30.780	100.000	277.730	100.000	188.100
neos-787933	miplib	90.400	10800.000	91.904	10800.000	90.615	10800.000	90.574	10800.000	91.978	10800.000
neos-791021	miplib	100.000	42.610	100.000	34.070	100.000	41.260	100.000	33.860	100.000	24.670
neos-799838	miplib	78.720	10800.000	100.000	20.330	100.000	21.080	83.333	10800.000	100.000	17.150
neos-808214	miplib	0.000	10800.000	40.000	10800.000	40.000	10800.000	0.000	10800.000	40.000	10800.000
neos-825075	miplib	100.000	21.150	100.000	23.410	100.000	23.370	100.000	17.350	100.000	12.870
neos-848589	miplib	97.467	10800.000	97.467	10800.000	97.467	10800.000	97.467	10800.000	97.467	10800.000
neos-860300	miplib	100.000	172.510	100.000	81.660	100.000	79.820	100.000	81.010	100.000	177.550
neos-873061	miplib	98.631	10800.000	98.631	10800.000	98.631	10800.000	98.631	10800.000	98.631	10800.000
neos-905856	miplib	100.000	746.780	100.000	1684.630	100.000	1731.350	100.000	6094.620	100.000	1719.810
neos-911970	miplib	99.820	10800.000	99.820	10800.000	99.820	10800.000	99.820	10800.000	99.820	10800.000
neos-912023	miplib	100.000	239.300	100.000	4.860	100.000	4.910	100.000	8.700	100.000	8.940
neos-931538	miplib	76.462	10800.000	82.430	10800.000	80.621	10800.000	76.462	10800.000	80.537	10800.000

Table A.2: Results for the execution of COIN-OR Branch-and-Cut solver (continued).

instance	group	cbc		cbc+cg		-{clqstr}		-{bkclqext}		-{oddw}	
		gap closed	time	gap closed	time	gap closed	time	gap closed	time	gap closed	time
neos-934531	miplib	100.000	1747.890	100.000	1455.610	100.000	1518.730	100.000	795.240	100.000	783.770
neos-948346	miplib	90.269	10800.000	90.995	10800.000	91.282	10800.000	91.191	10800.000	91.366	10800.000
neos-950242	miplib	100.000	1645.330	100.000	1593.560	100.000	1591.040	100.000	1664.470	100.000	1783.740
neos-957323	miplib	100.000	101.670	100.000	102.930	100.000	372.510	100.000	96.670	100.000	81.070
neos1	miplib	100.000	2.520	100.000	3.700	100.000	3.690	100.000	2.140	100.000	5.150
neos17	miplib	100.000	3463.280	100.000	3330.070	100.000	3417.520	100.000	3264.340	100.000	3334.560
neos18	miplib	100.000	167.000	100.000	755.180	100.000	745.130	100.000	391.070	100.000	1918.610
neos5	miplib	100.000	2551.660	100.000	2730.980	100.000	2739.850	100.000	2763.080	100.000	2594.350
neos8	miplib	100.000	45.620	100.000	44.470	100.000	45.710	100.000	44.870	100.000	45.670
net12	miplib	36.656	10800.000	36.656	10800.000	36.656	10800.000	36.656	10800.000	36.656	10800.000
netdiversion	miplib	46.429	10800.000	46.429	10800.000	46.429	10800.000	46.429	10800.000	46.429	10800.000
nexp-150-20-8-5	miplib	8.457	10800.000	8.457	10800.000	8.457	10800.000	8.457	10800.000	8.457	10800.000
ns1208400	miplib	100.000	10800.000	100.000	10800.000	100.000	10800.000	100.000	956.250	100.000	10800.000
ns1688347	miplib	15.794	10800.000	88.000	10800.000	80.000	10800.000	80.000	10800.000	88.000	10800.000
ns1696083	miplib	7.147	10800.000	32.558	10800.000	33.209	10800.000	28.424	10800.000	31.395	10800.000
ns1760995	miplib	0.887	10800.000	22.341	10800.000	11.499	10800.000	10.041	10800.000	11.267	10800.000
ns1830653	miplib	89.700	10800.000	100.000	6505.540	100.000	6585.590	93.469	10800.000	100.000	8955.170
ns894236	miplib	36.099	10800.000	36.099	10800.000	36.099	10800.000	36.099	10800.000	36.099	10800.000
ns903616	miplib	38.948	10800.000	52.940	10800.000	52.940	10800.000	52.940	10800.000	52.940	10800.000
nu25-pr12	miplib	100.000	94.960	100.000	85.250	100.000	94.980	100.000	93.370	100.000	92.530
nursesched-medium-hint03	rostering	40.647	10800.000	78.698	10800.000	72.589	10800.000	68.742	10800.000	77.674	10800.000
nursesched-sprint02	rostering	100.000	64.330	100.000	43.890	100.000	25.090	100.000	43.420	100.000	49.610
nw04	miplib	100.000	15.550	100.000	436.930	100.000	436.440	100.000	25.060	100.000	419.120
opm2-z10-s4	miplib	13.271	10800.000	23.925	10800.000	15.910	10800.000	13.271	10800.000	13.271	10800.000
p0033	miplib	100.000	0.040	100.000	0.050	100.000	0.060	100.000	0.050	100.000	0.040
p0201	miplib	100.000	2.060	100.000	1.960	100.000	1.610	100.000	1.950	100.000	2.030
p0282	miplib	100.000	0.730	100.000	0.870	100.000	0.730	100.000	0.740	100.000	0.880
p0548	miplib	100.000	0.110	100.000	0.120	100.000	0.080	100.000	0.090	100.000	0.100
P1	bandwidth	17.080	10800.000	59.915	10800.000	51.007	10800.000	24.369	10800.000	59.727	10800.000
P2	bandwidth	7.734	10800.000	46.340	10800.000	41.829	10800.000	6.859	10800.000	45.590	10800.000
p200x1188c	miplib	100.000	1576.980	100.000	1287.280	100.000	1576.350	100.000	1515.420	100.000	1573.960
p2756	miplib	100.000	1.160	100.000	0.770	100.000	0.790	100.000	1.220	100.000	0.750
P3	bandwidth	15.408	10800.000	24.795	10800.000	21.557	10800.000	18.529	10800.000	22.126	10800.000
P4	bandwidth	9.407	10800.000	22.716	10800.000	11.796	10800.000	9.973	10800.000	9.992	10800.000
P5	bandwidth	4.015	10800.000	8.563	10800.000	7.115	10800.000	3.975	10800.000	3.783	10800.000

Table A.2: Results for the execution of COIN-OR Branch-and-Cut solver (continued).

instance	group	cbc		cbc+cg		-{clqstr}		-{bkclqext}		-{oddw}	
		gap closed	time	gap closed	time	gap closed	time	gap closed	time	gap closed	time
P6	bandwidth	4.391	10800.000	100.000	9919.280	11.242	10800.000	4.571	10800.000	100.000	9811.850
p6b	miplib	97.454	10800.000	97.535	10800.000	97.531	10800.000	97.604	10800.000	97.525	10800.000
P7	bandwidth	6.660	10800.000	8.529	10800.000	8.746	10800.000	7.425	10800.000	8.008	10800.000
P8	bandwidth	17.080	10800.000	59.915	10800.000	51.007	10800.000	24.369	10800.000	59.727	10800.000
P9	bandwidth	1.683	10800.000	3.826	10800.000	3.719	10800.000	1.855	10800.000	3.774	10800.000
pb-simp-nonunif	miplib	33.333	10800.000	41.667	10800.000	38.889	10800.000	38.889	10800.000	41.667	10800.000
pdistuchoa	miplib	93.865	10800.000	97.097	10800.000	96.605	10800.000	94.183	10800.000	97.636	10800.000
pg	miplib	100.000	45.460	100.000	44.300	100.000	45.130	100.000	34.990	100.000	44.140
pg5_34	miplib	100.000	2166.540	100.000	1717.610	100.000	2160.710	100.000	2190.190	100.000	2132.100
physiciansched3-3	miplib	92.161	10800.000	93.952	10800.000	94.004	10800.000	92.828	10800.000	93.636	10800.000
physiciansched6-2	miplib	100.000	39.120	100.000	37.730	100.000	40.270	100.000	40.360	100.000	41.120
piperout-08	miplib	100.000	549.500	100.000	555.500	100.000	576.580	100.000	572.640	100.000	568.550
piperout-27	miplib	100.000	595.450	100.000	917.900	100.000	350.800	100.000	885.590	100.000	918.920
pk1	miplib	100.000	27.680	100.000	36.120	100.000	35.960	100.000	36.960	100.000	36.100
proteindesign121hz512p9	miplib	32.784	10800.000	32.784	10800.000	32.784	10800.000	32.784	10800.000	32.784	10800.000
proteindesign122trx11p8	miplib	35.943	10800.000	35.943	10800.000	35.943	10800.000	35.943	10800.000	35.943	10800.000
qap10	miplib	100.000	99.410	100.000	99.090	100.000	78.120	100.000	78.720	100.000	128.940
radiationm18-12-05	miplib	91.304	10800.000	91.304	10800.000	91.304	10800.000	91.304	10800.000	91.304	10800.000
radiationm40-10-02	miplib	95.305	10800.000	95.305	10800.000	95.305	10800.000	95.305	10800.000	95.305	10800.000
rail01	miplib	63.942	10800.000	64.029	10800.000	64.526	10800.000	63.611	10800.000	64.029	10800.000
rail02	miplib	0.003	10800.000	0.003	10800.000	0.003	10800.000	0.003	10800.000	0.003	10800.000
rail507	miplib	100.000	3899.370	100.000	4769.510	100.000	3809.920	100.000	4636.690	100.000	4816.900
ran14x18-disj-8	miplib	65.926	10800.000	65.926	10800.000	66.152	10800.000	66.037	10800.000	66.821	10800.000
rd-rplusc-21	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
reblock115	miplib	91.211	10800.000	91.211	10800.000	91.211	10800.000	91.211	10800.000	91.211	10800.000
reblock67	miplib	100.000	2803.170	100.000	5053.360	100.000	5119.770	100.000	5054.490	100.000	2770.690
rmatr100-p10	miplib	100.000	145.080	100.000	145.760	100.000	150.170	100.000	145.970	100.000	144.890
rmatr200-p5	miplib	0.000	10800.000	32.232	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
rocI-4-11	miplib	0.404	10800.000	0.404	10800.000	0.404	10800.000	0.404	10800.000	0.404	10800.000
rocII-5-11	miplib	2.051	10800.000	61.198	10800.000	2.750	10800.000	2.327	10800.000	3.241	10800.000
rococoB10-011000	miplib	74.943	10800.000	74.943	10800.000	74.943	10800.000	74.943	10800.000	74.943	10800.000
rococoC10-001000	miplib	100.000	981.130	100.000	976.550	100.000	984.450	100.000	974.730	100.000	988.440
roi2alpha3n4	miplib	100.000	930.880	100.000	910.160	100.000	897.790	100.000	852.360	100.000	945.380
roi5alpha10n8	miplib	46.571	10800.000	68.931	10800.000	68.931	10800.000	46.574	10800.000	68.774	10800.000
roll3000	miplib	100.000	1672.670	100.000	619.720	100.000	483.030	100.000	427.350	100.000	492.820

Table A.2: Results for the execution of COIN-OR Branch-and-Cut solver (continued).

instance	group	cbc		cbc+cg		-{clqstr}		-{bkclqext}		-{oddw}	
		gap closed	time	gap closed	time	gap closed	time	gap closed	time	gap closed	time
s100	miplib	79.871	10800.000	85.156	10800.000	79.871	10800.000	79.871	10800.000	79.871	10800.000
s250r10	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
satellites2-40	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
satellites2-60-fs	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
savsched1	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
sct2	miplib	69.619	10800.000	69.619	10800.000	69.619	10800.000	69.619	10800.000	69.619	10800.000
seymour	miplib	52.949	10800.000	52.949	10800.000	52.949	10800.000	52.949	10800.000	52.949	10800.000
seymour1	miplib	100.000	742.790	100.000	744.000	100.000	745.000	100.000	736.770	100.000	731.950
sing326	miplib	62.709	10800.000	62.709	10800.000	62.709	10800.000	62.709	10800.000	62.709	10800.000
sing44	miplib	69.554	10800.000	73.121	10800.000	70.861	10800.000	70.712	10800.000	71.738	10800.000
snp-02-004-104	miplib	98.971	10800.000	98.971	10800.000	98.971	10800.000	98.972	10800.000	98.971	10800.000
sorrell3	miplib	69.706	10800.000	99.250	10800.000	99.254	10800.000	99.176	10800.000	99.250	10800.000
sp150x300d	miplib	95.011	10800.000	94.932	10800.000	95.011	10800.000	95.011	10800.000	95.011	10800.000
sp97ar	miplib	49.496	10800.000	49.496	10800.000	49.496	10800.000	49.496	10800.000	49.496	10800.000
sp98ar	miplib	77.481	10800.000	77.481	10800.000	77.481	10800.000	77.481	10800.000	77.481	10800.000
splice1k1	miplib	0.068	10800.000	0.068	10800.000	0.068	10800.000	0.068	10800.000	0.068	10800.000
sprint_early01	rostering	100.000	561.370	100.000	32.480	100.000	21.830	100.000	33.460	100.000	37.050
sprint_early02	rostering	100.000	1251.020	100.000	34.470	100.000	37.650	100.000	34.350	100.000	37.460
sprint_hidden01	rostering	100.000	10381.210	100.000	200.570	100.000	113.170	100.000	201.600	100.000	102.600
sprint_hidden02	rostering	100.000	426.290	100.000	49.980	100.000	29.000	100.000	52.130	100.000	47.760
sprint_late01	rostering	100.000	8850.080	100.000	104.010	100.000	119.420	100.000	105.300	100.000	107.300
sprint_late02	rostering	100.000	1463.580	100.000	57.840	100.000	36.000	100.000	60.270	100.000	60.910
square41	miplib	0.006	10800.000	0.006	10800.000	0.006	10800.000	0.006	10800.000	0.006	10800.000
square47	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
stdc6262p	timetabling	95.220	10800.000	97.106	10800.000	97.123	10800.000	95.454	10800.000	94.931	10800.000
supportcase10	miplib	8.085	10800.000	8.257	10800.000	8.257	10800.000	8.257	10800.000	8.105	10800.000
supportcase18	miplib	0.041	10800.000	0.041	10800.000	0.041	10800.000	0.041	10800.000	0.041	10800.000
supportcase26	miplib	100.000	3475.990	100.000	3618.420	100.000	3031.720	100.000	3079.010	100.000	2926.680
supportcase33	miplib	54.217	10800.000	54.217	10800.000	54.217	10800.000	54.217	10800.000	54.217	10800.000
supportcase40	miplib	92.973	10800.000	96.424	10800.000	95.323	10800.000	93.518	10800.000	94.817	10800.000
supportcase6	miplib	36.371	10800.000	42.217	10800.000	36.371	10800.000	36.371	10800.000	40.568	10800.000
supportcase7	miplib	100.000	2168.190	100.000	2025.680	100.000	2045.170	100.000	2118.790	100.000	2169.270
swath1	miplib	100.000	106.720	100.000	97.310	100.000	95.920	100.000	94.550	100.000	96.460
swath3	miplib	100.000	441.370	100.000	458.820	100.000	445.010	100.000	442.180	100.000	436.700
t1717	miplib	3.703	10800.000	3.683	10800.000	3.683	10800.000	3.598	10800.000	3.683	10800.000

Table A.2: Results for the execution of COIN-OR Branch-and-Cut solver (continued).

instance	group	cbc		cbc+cg		-{clqstr}		-{bkclqext}		-{oddw}	
		gap closed	time	gap closed	time	gap closed	time	gap closed	time	gap closed	time
t1722	miplib	14.742	10800.000	15.367	10800.000	15.367	10800.000	14.918	10800.000	15.367	10800.000
ta_BPWC_5_5_5	binpacking	100.000	56.150	100.000	22.630	100.000	57.450	100.000	66.540	100.000	19.880
ta_BPWC_5_7_1	binpacking	100.000	2.100	100.000	1.760	100.000	2.040	100.000	1.750	100.000	1.800
ta_BPWC_5_7_4	binpacking	100.000	2.030	100.000	2.000	100.000	2.230	100.000	1.970	100.000	1.880
ta_BPWC_6_9_8	binpacking	100.000	4.200	100.000	2.760	100.000	4.170	100.000	2.760	100.000	2.750
ta_BPWC_7_1_8	binpacking	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
tbfp-network	miplib	100.000	343.620	100.000	351.870	100.000	349.800	100.000	348.930	100.000	336.240
tELGN_BPWC_6_6_20	binpacking	88.656	10800.000	100.000	693.330	90.834	10800.000	90.834	10800.000	90.834	10800.000
tELGN_BPWC_6_8_9	binpacking	100.000	6.600	100.000	7.070	100.000	5.720	100.000	6.190	100.000	3.690
tELGN_BPWC_7_6_16	binpacking	15.179	10800.000	100.000	52.670	100.000	406.490	88.629	10800.000	100.000	40.980
thor50dday	miplib	21.224	10800.000	26.627	10800.000	27.203	10800.000	21.224	10800.000	24.494	10800.000
timtab1	miplib	85.386	10800.000	85.669	10800.000	85.669	10800.000	85.615	10800.000	85.386	10800.000
tMIMT_BPPC_6_3_4	binpacking	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
tMIMT_BPPC_8_7_5	binpacking	6.790	10800.000	100.000	835.480	100.000	6273.320	91.288	10800.000	100.000	433.370
tr12-30	miplib	74.756	10800.000	74.756	10800.000	74.756	10800.000	74.756	10800.000	74.756	10800.000
traininstance2	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
traininstance6	miplib	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000
trd445c	timetabling	0.021	10800.000	100.000	46.560	100.000	40.440	100.000	38.930	100.000	44.630
trdcrooms	timetabling	0.000	10800.000	0.000	10800.000	0.000	10800.000	0.000	10800.000	9.935	10800.000
trdnc18	timetabling	19.901	10800.000	25.913	10800.000	25.925	10800.000	19.160	10800.000	29.795	10800.000
trdta0010	timetabling	100.000	1433.880	100.000	9.440	100.000	17.030	100.000	18.830	100.000	9.110
trdta449	timetabling	100.000	154.720	100.000	23.160	100.000	29.620	100.000	76.880	100.000	19.290
trdta8265	timetabling	84.111	10800.000	100.000	1499.050	100.000	236.910	92.106	10800.000	100.000	136.080
trdta99	timetabling	85.447	10800.000	89.177	10800.000	89.880	10800.000	86.326	10800.000	89.116	10800.000
trdtatl9220	timetabling	99.377	10800.000	99.660	10800.000	100.000	5017.140	100.000	5991.980	100.000	1763.340
trento1	miplib	51.125	10800.000	50.957	10800.000	51.113	10800.000	50.897	10800.000	50.955	10800.000
ua_BPWC_1_8_10	binpacking	100.000	229.610	100.000	13.800	100.000	57.380	100.000	15.480	100.000	12.780
ua_BPWC_1_9_2	binpacking	100.000	1.220	100.000	1.040	100.000	1.200	100.000	1.050	100.000	1.030
uccase12	miplib	95.773	10800.000	95.773	10800.000	95.773	10800.000	95.773	10800.000	95.773	10800.000
uccase9	miplib	6.579	10800.000	6.579	10800.000	6.579	10800.000	6.579	10800.000	6.579	10800.000
uct-subprob	miplib	57.447	10800.000	62.111	10800.000	62.817	10800.000	57.155	10800.000	61.364	10800.000
uELGN_BPWC_3_2_18	binpacking	0.021	10800.000	0.021	10800.000	0.021	10800.000	0.021	10800.000	0.021	10800.000
uELGN_BPWC_3_9_18	binpacking	100.000	48.750	100.000	7.570	100.000	12.790	100.000	8.070	100.000	7.530
uMIMT_BPPC_2_5_2	binpacking	4.659	10800.000	96.872	10800.000	96.872	10800.000	60.605	10800.000	96.872	10800.000
uMIMT_BPPC_2_9_1	binpacking	100.000	3.830	100.000	1.900	100.000	3.860	100.000	1.870	100.000	1.800

Table A.2: Results for the execution of COIN-OR Branch-and-Cut solver (continued).

instance	**group**	**cbc**		**cbc+cg**		**-{clqstr}**		**-{bkclqext}**		**-{oddw}**	
		gap closed	**time**	**gap closed**	**time**	**gap closed**	**time**	**gap closed**	**time**	**gap closed**	**time**
uMIMT_BPPC_3_7_6	binpacking	21.248	10800.000	89.981	10800.000	89.981	10800.000	49.832	10800.000	89.981	10800.000
unitcal_7	miplib	100.000	8495.040	100.000	7976.620	100.000	7973.080	100.000	8529.890	100.000	8388.870
var-smallemery-m6j6	miplib	66.979	10800.000	66.979	10800.000	66.979	10800.000	66.979	10800.000	66.979	10800.000
wachplan	miplib	100.000	5516.000	100.000	4344.020	100.000	8256.950	100.000	6382.320	100.000	6447.990
wnq-n100-mw99-14	miplib	12.716	10800.000	63.703	10800.000	63.703	10800.000	44.250	10800.000	63.703	10800.000