UNIVERSITÀ POLITECNICA DELLE MARCHE

Facoltà di Ingegneria

Corso di laurea in Ingegneria Informatica e dell'Automazione

Tesi di Laurea

Micol Zazzarini

Analisi di Strategie di Branching attraverso la Regolazione dei Parametri di Configurazione

Analysis of Branching Strategies through Configuration Parameter Tuning

Relatore:
Prof. **Fabrizio Marinelli**,
Ancona

Correlatore:
Dott. **Luciano Porretta**,
Antwerp

A.A. 2022-2023

In primis, un ringraziamento speciale al professor Fabrizio Marinelli che attraverso le sue lezioni ha suscitato in me l'interesse per una materia complessa quale la Ricerca Operativa, per avermi incoraggiato a prendere il volo quando mi chiedevo se fossi stata in grado, e per avermi fornito gli strumenti necessari per farlo con successo.

Un sentito grazie al Dott. Luciano Porretta, correlatore di tesi, per aver risposto quotidianamente alle mie mille domande in azienda, per il supporto costante, le dritte indispensabili e la sua complicità nella realizzazione di ogni capitolo della mia tesi.

Ringrazio l'azienda OMPartners per avermi dato la possibilità di svolgere il mio lavoro di tesi in un luogo interessante e dinamico, che mi ha permesso di mettermi in gioco e fare un'esperienza che sarà preziosa per il mio futuro.

Ringrazio i miei genitori per avermi sempre spinto a dare il meglio e a far nascere in me la sana competizione con la quale appoccio qualsiasi aspetto della mia quotidianeita'.

Un grazie di cuore a Ramon, che da guida presso l'universita' di Ingegneria di Las Palmas e' diventato poi compagno di vita, migliore amico, confidente, colui che mi da forza e mi incoraggia tutti i giorni.

La dedica e' per me stessa. Fiera di rimanere ogni giorno coerente alle mie priorita', grata per amarmi e scegliere sempre il meglio per la mia vita, orgogliosa per tutti i sacrifici con i quali raggiungo quotidianamente le mie piccole e grandi soddisfazioni.

"Se l'uomo non sapesse di Matematica non si eleverebbe di un sol palmo da terra."

Galileo Galilei

Sommario

Questo lavoro e' il risultato di una esperienza lavorativa nella sede belga di OMP, ad Anversa, Wommelgem, dal 1 Settembre 2023 al 5 Gennaio 2024. Il tirocinio e' stato a conclusione del percorso di laurea in Ingegneria Informatica e dell'Automazione.
OMP ha offerto l'opportunita' di mettere in pratica le nozioni teoriche apprese durante il corso di Ricerca Operativa, e l'esperienza e' stata accompagnata dalla scrittura della tesi in oggetto.
Infatti, molte delle sfide e dei problemi affrontati da OMP, in quanto leader nell'ambito della pianificazione della supply chain, interessano sistemi complessi che possono essere formulati in termini di Programmazione Matematica, e in particolare Programmazione Lineare Intera. In dettaglio, l'attenzione e' focalizzata sull'algoritmo di Branch and Bound, cercando di affrontare attraverso la configurazione dei parametri il problema di "Ping-Pong branching", per poter fornire a OMP utili informazioni e dati, aiutando a migliorare la loro applicazione.
Verra' fornita una presentazione completa dell'esperienza, partendo dall'azienda e gli aspetti piu' pratici del lavoro, fino ad arrivare allo specifico problema affrontato. La comprensione della parte sperimentale sara' accompagnata da esaustive definizioni matematiche e da un'introduzione del modello di Lot Sizing analizzato. Inoltre, prima di immergersi nei dati, il lettore sara' introdotto al problema di Configuration Parameter Tuning e a una guida per l'utilizzo di SCIP Optimizatio Suite, con il quale viene eseguito il tuning manuale nell'intera analisi. A conclusone, in seguito a una verifica dei risultati ottenuti, sara' anche possibile conoscere similarita' e differenze tra il tool in questione e l'applicazione sviluppata da OMP.

Abstract

This work is the result of a work experience at the belgian headquarters of OMP, in Antwerp, Wommelgem, from the 1st of September 2023 to the 5th January 2024. The internship was the conclusion of a first three-year degree in Computer and Automation Engineering.
OMP offered the opportunity to put into practice the theoretical notions learned during the Operations Research course , and the experience was accompanied by the writing of this thesis.
In fact, many of the challenges and problems faced by OMP, as a leader in supply chain planning, affect complex systems that can be formulated in terms of Mathematical Programming, and in particular Integer Linear programming. In detail, the attention is focused on the Branch and Bound Algorithm, trying to face through Configuration Parameter Tuning the problem of "Ping-Pong branching", in order to provide useful informations and data to OMP helping them to improve their application.
A complete presentation of the experience will be provided, starting from the company and the practical work aspects, up to the specific project addressed. The understanding of the experimental part will be accompanied by exhaustive mathematical definitions and an introduction of the analyzed Lot Sizing model. Furthermore, before delving into the data, the reader will be introduced to the problem of parameter configuration and to a guide on the use of SCIP Optimization Suite, with which manual tuning is performed and which will accompany the entire analysis. In conclusion, following a verification of the results obtained, it will also be possible to know similarities and differences between the tool in question and the application developed by OMP.

Contents

Index v

Glossary xiii

List of symbols xv

I Theoretical Setting 1

Introduction 3

1 OMPartners supply chain planning solution 5
1.1 OMPartners . . . 5
1.2 Field of action . . . 6
1.3 Solution . . . 7
1.4 Work teams . . . 7
1.5 Experience . . . 9

2 From the birth of linear programming to B&B 11
2.1 Linear Programming . . . 12
2.1.1 Duality . . . 13
2.2 Combinatorial Optimization and Integer Linear Programming . . . 13
2.3 Discrete optimization . . . 14
2.3.1 Relaxations and Bounds . . . 15
2.4 Branch-and-Bound . . . 16
2.4.1 Preprocessing . . . 19
2.4.2 Branching . . . 19
2.5 Branch-and-Cut . . . 19
2.5.1 Managing the LP relaxations . . . 22

3 The capacitated Lot Sizing Model 23
3.1 Mathematical models for Lot Sizing problems . . . 23
3.2 Logistics applications . . . 27

3.2.1 Lot sizing model under study 29

4 Parameter Configuration Problem **33**
4.1 Automatic Parameter Tuning 33
4.2 Manual Parameter Tuning with Scip Optimization Suite 35
4.2.1 Branch-and-Bound in SCIP 36
4.2.2 Using SCIP 37

5 Focusing on Primal Heuristics **41**
5.1 Locks 41
5.2 Feasibility Pump 43
5.3 Shift and Propagate 44
5.4 Zirounding 45
5.5 Intshifting 46

II Experiments **49**

6 Analysis **51**
6.0.1 Presolving 52
6.0.2 Primal Heuristics and Separation configuration search 60

7 Conclusions **83**

III Final Considerations **87**

8 Running on OMP application **89**

A Tables **95**
A.1 Supporting Data 95
A.2 Additional Data 100
A.2.1 PingPong 100
A.2.2 PingPong10 102
A.2.3 PingPong100 103
A.2.4 PingPong200 and PingPong1000 104

B Definitions **107**

Bibliography **111**

List of Tables

6.1 Branch and Bound tree data analysing presolving on model PingPong 52
6.2 Branch and Bound tree data analysing presolving impact on the resolution of the problem . . . 53
6.3 Branch and Bound tree data analysing presolving impact mantaining primal heuristics active. $x \in \{10, 100, 200\}$. . . . 54
6.4 Branch and Bound tree data analysing presolving impact mantaining separators active . . . 54
6.5 Branch and Bound tree data analysing presolving on model PingPong, mantaining primal heuristics active . . . 55
6.6 Branch and Bound tree data analysing presolving on model PingPong, mantaining separators active . . . 56
6.7 Presolving behavior switching setting for all presolvers . . . 57
6.8 Analysis of Branch and Bound tree switching the setting for all presolvers . . . 58
6.9 Branch and Bound tree data analysing heuristics setting all separators off . . . 60
6.10 Branch and Bound tree data analysing separators setting off all heuristics 62
6.11 Heuristics behavior switching setting for all heuristics . . . 64
6.12 Propagators increment setting heuristics emphasis fast . . . 65
6.13 Presolving increment setting heuristics emphasis aggressive . . . 66
6.14 Branch and Bound tree switching setting for all heuristics . . . 67
6.15 Heuristics behavior disabling locks on each model, compared with results obtained running the program in default setting . . . 69
6.16 Branch and Bound tree improvement disabling locks heuristic in PingPong1000 . . . 70
6.17 PingPong100.lp heuristics behavior with heuristics emphasis aggressive and heuristics locks freq -1. . . . 71
6.18 PingPong100.lp branch tree with heuristics emphasis fast and zirounding freq -1 . . . 71
6.19 Branch and Bound tree improvement given by locks setting heuristics emphasis off and heuristics emphasis fast, for each model. . . . 73
6.20 Branch and Bound tree data setting on heuristics one to one . . . 74

6.21 Branch and Bound comparison activating locks and feaspump 75
6.22 Separators behavior switching setting for heuristics 76
6.23 Ping-Pong.lp B&B tree changing the setting of separators when disabling heuristics 77
6.24 Branch and Bound tree improvement strengthening separators when setting heuristics emphasis fast 79
6.25 Branch and Bound tree worsening disabling aggregation separator . 80
6.26 PingPong1000.lp separators behavior switching the setting for all heuristics 80
6.27 PingPong1000.lp branch tree improvement strengthening separating 81
6.28 PingPong1000.lp branch and bound tree resulting from additional configurations 82

7.1 Results given by all possible configurations in terms of branch and bound tree size and time. 84
7.2 Comparison between primal heuristics used running PingPong2000 in default configuration by SCIP and the best one 85
7.3 Comparison between results pn B&B tree and solving time running PingPong2000 in default configuration by SCIP and the best one . . . 85

8.1 Statistics data given by default setting of OMP solver 91
8.2 Statistics data given by the use of aggregation 92
8.3 Statistics data given by feaspump 93

A.1 Presolvers used setting off heuristics and separators. $x \in \{10, 100, 200, 1000\}$ 95
A.2 Primal heuristics data analysing presolving impact mantaining primal heuristics active. $x \in \{10, 100, 200\}$ 96
A.3 Separators used evaluating presolving effect on separators, maintaining heuristics disabled 96
A.7 Separators behavior strengthening separators when setting heuristics emphasis fast, for models PingPong10,PingPong200 96
A.4 Primal heuristics evaluated setting off all separators. $x \in \{10, 100, 200\}$ 97
A.5 B&B tree changing the setting of separators when disabling heuristics 98
A.6 Branch and Bound tree changing the setting of separators when imposing heuristics emphasis fast 99
A.8 PingPong1000.lp separators behavior switching the setting for all separators 100
A.10 Ping-Pong.lp propagators behavior changing heuristics setting 100
A.9 Ping-Pong.lp solutions changing the setting for all heuristics 101
A.11 Ping-Pong.lp use of constraint handlers changing the setting of separators when disabling heuristics 102
A.12 Ping-Pong.lp use of propagators changing the setting of separators when disabling heuristics 102
A.13 PingPong10.lp use of constraint handlers switching the setting for all heuristics 102
A.14 PingPong10.lp use of propagators changing the setting of heuristics . 103
A.15 PingPong100.lp solutions changing the setting for all heuristics . . . 103
A.16 PingPong100.lp solutions disabling single heuristics 103

A.17 PingPong200.lp solutions changing the setting for all heuristics . . . 104
A.18 PingPong1000.lp solutions changing the setting for all heuristics . . . 105

List of Figures

1.1 OMP building in Antwerp . . . 5
1.2 OMP offices around the world . . . 6
1.3 Unison Planning . . . 8

3.1 DLSP as network flow problem . . . 25
3.2 Reverse rappresentation of a DLSP . . . 25
3.3 Representation Ping-Pong branching (I) . . . 30
3.4 Representation Ping-Pong branching (II) . . . 31
3.5 Representation Ping-Pong branching (III) . . . 31

4.1 Automatic Parameter Tuning algorithms . . . 34
4.2 Graphical representation of all parameters viewable via SCIP . . . 39
4.3 Graphic representation of SCIP cycle resolution of a problem . . . 40

5.1 locks fixing algorithm . . . 42
5.2 Feasibility Pump basic scheme . . . 44
5.3 Basic Shift and Propagate algorithm . . . 45
5.4 ZI Round . . . 46

6.1 Configurations given by combinations of settings of heuristics and separators. . . . 61
6.2 Configurations given by setting presolving and separating default, switching the setting for heuristics. . . . 63
6.3 Configurations given by setting presolving default, heuristics off, switching the setting for separators. . . . 77
6.4 Configurations given by setting presolving default, heuristics fast, switching the setting for separators. . . . 78
6.5 Configurations given by setting presolving default, heuristics default, switching the setting for separators. . . . 81
6.6 Additional Configurations evaluated for PingPong1000. . . . 82

8.1 PingPong model on OMP Optimizer . . . 90
8.2 PingPong general statistics by OMP Optimizer . . . 91

8.3 PingPong1000 statistics given by OMP solver on branch and Bound tree 92
8.4 PingPong statistics using aggregation 92
8.5 PingPong2000 results found by FeasPump 93

Part I

Theoretical Setting

Introduction

Object of study

Material requirements planning (MRP) is a production planning, scheduling, and inventory control system used to manage manufacturing processes. An MRP system is useful to ensure raw materials are available for production and products are available for delivery to customers, to maintain the lowest possible material and product levels in store, to plan manufacturing activities and delivery schedules and purchasing activities.
Many of these problems that affect complex systems, such as a *supply chain* and MRP, are decision problems that can be formulated in terms of *Mathematical Programming*, and in particular *integer linear programming*. Most of these problems are computationally difficult and requires the application of sophisticated computation.
In particular, MRP systems based on mathematical programming models use the B&B, an algorithm design paradigm for discrete and combinatorial optimization problems and mathematical optimization.
A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the complete set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.
The problem is that the algorithm is very effective in most cases, but it can be very inefficient in some specific ones, because its search space expands very rapidly as the domain sizes of the problem variables grow.
For example, this happens whenever the demand for multiple products exceeds the capacity of a shared machine and the lot size of one or several products is not a multiple of the machine's capacity.
In a standard branching scheme, integer variables linked to the lot size can be branched up and down consecutively because they exchange some left-over capacity that is a fraction of a lot size. This "ping pong" behavior of the branching could lead to performance degradation of the whole algorithm. Instead, a more intuitive and

fast approach is to branch all these variables down in one go because there is no way to make an extra lot size. The idea of this strategy is to cluster the values of a variable's domain into sets. Branch and bound can then branch on these sets of values rather than on individual values in order to construct a collection of sets on which branching will still allow effective bounding, reducing in this way branching factor and the size of the explored search space.

Objectives and contents

After analyzing the literature on the B&B algorithm, a series of manual tests have been performed through a particular tool, on different instances of a Lot Sizing problem, comparing the results with those obtained through the OMP device. Being a commercial product, the implementation of the latter application is unknown. However, the goal was to understand what the best parameter configuration was to solve many batch sizing problems using SCIP Optimization Suite, in order to support the OMP application to work as efficiently as possible.
In detail, it was noted that in the context of a particular Lot Sizing problem, and in correspondence with certain instances with a certain number of variables, SCIP presents a different resolution behavior compared to that of the OMP application, still reaching optimal results, if not better than those of this last one.
The algorithms used are very sophisticated and depend on many parameters. So, through a manual tuning of the most important parameters, it was thus possible to arrive at the best parameter configuration to solve the problem via SCIP. Given this result, useful information was extrapolated to improve the application of OMP.

CHAPTER 1

OMPartners supply chain planning solution

Let's start with a presentation of the company where the intership was carried out. You can find here a description of what the company does, what the challenges and goals of this great company are and how they carry it out. I am grateful for the opportunity I have been given.

Figure 1.1: OMP building in Antwerp

1.1 OMPartners

OMP is a software and consulting company delivering advanced solutions with its innovative planning software. It was founded in 1985 and it has strong roots in mathematical optimization. Over the years it was subject of a subsequent evolution towards *supply chain* solutions and now it is an internationally recognized player, one of the leaders in *supply chain planning*.
A supply chain describes a network of organizations, resources, activities and technologies involved in the creation and sale of a product and includes everything: from the delivery of basic materials from the supplier to the manufacturer to delivery to

the end user.
With a workforce of more than 1,000 people in offices around the world, Belgium, China, France, Germany, India, the Netherlands, Spain, Ukraine and the US, five core teams and six supporting teams keep things running smoothly (see Fig.1.2), they supervise, optimize and control the entire development process to guarantee top quality to their customers, including many leading global companies, such as ArcelorMittal, BASF, Dow, L'Oréal, Michelin, Procter & Gamble, Shaw, Shell, Smurfit Kappa, and Yoplait.
The product offered by OMP, Unison Planning, is a large generic application capable of adapting to the diversity of its customers, a layered solution offering a generic planning functionality with templates providing all the necessary to cover the specific industries, as supply chain planning challenges differ across them.

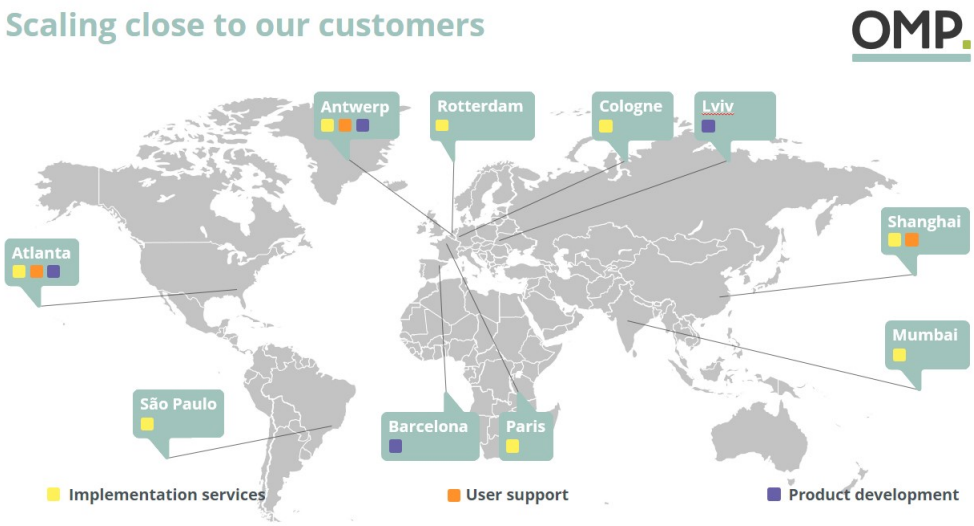


Figure 1.2: OMP offices around the world

1.2 Field of action

OMP work for different types of industries such as chemicals, consumer goods, life sciences, metals, paper, plastic film and packaging industries.
Orchestrating the chemicals supply chain to boost profitability means minimize changeovers, taking into account fixed-wheel, flexible wheel policies, throttling decisions, co-product fractions, and the performance of catalyst installations. Moreover, it also consists in the optimization of consumption through a mix of make-to-order and make-to-stock production strategies, quality control processes, networks of subcontractors and distribution channels, in order to maximize asset initialization, keep inventory levels within reasonable limits, and reduce waste and costs.
Tough challenges in the consumer goods industry are fierce competition, tight margins, shorter and shorter product life cycles, and difficult management of global operations. For them OMP offers sensing instruments, smart solvers and scenario planning tools to manage phase-ins and phase-outs, optimize asset utilization and continuously rebalance DC network. At all times, supply and demand must be in

tune across the entire consumer goods supply chain, and production must happen on a just-in-time basis as much as possible.
The complexities of metal supply chain require to maximize asset utilization, minimize scrap, keep inventory levels low, and improve delivery performance. OMP gives the solution to design and manage all operations in perfect unison. Key processes such as forecasting, sales and operations planning, campaign planning, order promising, and detailed order planning an scheduling are perfectly harmonized, leading to continuous improvement. Bills of materials and complex routing plans are generated dynamically.
In order to embrace the global digital era, OMP adjust manufactoring network, keep campaigns tuned to demand, balance the mix of make-to order and make-to stock operations, and optimize cutting and trimming plans. Paper and plastic film markets are increasingly diverse and complex. Paper mills and plastic film plants have to be flexible and have to constantly adapt their cutting plans as well as their master and campaign plans. The packaging industry is challenged by diminishing margins and increasing traceability requirements. OMP allows to manage all complexities with its smart forecasting and S&OP tools, planning, cutting, multistage trimming, scheduling and retrimming tools, with a unique corrugator optimization functionality.
In addition to the industries featured above, they develope solutions for aerospace, animal feed, cement, floor covering, glass, mining, rubber, starch, textiles, tires, wood, and more.

1.3 Solution

Supply chain planning is complex business. A decision maker have to face many challenges all at the same time.
Unison Planning responds to the needs of create sustainable value for the long run, control the risks while keeping all stakehlders in sync and garantee good performances. It's built upon one model that captures supply chain in all its dimensions, supported by Applied AI, data science and embedded intelligence, for capital-intensive industries. These companies in fact share one common challenge. They all need to combine optimal planning of material flows and optimal use of bottleneck capacities.
Instead of bringing together a disparate series of individual apps into one overarching solution, the one-model approach of UP treats the supply chain as a whole, from the ground up. It embraces the supply chain as a mix of intertwined dimensions, to be solved in their dependencies. (see Fig.1.3).
In this way users can now move back and forth easily between the strategic, operational and execution levels. That's helpful during analysis and for decision-making.

1.4 Work teams

OMP's work force include teams working for customer solutions, product design and software development. In this way they can assist their customers during the entire process, from the initial business case to the detailed configuration, they

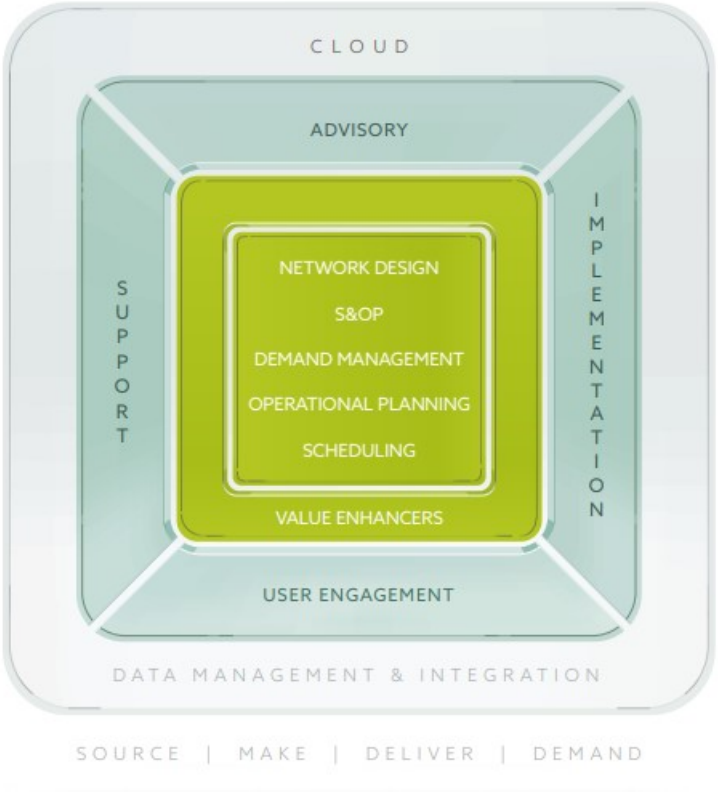


Figure 1.3: Unison Planning

provide assistance before, during and after implementation, they provide training and documentation and helping. Through the product design team they can identify opportunities for product improvement, they work for marketing and they can give support for the use. Software development team work with codes, software, algorithms through solvers, AI, machine learning, strong logic, smart data structures, and solid architecture, focusing on the performance of the application.

Even if focused on my own project, during my experience I worked in strict contact with the Product Design team with supply chain specialists, mathematical modelers, and data scientists. They work to provide a smart supply chain planning solution and steer the development process from drawing board to product launch. They help users and consultants to fully understand and use the OMP Solution, creating training materials, manuals and documentation for end users and consultants. The advisory and implementation teams count on them for expert advice throughout the entire project life cycle. In addition, they provide third line support for customer questions after the go-live.
As the architects of the OMP Solution, they play a key role in product marketing. They create demos and promotional materials to guide customers through the software's functionalites. They host webinars on new product features and offer domain expertise during presales.

1.5 Experience

Like all new arrivals, during my first weeks at the company I underwent a training period in order to be able to face the next work.
In particular, during my first two week, after visiting the company, I completed and passed the final tests of the following courses:

- Information Security Policy
- Introduction to OMP
- Introduction to office management
- Introduction to ICT

After that, getting more into the subject, I started my training with the tool that I would later use throughout my experience, SCIP Optimization Suite. After having studied a simple presentation focused on the Configuration Parameter Tuning and having consulted the official page and the documentation, I passed immediately to practice starting with simple models with a few variables, and then gradually increasing the complexity.
At the beginning of October I was ready to start working on the project and my colleagues introduced me specifically to the problem I would have to face. In detail, through their presentation and the bibliographic material provided, I studied the structure of the general Lot Sizing Problem and I was provided with the model that I would analyze.
Having the model instances available, I began my analysis. At the end of each phase of work I had to prepare a presentation of the results obtained, and the next work to be carried out was planned.
In the meantime, I participated with interest to the Product Development meetings that were held in the company.

CHAPTER 2

From the birth of linear programming to B&B

Entering into the subject, here is given a quick explanation of the mathematical prerequisites necessary to tackle the project, acquired during the Operations Research course, and result of further bibliographical research. Starting from the basics of mathematical programming, we then move on to linear and integer linear programming. Therefore, in this context, the fundamental definitions of discrete and combinatorial optimization, up to the definition of the Branch and Bound algorithm, which will accompany the entire drafting of the thesis and experimentation, are provided in detail.

Mathematical programming is the use of mathematical models to assist in taking decisions. In particular it makes use of optimization models with the aim to obtain the best solution of the problem associated with the mathematical model. When the mathematical representation uses linear functions exclusively, we have a linear-programming model, and it's especially this one of the best developed and most used branches of management science. It concerns for example the optimum allocation of limited resources among competing activities, under a set of constraints imposed by the nature of the problem being studied. They can be financial, technological, marketing, organizational constraints and many others types.
Linear programming and the first problems connected to it date back to 1945, when George Stigler formulated "the diet problem", in order to find the best diet model, best satisfying nutritional requirements with the minimum expense. Starting from this study, it was realized that this model can be considered a particular case of a more general class of objective function and linear constraint problems and that there is not yet a direct method for solving this type of problem. At the same time Dantzig, studying military problems, realized that many organizational problems can be formulated in the form of systems of linear inequalities, until in 1947 he formalized the concept of linear programming and also developed the first calculation method resolution, nowadays known as the "Simplex method", the only method until the introduction of the Interior Point method to solve larger problems by improving timing.
It is soon realized that the fractional values of some variables are not acceptable, in fact in many real-world situations it is often impossible to represent certain aspects of a problem using only continuous variables. it is sometimes necessary to represent

discrete quantities through variables that are forced to take only integer values, so that a new type of problem is therefore created which combines continuous variables with variables of a discrete nature: Mixed Integer Linear Programming (MILP). Since the most used technique to solve this type of problem (the so-called Branch&Bound method) consists in solving a large number of LP problems until the optimal solution of the MILP problem is identified, it becomes increasingly important to make the algorithms even more efficient and performing in such a way as to be able to solve even this class of problems in a reasonable time.

2.1 Linear Programming

A linear programming (LP) problem therefore consists of an objective function to be minimized (or maximized) subject to constraints (which may be in the form of equation or inequality), both of a linear type.

$\max z = \mathbf{c}^{\mathrm{T}}\mathbf{x}$	$\mathbf{c}^{\mathrm{T}}\mathbf{x}$ is the objective function
$\mathbf{A}\mathbf{x} \leq \mathbf{b}$	$\{X = \mathbf{x} \in \mathbb{R}^n$ s.t. $\mathbf{A}\mathbf{x} \leq \mathbf{b}\}$ is the feasible region
$\mathbf{x} \geq \mathbf{0}$	

Unknowns

- $\mathbf{x} \in \mathbb{R}^n$ Decision variables vector. Every $\mathbf{x} \in \mathbf{X}$ is a feasible solution.
- $z \in \mathbb{R}$ Value assumed by the objective function in correspondence with a solution $\mathbf{x} \in \mathbf{X}$.

Parameters

- $\mathbf{c} \in \mathbb{R}^n$ vector of coefficients of the objective function.
- $\mathbf{b} \in \mathbb{R}^n$ vector of the known terms of the constraints
- $\mathbf{A} \in \mathbb{R}^{m \times n}$ matrix of constraint coefficients

Redundant constraint: The constraint $\mathbf{a}^{\mathrm{T}}\mathbf{x} \leq \mathbf{b}$ is redundant with respect to the system of constraints $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ if every solution of $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ is also a solution of $\mathbf{a}^{\mathrm{T}}\mathbf{x} \leq \mathbf{b}$.

PL problem solution: A PL problem (in maximal form) can:

1. be feasible with one or more finite optimal solutions.The solution $\mathbf{x} \in X$ is optimal if $\forall \mathbf{y} \in X\, \mathbf{c}^{\mathrm{T}}\mathbf{x} \geq \mathbf{c}^{\mathrm{T}}\mathbf{y}$.
2. be empty or inadmissible ($X = \emptyset$)
3. be unlimited above, this happens when $\forall \delta \in R\, \exists \mathbf{x} \in X : \mathbf{c}^{\mathrm{T}}\mathbf{x} > \delta$.

Solving a PL problem means determining whether it is unbounded or inadmissible, i.e. producing a finite optimal solution.

Polyhedron: A polyhedron is the intersection of a number finite m of affine halfspaces

of R^n.
Polytope: A polytope is a bounded polyhedron.

Any system of linear equations/inequalities defines a polyhedron.The feasible region X of a PL problem is a polyhedron denoted by P(**A,b**);

Vertex: a point v of a polyhedron P is called vertex of P if $\exists$ a vector $\mathbf{c}$ such that $\mathbf{c}^T\mathbf{v} < \mathbf{c}^T\mathbf{x} \forall \mathbf{x} \in P \neq \mathbf{v}$.

Fundamental Theorem of PL: If a PL problem admits a finite optimum then exists one optimal solution which is a vertex of P.
If the problem is posed in standard form P: $max\{\mathbf{c}^T\mathbf{x}$ s.t. $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}, \mathbf{x} \in \mathbb{R}^n\}$, any feasible solution of P is also one solution of the system of linear equations $\mathbf{Ax} = \mathbf{b}$.

2.1.1 Duality

A linear programming problem is called a primitive problem in its original formulation, but always has a dual problem. The solution of the primitive problem allows you to easily obtain that of the dual problem.

$$\begin{aligned} \text{P:} \quad & z* = \max \mathbf{c}^T\mathbf{x} \\ & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

$$\begin{aligned} \text{D:} \quad & w* = \min \mathbf{y}^T\mathbf{b} \\ & \mathbf{A}^T\mathbf{y} \geq \mathbf{c} \\ & \mathbf{y} \geq \mathbf{0} \end{aligned}$$

2.2 Combinatorial Optimization and Integer Linear Programming

Combinatorial Optimization is a branch of Operations Research which, in mathematically modeling and solving complex problems of a discrete nature, combines combinatorial calculus techniques with the theory of algorithms and the theoretical and methodological results of linear programming.
It is one of the most important research areas in the field of optimization. In fact, it has multiple practical applications in many fields that have all a common characteristic: that of want to achieve an objective in compliance with constraints that regulate the use of resources available only in limited and finite quantities.
A discrete optimization problem can always be posed in form of linear mathematical model with integer variables, and a linear programming problem in which

all variables are constrained to assume only integer values is called linear integer programming. If the integrality clause concerns only some variables, it is called mixed-integer linear programming (MIP). In the last decade the use of mixed-integer programming models has increased dramatically. Today is possible to solve problems with thousands of integer variables and obtain good approximate solutions, thanks to developments in modeling, algorithms, software and hardware.

The program:

$$\begin{aligned} &\max \mathbf{cx} \\ &\mathbf{Ax} \leq \mathbf{b} \\ &\mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \\ &x_j \, integral, j = 1, ...p, \end{aligned}$$

is called **Mixed Integer Program (MIP)**. The input data are the matrices $\mathbf{c}(\mathbf{1xn})$,$\mathbf{A}(\mathbf{mxn})$, $\mathbf{b}(\mathbf{mx1})$, $\mathbf{l}(\mathbf{1xm})$, $\mathbf{u}(\mathbf{1xn})$, and the n vectors $\mathbf{x}$ to be determined. We assume $1 \leq p \leq n$ otherwise the problem is a **linear program (LP)**. If $p = n$, the problem is a **pure integer program (PIP)**. A PIP in which the variables have to be equal to 0 or 1 is called a **Binary Integer Program (BIP)** and a MIP in which all integer variables have to be equal to 0 or 1 is called a **Mixed Binary Integer Program (MBIP)**. Binary integer variables occurr very frequently in MIP models of real problems.

In these cases, linear programming cannot be used directly, we therefore use exact algorithms based on LP:

- Cutting planes algorithm (polyhedral approach) : generation of a sequence of hyperplanes that separate the optimal solutions provided by the simplex from the optimal solution of the problem of PLI.
- LP-based Branch-and-Bound (implicit enumeration) : recursive decomposition of the solution space and exclusion a priors of non-useful solutions.

They are general algorithms capable of solving any problem of discrete optimization expressed in terms of Integer Linear Programming.

2.3 Discrete optimization

$$\mathrm{P} = \max\{f(\mathbf{x}) \text{ s.t. } \mathbf{x} \in X\}$$

Solving a discrete optimization problem means determining an $\mathbf{x} \in X$ that maximizes (or minimizes) the function f.

Due to the high computational complexity, the universal algorithm turns out to be impractical. In fact, it is generally of exponential complexity because constructs, evaluates and verifies all possible solutions of the space of search, which in a combinatorial optimization problem consists of 2^n subsets of U (if $|U| = n$).
However an exponential algorithm does not always perform a total enumeration, it only happens in the worst case. The objective therefore becomes to avoid as much as possible the worse case, making the enumeration on average efficient.

2.3.1 Relaxations and Bounds

Relaxation: Let $P : max\{c(\mathbf{x}) : \mathbf{x} \in X \subseteq \mathbb{R}^n\}$ be an optimization problem. The problem $R : max\{c(\mathbf{x}) : \mathbf{x} \in Y \subseteq \mathbb{R}^n\}$ is a relaxation of P if $X \subseteq Y$, that is, if the feasible region of P is contained in that of R.
if $\mathbf{y}*$ is optimal for R and $\mathbf{x}*$ is optimal for P then $c(\mathbf{y}*) \geq c(\mathbf{x}*)$. The optimal solution of the relaxation provides an upper bound on $c(\mathbf{x}*)$.
A relaxation is obtained, for example, by eliminating any set of constraints from the model. In particular, by eliminating the integrality constraints, **continuous relaxation** is obtained.

Continuous relaxation: Let P: $z^* = max\{\mathbf{c^T x} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \in \mathbb{Z^n}\}$ be a PLI problem and $P_r : z_r^* = max\{\mathbf{c^T x} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \in \mathbb{R}^n\}$ its continuous relaxation.
It always applies $z_r^* \geq z^*$ that is, the optimal value of the continuous relaxation is always a higher limitation to the optimal value of the problem. Furthermore, if $\mathbf{x_r^*}$ is integer then $\mathbf{x_r^*}$ is the optimal solution of P. Similarly, in the case of a minimum problem we always have $z_r^* \leq z^*$ that is, the optimal value of the continuous relaxation is always a lower bound than the optimal value of the problem.

Lower and upper bounds: Let $z*$ be the optimal value of an optimization problem.

- Each $z_l \in \mathbb{R} : z_l \leq z*$ is called **lower bound** to the optimal value of the problem;
- Each $z_u \in \mathbb{R} : z_u \geq z*$ is called **upper bound** to the optimal value of the problem.

In a minimum (maximum) problem every feasible solution provides an upper (lower) bound, but that's not true each infeasible solution provides a lower bound (upper).

We call **primal bound** a bound provided by a feasible solution and **dual bound** a lower bound in a minimization problem or an upper bound in a maximum problem.
The quality (the distance from the optimal value) of the primal bounds and dual bounds is a critical factor for the viability of exact algorithms.

2.4 Branch-and-Bound

Branch-and-Bound algorithm is based on the recursive decomposition of the solution space and exclusion of non-useful solutions. In particular, foundamental elements of a Branch-and-Bound algorithm are:

1. separation into subproblems (partition)
2. relaxation (upper bounding)
3. fathoming of subproblems (lower bounding)
4. selection of subproblems (branching)

The basic structure of branch-and-bound is an enumeration tree. The root node of the tree corresponds to the original problem. Through the algorithm the tree grows by a process called branching that creates two or more child nodes of the parent node. In particular, the problems corrisponding to the child nodes are formed by adding constraints to the parent node's problem. The new constraint is obtained by bounding on a single integer variable. In this way each feasible solution to the parent node problem is feasible to at least one of the child node problems.
The most popular choice of relaxation is LP relaxation. The LP relaxation of a MIP is obtained by dropping the integrality restrictions. The main use of the LP relaxation in solving a MIP is that the optimal value of the LP relaxation provides an upper bound on the optimal value of the corrisponding MIP, and in many cases it gives a good approximation to the solution.
In addiction, if an optimal solution to the LP relaxation is found that satisfies the optimality restrictions, then that solution is also optimal to the MIP. If the LP relaxation is infeasible, then the MIP is also infeasible.
This is important because the most important feature in the success of Branch-and-Bound codes is the "distance" from the LP optimum to the MILP optimum.
There are also others types of relaxations, such as the Lagrangian relaxation, even if the LP relaxation still remains the main used.

Lagrangian relaxation: For MILP's of the form:

$$\begin{aligned} &\max \mathbf{cx} \\ &\mathbf{Ax} \leq \mathbf{b} \\ &\mathbf{Dx} \leq \mathbf{d} \\ &\mathbf{x} \geq \mathbf{0},\ \mathbf{x_j integer}, \mathbf{j} \in \mathbf{I} \end{aligned}$$

a Lagrangian relaxation is:

$$\begin{aligned} &\max \mathbf{cx} - \mathbf{\Lambda}(\mathbf{Ax} - \mathbf{b}) \\ &\mathbf{Dx} \leq \mathbf{d} \\ &\mathbf{x} \geq \mathbf{0},\ x_j integer, j \in I \end{aligned}$$

where $\mathbf{\Lambda}$ is a non-negative vector.

Since mostly of time consumed by a branch and bound algorithm is spent on solving the ralaxation, the speed improvements that have taken place in the last decade are extremely important, and are essential for the new Branch-and-Bound algorithms called Branch-and-Cut and Branch-and-Price.

Basic Algorithm

1. **Initialization:** $l = P, \mathbf{x} = \emptyset, \mathbf{z_l} = -\infty$

2. **Stop criterion:** if $l = \emptyset$ then x is the optimal solution. STOP.

3. **Subproblem selection:** choose a P problem i from list l and remove it from the list.

4. **Evaluation:** solve P_r^i, LP relaxation of P^i.
   if P_r^i is infeasible, then P^i is infeasible; go to 1.
   Let $\mathbf{x_r}^\mathbf{i}$ be an optimal solution of P_r^i.

5. **Bounding:** Bounding: if $z_r^i = z_u^i \leq z_l$ then go to 1.
   if $\mathbf{x_r}^\mathbf{i}$ is integer, then $z_l = z_r^i; \mathbf{x} = \mathbf{x_r}^\mathbf{i};$ go to 1.

6. **Branching:** Split P^i in the subproblems $P_1^i, ..., P_k^i$ such that $P^i = \cup_j P_j^i$ and add the problems to l; go to 1.

For LP based Branch-and-Bound algorithms the partitioning consists in separating a problem into subproblems introducing a set of contradictory constraints on one of the variables required to be integer.
In the normal course of a branch-and-bound algorithm, an unevaluated node is chosen, the LP relaxation is solved, and a fractional variable is chosen to branch on, so it's necessary to choice the active node to evaluate and the fractional value to branch on. These choices are important to keep the tree size small.
Suppose the LP relaxation has been solved at node k and the solution doesn't satisfy x_j integer, $j \in I$.
A variable $x_r, r \in I$ is chosen which has fractional value. $y_{r0} = y_{r0} + f_{r0}, f_{r0} \geq 0. S_k$ is partitioned into:

$$S_k \bigcap \{\mathbf{x} | x_r \leq |y_{r0}|\}$$
$$S_k \bigcap \{\mathbf{x} | x_r \geq |y_{r0}| + 1\}$$

For this we can consider:

- **Priorities:** a priority structure is an ordering in importance of the set of variables. To establish priorities the user may know that certains variables are critical and others are of secondary importance, while in absence of prior information, priorities can be set by ordering the variables by cost, using the down and up preudo costs of x_i, C_i^D and C_i^U, partitioning on that variable that maximize:

$$\min\{C_i^D f_{i0}, C_i^U(1 - f_{i0})\},\ i \in I$$

  Pseudo costs permit to estimate the change in objective function value caused by forcing a variable which is currently fractional to be integer. They can be estimated by the user or by the original costs of the variables, or performing small computation. Assume that the node k has been partitioned based on $x_i,\ i \in I$, and note that at nodes $k+1$ and $k+2$, x_i will be integer valued:

$$C_i^D = \frac{\bar{z_k} - \bar{z_{k+1}}}{f_{i0}}$$
$$C_i^U = \frac{\bar{z_k} - \bar{z_{k+2}}}{1 - f_{i0}}$$

- **Quasi-Integer variables:** it's possible to specify a tolerance t such that variable x_i is considered integer if $max\{f_{i0}, (1 - f_{i0})\} < t$. Such variables can be considered less important then those which are more seriously fractional.

Fathoming of node k occurs when $\bar{z_k} \leq z$ where z is the value of the best known solution of the MILP. Fathoming will be accelerated is z is high, close to $z*$. There are different ways to attempt to get a good lower bound on $z*$. Sometimes a good feasible solution is known a priori.
In the normal course of algorithm good feasible solution are discovered at various nodes, and the nodes that seem to have probability of containing feasible solution would be good candidates for early investigation.
Another way to find good feasible solution is the use of heuristics for any particular instance of the combinatorial problem. In particular, they can be incorporated at node zero in order to obtain an initial good lower bound, but also later in the tree.

The efficiency of the algorithm depends on many factors, from the selection criterion of the subproblem, from the choice of the branching variable, from the quality of the problem formulation, the type of relaxation adopted, the heuristics adopted. In particular, to speed up the execution, techniques such as the Preprocessing and various Branching strategies are adopted.

2.4.1 Preprocessing

Preprocessing consists on applying logic to reformulate the problem in a more convenient way, reducing the size by fixing variables and eliminating constraints, and detecting sometimes infeasibility.
The simplest test is the one based on the bound, through which redundant constraints can be eliminated or a bound on a variable can be tightened by recognizing that a constraint becomes infeasible if the variable is set at that bound. In this way spending more time initially it's possible to reduce the possibility of long solution times.
Furthermore we can increase the power of these simple logical tests with probing, that means setting temporarly a 0-1 variable at 0 or 1 and then redoing the logical testing. If the logical testing show that the problem has become infeasible, then the variable on which we probe can be fixed to the other bound. If the logical testing show that a constraint has become redundant, then it can be tightened by the coefficient of reduction.
So, preprocessing can identify infeasibility, redundant constraints, improve bounds,fix variables, generate new valid inequalities.

2.4.2 Branching

At any stage of the enumeration process, there are living nodes to be chosen for the next step of the algorithm. Node choice rules are motivated by the desire to find good feasible solutions early in the search, to limit the growth of the tree. In fact, the objective is always to solve the problem in a short time and this is function of the tree size and of the computation done at each node of the tree.
It's difficult to balance the advantages and disadvantages of selecting nodes near the top or bottom of the tree.
In general, the number of active nodes may explode if the active node is always chosen high up in the tree. On the other hand, if the node is always chosen from down low in the tree, the number of active nodes stay small but it may take a long time to find a good feasible solution.
A reasonable compromise is a mix of these two strategies, starting by investigating on one of the successors of the current node in order to find a good feasible solution, and then to keep the tree small examining the node k for witch $\bar{z}_k$ is greater over all live nodes.

2.5 Branch-and-Cut

The idea of Branch-and-Cut is the most significant advance in computational integer programming since basic Branch-and-Bound. It's a solution techniques to solve integer linear programs with a really high number of constraints.

A **valid inequality** for a MIP is an inequality that is satisfied by all feasible solutions.

A **cut** is a valid inequality that is not part of the current formulation and it's not satisfied by all feasible points to the LP-relaxation. A cut that is not satisfied by the

given optimal solution to the LP- relaxation is called **violated cut**.

If we have a violated cut , we can add it to the LP-relaxation and tighten it, modifying the current formulation in such a way that the LP-feasible region becomes smaller but the MIP feasible solution doesn't change. Then we can resolve the MIP and repeat this process continuing to find violated cuts. If none are found we branch.

A significant question in branch and cut is what classes of valid inequalities to generate and when. For generic mixed-integer programs, for example, cut generation is most important in the root node.
So, depending on the structure of the instance, different classes of valid inequalities may be effective. Sometimes, this can be predicted ahead of time (knapsack inequalities), while in other cases, we have to use past history as a predictor of effectiveness. Anyway, predicting what cuts will be effective is difficult in general. We can use the Degree of violation or other measures such as the Bound improvement or the Euclidean distance from point to be cut off but it is also possible to generate cuts using a different measure than that which is used to add them from the local pool. This might be done because generation by a criteria other than degree of violation is difficult.

We have two ways of obtaining cutting-planes for linear constraints on integer variables: taking linear combinations of the constraints, and using modular arithmetic, sufficient to obtain all the cutting planes when an integer program has a bounded feasible region.

For $h \geq 0$, let $IP(h)$ be a linear program consisting of a reasonable size subset of the constraints of $LP(\infty)$. Solve $LP(h)$, which yields an optimal solution $\bar{x}^h$. If this solution is feasible for $IP(\infty)$, it is an optimal solution, else assume we have a black box algorithm that gives us at least one constraint of $LP(\infty)$ violated by $\bar{x}^h$, if one exists, or tells us that all constraints are satisfied. If at least one violated constraint is returned, $LP(h+1)$ is obtained from $LP(h)$ by adding these constraints to those of $LP(h)$.
Note that for every $h \geq 0$, if $z_{LP(h)}$ is the optimal value of $LP(h)$, we have $z_{LP(h)} \leq z_{LP(h+1)} \leq z_{LP(\infty)} \leq z_{IP(\infty)}$.

The black box algorithm is called the **separation algorithm**.
So , given a solution to the LP-relaxation of a MIP that doesn't satisfy all the integrality constraints, the separation problem is trying to find a cut.
Separation routines are frequently based on fast heuristics. There are three types of valid inequalities that can be used to archieve integrality:

1. **Type 1: no structure:** based only on variables being integral or binary, they can always be used to separate a fractional point.

2. **Type 2: relaxed structure:** derived from relaxation of the problem.

3. **Type 3: Problem specific structure:** derived from the full problem structure.

Branch-and-Cut may fail for the following reasons:

1. We do not have a good algorithm to perform the cutting plane phase;

2. The number of iterations of the cutting plane phase is too high;

3. The linear program becomes unsolvable because of its size;

4. The tree generated by the branching procedure becomes too large and termination seems unlikely within a reasonable amount of time.

Cut management refers to strategies in Branch-and-Cut algorithms to ensure effective and efficient use of cuts in an LP-based Branch-and-bound algorithm.
They decide when to generate cuts, which of the generated cuts to add to the active linear program, and when to delete previously generated cuts from the active linear program, in order to try to decrease the time spent on generating cuts without reducing the effectiveness of the branch-and-cut algorithm.
This can be done by limiting the number of times cuts are generated during the evaluation of a node or by not generating cuts at every node of the search tree. Standard methods for generating cuts are:

- Gomory, GMI, MIR, and other tableau-based disjunctive cuts.

- Cuts from the node packing relaxation (*clique,odd hole*).

- Knapsack cuts (*cover cuts*).

- Single node flow cuts (*flow cover*).

- Simple cuts from pre-processing (*probing*).

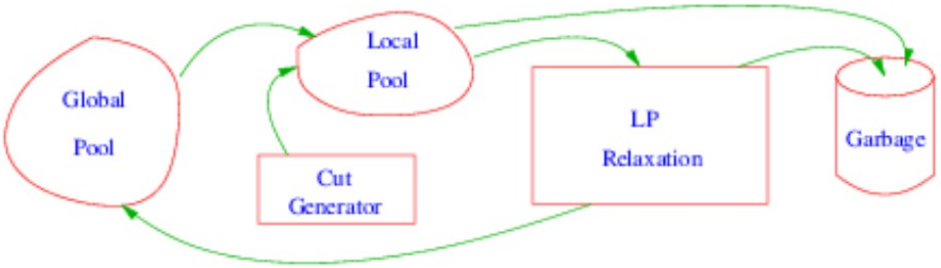
The problem is to choose among these methods which to apply in each node, and it depends on the level of effort we want to put into cut generation.

2.5.1 Managing the LP relaxations

Sometimes the number of inequalities generated can be huge, so that it's important to keep the size of the LP relaxations small in order to not sacrifice efficiency. This is done in two ways:

- Limiting the number of cuts that are added each iteration.
- Systematically deleting cuts that have become ineffective.

An ineffective cut is one whose dual value is near zero,or whose slack variable is basic or positive.
In practice, newly generated cuts enter a buffer (the local cut pool), but only a limited number of what are predicted to be the most effective cuts from the local are added in each iteration. Cuts that prove effective locally may eventually sent to a global pool for future use in processing other subproblems.



CHAPTER 3

The capacitated Lot Sizing Model

Since the addressed problem is an example of a Lot Sizing Model, an introduction to this family of problems is necessary. A general presentation of the Lot Sizing problem model is therefore given, in its direct or inverse form, highlighting how this can be adapted to an enormous variety of real problems. The model adapted to the original is then given, the instances of which were analyzed in the experimental part at OMP.

Starting from the developments of the Economic Order Quantity model and of the formulation of the Dynamic Lot Sizing Problem, inventory models have been widely employed to solve a wide range of theoretical and real-world problems. In fact, modifying the original models adapting them to the situations, it's possible to describe complex supply chains and logistics systems.
The history of inventory problems dates back to the Economic Order Quantity (EOQ) model presented by Harris in 1913, and firstly used in practice by Wilson in 1934. It presents a single item whose demand is continuous with an infinite plannng horizon, and its scope is to find the optimal quantity to be ordered, balancing setup and holding costs.
In presence of multiple items and capacity restrictions the model becomes NP-hard, and the Dynamic Lot Sizing Problem, first proposed by Wagner and Whitin in 1958 is an extension of the previous. In this case there are a deterministic and dynamic demand and finite time horizon but with the same objective of EOQ.
From this point onwards, different variants were born by adding or varying simple constraints, introducing new conditions, limitations, resources and objectives, inspired by specific real life applications and focusing on industrial production planning problems.

3.1 Mathematical models for Lot Sizing problems

We represent the time through N buckets denoted with $t \in 1...N$, so we have:

- d_t: demand forecast

- p_t: unit production or purchasing cost
- h_t: unit inventory cost
- f_t: fixed setup or ordering cost
- C_t: maximum feasible lot size capacity

Variables:

- s_t: stock at the end of period t
- x_t: quantity to be produced or ordered during period t
- y_t: binary variable equal to 1 if units of the product are manufactured or ordered in period t

DLSP model

$$\min z = \sum_{t=1}^{N} (p_t x_t + h_t s_t + f_t y_t) \tag{3.1.1}$$

$$s_t = s_{t-1} + x_t - d_t \qquad t = 1, ..., N \tag{3.1.2}$$

$$s_t = 0 \qquad t = 0 \, and \, t = N \tag{3.1.3}$$

$$x_t \leq C_t y_t \qquad t = 1, ..., N \tag{3.1.4}$$

$$s_t \geq 0;\ x_t \geq 0;\ y_t \in 0, 1 \qquad t = 1, ..., N \tag{3.1.5}$$

The objective function (3.1.1) aims to minimize production, inventory and setup costs. Among the constraints, the equilibrium constraint (3.1.2), aims to balance inventory levels, whereas the condition (3.1.3) imposes inventory levels at the beginning and the end of the planning horizon equal to 0. It is possible to have a positive production between 0 and C_t in period t only if the setup variable is equal to 1 (3.1.4), while the constraint (3.1.5) impose the non-negativity and binary restrictions of the variables. Zangwill in 1969 provided an interpretation of the problem as a fixed charge network problem, represented in *Figure 3.1*. The flow from the node 0 to another node, represented by an arc (0,t), is the production x_t in period t, while the flow from t to $t+1$ represented by arc $(t, t+1)$ reproduces the stock lend s_t at the end of the period t. The aim is to define the production inflows x_t able to satisfy the outflows d_t with the minimum costs, considering also the holdover flows from the previous periods.

It is possible also to use a reverse representation (see *Figure 3.2*), reversing flows x_t and d_t, so in this case the outflows (x_t) have to be determined in order to absorb

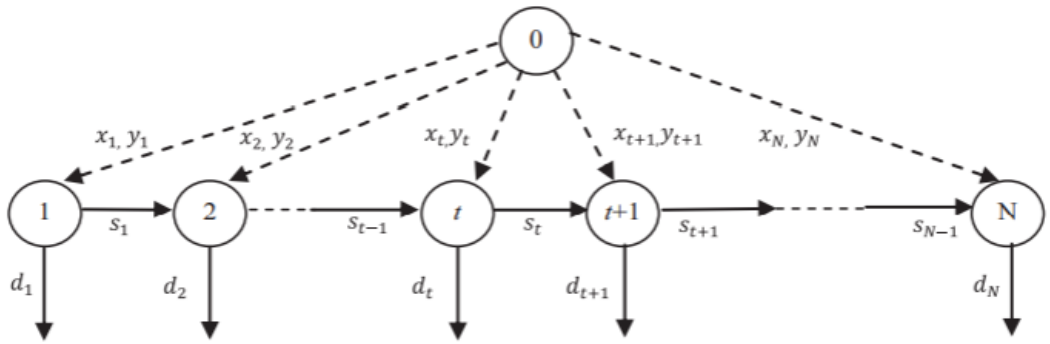


Figure 3.1: DLSP as network flow problem

the sum of the demand inflows (d_t) and of holdover flows from the previous period (s_{t-1}). The constraints have to be written reversing the signs of the variables x_t and parameters d_t:

$$s_t = s_{t-1} - x_t + d_t t = 1, ..., N$$

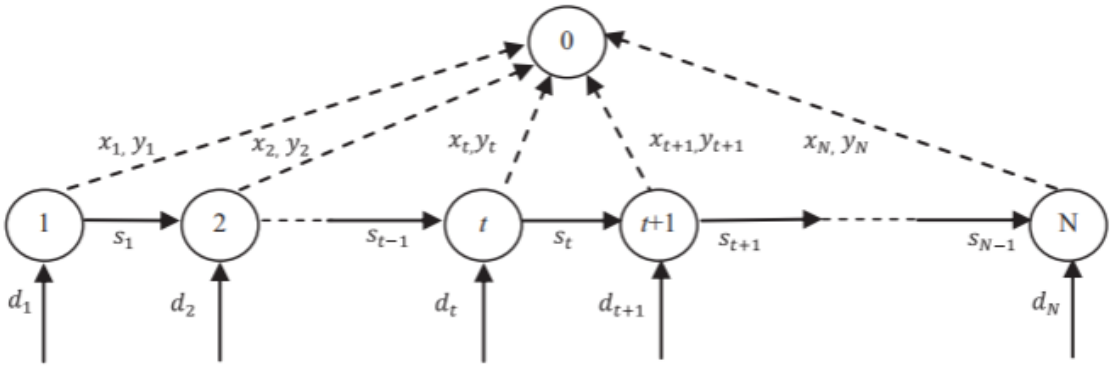


Figure 3.2: Reverse rappresentation of a DLSP

In the case of a multi-item problem, with M items,the index $j \in 1, ..., M$ represents one of the M items, so each parameter and variable presents a double index j and t. The formulation of the DLSP becomes

$$\min z = \sum_{t=1}^{N}\sum_{j=1}^{M}(p_{tj}x_{tj} + h_{tj}s_{tj} + f_{tj}y_{tj}) \tag{3.1.6}$$

$$s_{tj} = s_{t-1j} + x_{tj} - d_{tj} \qquad t = 1,...,N; j = 1,...,M \tag{3.1.7}$$

$$s_{tj} = 0 \qquad t = 0\, and\, t = N; j = 1,...,M \tag{3.1.8}$$

$$x_{tj} \leq C_{tj}y_{tj} \qquad t = 1,...,N; j = 1,...,M \tag{3.1.9}$$

$$\sum_{j=1}^{M} a_j x_{tj} + \sum_{j=1}^{M} b_j y_{tj} \leq R_t \qquad t = 1,...,N; j = 1,...,M \tag{3.1.10}$$

$$s_{tj} \geq 0;\ x_{tj} \geq 0;\ y_{tj} \in 0,1 \qquad t = 1,...,N; j = 1,...,M \tag{3.1.11}$$

where a_j is the capacity consumed for the production of one unit of item j,b_j is the capacity consumed for the setup of item j and R_t the total available capacity in period t.
Clearly according to the particular situations and needs it's possible to add to the model constraints to describe different production mode options, such as limits on the setups per period, limitations on the inventory and others:

$$\sum_{j=1}^{M} y_{tj} \leq K_t \qquad t = 1,...,N \tag{3.1.12}$$

$$\sum_{j=1}^{M} s_{tj} \leq S_t \qquad t = 1,...,N \tag{3.1.13}$$

$$s_{tj} \leq \sum_{k=1}^{\delta} d_{t+kj} \qquad t = 1,...,N-\delta; j = 1,...,M \tag{3.1.14}$$

$$y_{t+\lambda j} \leq (1 - y_{tj}) \qquad t = 1,...,N-\lambda; j = 1,...,M \tag{3.1.15}$$

$$y_{t+1j} \geq y_{tj} \qquad t = 1,...,N-1; j = 1,...,M \tag{3.1.16}$$

Contstraints (3.1.12) assume that at most K_t setups per period are allowed and (3.1.13) express a limitation to the total inventory level in each period. Constraint (3.1.14) impose an upper bound to the inventory level, (3.1.15) impose a minimum interval λ between two consecutive setups, and (3.1.16) impose that once the production of an item has been started, it will continue until the end of the planning horizon as y is a binary variable equal to 1 if item j is produced in period i. From this, for example, if in period 1 the production of j is active ($y_{1j} = 1$), it can no longer be interrupted until the end ($y_{t+1j} \geq y_{tj}, t = 1,...,N-1$). These last two constraints are useful to represent semi-continuous production processes.

For the reverse representation:

$$s_{tj} = s_{t-1j} - x_{tj} + d_{tj} \qquad t = 1,...,N; j = 1,...,M \tag{3.1.17}$$

$$s_{tj} \leq \sum_{k=1}^{\delta} x_{t+kj} \qquad t = 1,...,N-\delta; j = 1,...,M \tag{3.1.18}$$

3.2 Logistics applications

The general model of flow control can be useful to describe various optimization problems, seeing item j as a logistic service. In this way dimensioning and synchronization problems related to logistic services can be solved. In this case variables x_{tj} and s_{tj} represent respectively the demand for service j to be satisfied in period t, and the residual demand for service j at the end of period t, while y_{tj} is the activation of service j during t.
For example we can consider the following applications:

- **The bus terminal schedule optimization problem:** we have a bus transit terminal, where lines starting from a set of origins converge and users through them lines can reach a set of destinations. Important is the schedule of output lines towards the set of most common destinations. The problem aims to find tradeoffs solutions minimizing the activation costs of the output lines and the user's waiting costs.
  Adapting the model to this specific case, here d_{tj} is the number of passengers arrived at transit terminal at time t and directed to one of the destinations j, while y_{tj} is a binary variable equal to 1 if a bus leaves the terminal at time towards destination j. Assuming a reverse network flows representation, the problem can be viewed as the determination of passengers leaving the terminal at each time t towards destination j. The objective function for example can describe a performance measure defined as the sum of the costs associated with users waiting times and the costs associated with departing lines activation. In particular, in absence of unit production costs.
  The formulation of the model can include flows (passengers) conservation constraints, with the conditions that no passenger must be in the terminal at the beginning and at the end of the planning horizon, and constraints associated with the capacity of buses.

- **The cross-docking operations optimization problem:** in a complex supply chain, a cross-docking platform receives goods from suppliers and sorts them into alternative arrangements which have to be delivered to given destinations. This kind of systems requires a relevant synchronization between inbound and outbound flows in order to obtain both lower lead times and inventory costs. The aim of this problem is reduce the total distribution costs considering the benefits of a warehousing strategy in terms of consolidation and keeping minimum storage costs.

- **The check-in service optimization problem:** in an airport terminal, the check-in service consists in processing and accepting passengers arriving at designated desks. In this case an efficient management of such service is due to increasing air passengers' traffic and to a concurrent decrease in resources employed in handling operations. There's the necessity of cutting costs for airlines and third party providers due to the congestions of the terminal infrastructures and long waiting times and queues at check-in-desks. So the aim in this case is the optimization of the use of available check-in capacity.

The adaptation of all the elements of the basic version is easy, parameters and decision variables of the CLSP can be interpreted, in order to describe these specific applications, in the following way:

- **j:** service
- d_{tj}**:**Units of demand
- f_{tj}**:**Cost associated with the activation of service j in period t
- p_{tj}**:**Cost for satisfying a unit of demand for service j in period t
- h_{tj}**:**Cost for maintaining a unit of demand for service j in queue at the end of period t
- C_{tj}**:** Maximum number of units of demand for service j that can be satisfied in period t
- R_t**:** Total service capacity in period t
- a_j**:** Capacity consumption for satisfying a unit of demand for service j
- b_j**:** Capacity consumption for the activation of service j
- K_t**:** Maximum number of services that can be activated in period t
- S_t**:** Maximum demand still to be satisfied at the end of period t
- δ**:** Maximum waiting time for service demand
- x_{tj}**:** Units of demand for service j being processed in period t
- s_{tj}**:** Residual demand units for service j waiting to be processed at the end of period t
- y_{tj}: Binary variable concerning the activation of service j in period t

Capacitated Lot-Sizing model can be seen as a general model of flow control, in fact it's possible to use it to describe and formulate a great variety of optimization problems through simple adaptations of the basic version.

3.2.1 Lot sizing model under study

Let's see now the particular model analyzed at OMP. We have:

- d_t: Demand forecast for period t

Variables:

- S_t: Stock at the end of period t, $S_t = SPos_t - SNeg_t$
- P_t: Binary variable concerning the activation of the service in period t
- $TotBin$: Integer variable used for the clustering.

In every period of time, or bucket, there is a certain demand by the customer, and on this depends the resulting production. However, production is divided into two parts, one aimed at satisfying demand, and the other intended for stock. The stock resulting from production can be then used to satisfy demand in the following period.

$$\min z = \sum_{t=1}^{N} SPos_t + 10 \sum_{t=1}^{N} SNeg_t \tag{3.2.1}$$

$$10P_1 = S_1 + d_1 \tag{3.2.2}$$

$$S_{t-1} + 10P_t = S_t + d_t \qquad t = 2, ..., N \tag{3.2.3}$$

$$S_t = SPos_t - SNeg_t \qquad t = 1, ..., N \tag{3.2.4}$$

$$d_1 = 9.99 \tag{3.2.5}$$

$$d_t = 10 \qquad t = 2, ..., N \tag{3.2.6}$$

$$SPos_t \geq 0 \qquad t = 1, ..., N \tag{3.2.7}$$

$$SNeg_t \geq 0 \qquad t = 1, ..., N \tag{3.2.8}$$

$$P_t \in \{0, 1\} \qquad t = 1, ..., N \tag{3.2.9}$$

The objective function (3.2.1), aims to minimize the total Stock and, in particular, it aims to penalize above all the negative stock quantities. In fact, as can be seen in constraint (3.2.4), the total stock of a bucket corresponds to the difference between the positive stock quantity, i.e. what is left over from production, and the negative stock quantity, i.e. the demand that cannot be satisfied. What we would like is to have at most positive stock quantities, satisfying all demand, and reduced to a minimum. For this reason, in the objective function the negative stock is penalized with a factor of 10, unlike the positive stock.

Among the constraints we find the equilibrium constraint (3.2.3), which aims to balance inventory levels. In detail, the stock of the previous period added to the production of the current period has to be equal to the stock that is being produced, added to the current demand that we have to satisfy. Note that in our model the initial inventory level,or stock, is equal to 0, expressed on constraint (3.2.2).
We have then the demand forecast values ((3.2.5),(3.2.6)). In detail, these values were chosen as they are the ones that "extreme" the most the so-called "Ping Pong" behavior of the model, and make it more evident.
The relation balancing all the program is the following:

$$-10P_t - SPos_{t-1} + SPos_t + SNeg_{t-1} - SNeg_t = -d_t$$

that corresponds to the equilibrium constraint.

Using the OMP solver, when the clustering strategy is not applied, the program analyzes the problem variables one by one. That is, it performs branching for each time bucket. Although effective at first, as the computational complexity of the instances increases, the performance deteriorates, making the resolution less efficient. To overcome this problem, therefore, OMP uses aggregation, that is, it introduces the TotBin variable on which to perform aggregate branching of the problem variables, reducing time, complexity, and size of the branching tree. However, it has been noted that using SCIP, although it does not use clustering, manage to obtain the same performance as OMP solver, until a limit number of variables.
When the clustering strategy is used, the following constraints are added:

$$-\sum_{t=1}^{N} P_t + TotBin = 0 \tag{3.2.10}$$

$$TotBin \in I \tag{3.2.11}$$

That is, the integer variable TotBin is used for clustering variables P_t, $t = 1...N$.

Let's go deeper into this question. Assuming to have ten buckets, see 3.3.
As you can note, the inventory (stock) corresponds to the difference between the plan

#####	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
Demand	9,99	10	10	10	10	10	10	10	10	10
Plan	0	0	0	0	0	0	0	0	0	0
Inventory	-9,99	-19,99	-29,99	-39,99	-49,99	-59,99	-69,99	-79,99	-89,99	-99,99

Figure 3.3: Representation Ping-Pong branching (I)

(production), and the demand. In this case we have negative stocks as the demand is never satisfied. Let's go on analyzing the resolution of the model.
Assuming not to use the clustering strategy, we begin by solving the continuous relaxation of our problem. In *figure* 3.4 you can see the result. This is the first node of the Branch and Bound tree. At this point, as can be seen, the variable P_1, corresponding to the production of the first bucket, doesn't satisfy the integrality constraint. This is the first branching variable. We have two choices: setting the variable to 9 or 10. Since our objective function requires penalizing negative stocks, and since we want

#####	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
Demand	9,99	10	10	10	10	10	10	10	10	10
Plan	9,99	10	10	10	10	10	10	10	10	10
Inventory	0	0	0	0	0	0	0	0	0	0
Sol(P)	0,999	1	1	1	1	1	1	1	1	1

Figure 3.4: Representation Ping-Pong branching (II)

to satisfy the demand, we set P_1 to 10, and we go on. Once fixed the variable P_1, let's we solve the continuous relaxation again.
See *figure* 3.5. The same situation as before occurs. That is, the variable P_2 does not

#####	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
Demand	9,99	10	10	10	10	10	10	10	10	10
Plan	10	9,99	10	10	10	10	10	10	10	10
Inventory	0,01	0	0	0	0	0	0	0	0	0
Sol(P)	1	0,999	1	1	1	1	1	1	1	1

Figure 3.5: Representation Ping-Pong branching (III)

satisfy the integrity constraint as it is equal to 0.999. By branching on this variable, for the same reasons as above we set it to 10, and we go on.
As can be imagined, the process is repeated through all the variables. This is the so called "Ping-Pong" behavior of the Branch-and-Bound algorithm.
It is clear how, when we only have 10 variables, the Ping-Pong behavior does not bring any problems to the resolution of our model. When the number of variables increases it can cause major problems of loss of efficiency of the algorithm.
Precisely to overcome this problem we introduce `TotBin`, an aggregate decision sort that has a direct impact on all the other variables, by forcing the algorithm to immediately branch on this variable.
We do this by imposing through the constraints (3.2.10) and (3.2.11), that the sum of all production variables must equal an integer value, variable. For example, in the case of `figure` 3.5, `TotBin` would take the value of 9.999, not satisfying the integrality constraint. By branching on the latter instead of P_2, we would have an impact not only on P_1, but also on all subsequent variables, greatly increasing the efficiency of Branch-and-Bound.

CHAPTER 4

Parameter Configuration Problem

The analysis of the Lot Sizing Problem conducted at OMP consisted of searching for the best parameter configuration to ensure maximum efficiency. In other words, this operation is called Configuration Parameter Tuning. After presenting the automatic tuning technique, we move on to the manual one and give an explanation of how in this context the SCIP Optimization Suite tool was used during the work carried out at OMP.

The mathematical modeling of real-life optimization problems gives rise to complex, large-scale mixed-integer linear programs (MILP) with integer and binary variables, that require long computational times to solvers to return feasible solutions.

The parameter configuration problem consists of finding a parameter configuration that gives a particular algorithm the best performance on the given instance space, based on a specific criterion.
For this reason today the Automatic Parameter Tuning is one of the most practical remedies to reduce time, but it's possible to apply also a Manual Parameter Tuning, having the opportunity to personally test the desired parameters, for a more precise study. In both cases the aim is to to reduce time and improve the solution quality of these algorithms, for which the performance depends on the parameter combination used.

4.1 Automatic Parameter Tuning

Many automatic tuners are problem-dependent algorithms, they work specifically with an algorithm to tune and its application areas[1]. In detail, Model-based algorithms use explicit statistical models for studying the dependence of the algorithms on their parameters, and include the Sequential Model-Based Optimization (SMBO) and Sequential Parameter Optimization (SPO) approaches.

[1]More about automatic tuning is discussed in Ilyas Himmic [2023], Iommazzo *et al.* [2020] on references

The SMBO approach consists in iteratively building statistical models with available data and using them to analyze the configuration space by studying the interaction between parameters.
The SPO approach starts with constructing the initial configuration using Latin Hypercube Sampling (LHS). The parameter value interval is divided into equal intervals. Then, a random number is chosen from each interval to generate configurations. The performance of each generated configuration is measured after some executions, and the best performing configuration is selected as the initial configuration. Then, a stochastic Gaussian model is run to estimate the algorithm performance. This model is updated after each iteration based on the best-found configuration, which is then used in the subsequent iteration.
Model-free algorithms, not based on specific models, on the other hand, incorporate four classes techniques: Design of Experiments (DoE), Racing, Iterated Local Search (ILS), and Genetic Algorithms (GA).
The DoE approach consists of collecting data before analyzing it by statistical methods to draw valid conclusions, it decomposes parameter space to apply automated tuning procedures efficiently.
The Racing approach sequentially evaluates the candidate configurationtions and discards poor ones by evaluating the statistical data. This speeds up the procedure and allows the evaluation of promising configurations obtaining more reliable estimates of their behavior.
The ILS approach consists of an iterative call of local searches starting from new solutions, i.e a new parameter configuration, obtained using a perturbation of a previously found local optimum.
The GA approach relies on the GA metaheuristic, commonly used to reach high-quality solutions to optimization problems by relying on biologically inspired operators such as mutation, crossover, and selection.
We summarize the presented automatic parameter tuning algorithms on Fig.4.1.

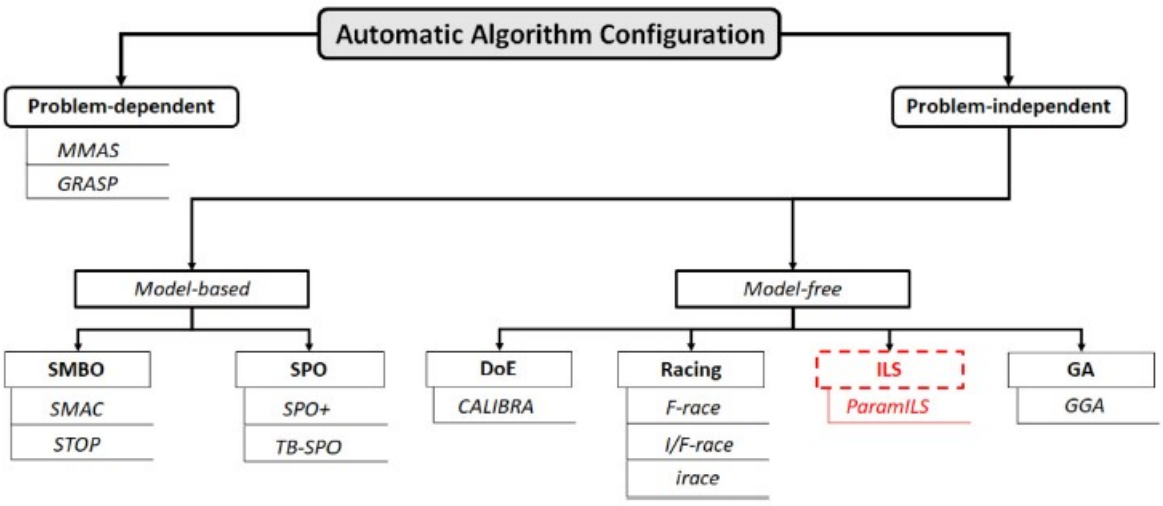


Figure 4.1: Automatic Parameter Tuning algorithms

4.2 Manual Parameter Tuning with Scip Optimization Suite

For the presented work, parameter tuning was performed manually through SCIP Optimization Suite.
SCIP is one of the fastest non-commercial solvers for mixed integer programming and mixed integer nonlinear programming, and it's also a framework for constraint integer programming and branch-cut-and-price. It combines solving techniques for CP, MIP, and satisfiability problems (SAT) such that all involved algorithms operate on a single search tree, which yields a very close interaction.
Integer Programming and Constraint Programming have different strenghts: Integer Programming uses LP relaxations and cutting planes to provide strong dual bounds, Constraint Programming can handle arbitrary constraints and uses propagation to tighten domains of variables.
It provides the infrastructure to implement very flexible branch-and-bound based search algorithms and it includes a large library of default algorithms to control the search.
Its approach provides high flexibility and the capability to manage differently each kind of constraint. This allows in many cases to consider constraints as a unique entity, without separating the inequalities it is composed by. The disadvantage of the constraint based approach is the limited global view of the problem, since a constraint knows its variable but a variable does not know the constraints it appears in.

It was created by the Zuse Institute of Berlin in 2002, a non-university research institute and computing centre, thanks to many developers who contributed to this project. The institute's research focuses on modeling, simulation and optimization with scientific cooperation partners from academia and industry.
Most ideas and algorithms of the state-of-the-art MIP solver SIP of Alexander Martin were transferred into the initial version of SCIP. Since then, many new features have been developed that further improved the performance and the usability of the framework.[2]
Nowadays SCIP counts more than 500 000 lines of source code and its development is still very active and the number of contributors is growing and growing.
SCIP can be used alone, but the SCIP Optimization Suite is available too. It is a complete source code bundle of SCIP, SoPlex, ZIMPL, GCG and UG. In combination with either SoPlex or CLP as LP solver, it is the fastest non-commercial MIP solver that is currently available.
SoPlex (Sequential object-oriented simPlex) is a Linear Programming solver based on the revised simplex algorithm. It features preprocessing techniques, exploits sparsity, and also offers primal and dual solving routines. It can be used as both a standalone solver and embedded into other programs.
ZIMPL (Zuse Institut Mathematical Programming Language) is a little language to translate the mathematical model of a problem into a linear or nonlinear mixed

[2] SCIP is freely available in source code for academic and non-commercial use and can be downloaded from http://scip.zib.de

integer mathematical program such that it can be read by a LP or MIP solver.
UG (Ubiquity Generator framework) is a generic framework to parallelize branch-and-bound based solvers in a distributed or shared memory computing environment.
GCG (Generic Column Generation) is a generic Branch-Cut-and-Price solver for mixed integer programs.

4.2.1 Branch-and-Bound in SCIP

SCIP is based on the branch-and-bound procedure. The idea of branching is to successively divide the given problem instance into smaller subproblems until the individual subproblems are easy to solve. The best of all solutions found in the subproblems yields the global optimum. During the course of the algorithm, a branching tree is created with each node representing one of the subproblems. The purpose of bounding is to avoid a complete enumeration of all potential solutions of the initial problem. If a subproblem's lower (dual) bound is greater than or equal to the global upper (primal) bound, the subproblem can be pruned. Lower bounds are calculated with the help of a relaxation. In order to improve a subproblem's lower bound, one can tighten its relaxation, e.g., via domain propagation or by adding cutting planes. Primal heuristics contribute to the upper bound. The selection of the next subproblem in the search tree and the branching decision have a major impact on how early good primal solutions can be found and how fast the lower bounds of the subproblems increase.
SCIP provides all necessary infrastructure to implement branch-and-bound based algorithms for solving CIPs. It manages the branching tree along with all subproblem data, automatically updates the LP relaxation, and handles all necessary transformations due to presolving problem modifications. Additionally, a cut pool, cut filtering, and a SAT-like conflict analysis mechanism are available. SCIP provides its own memory management and plenty of statistical output. Besides the infrastructure, all main algorithms of SCIP are implemented as external plugins. In particular, the following analysis is focused on Presolvers, Primal Heuristics, Separators plugins.

Presolvers

Presolving is a way to transform the given problem instance into an equivalent instance that is easier to solve.
The task of presolving is threefold: first, it reduces the size of the model by removing irrelevant information such as redundant constraints or fixed variables. Second, it strengthens the LP relaxation of the model by exploiting integrality information, e.g., to tighten the bounds of the variables or to improve coefficients in the constraints. Third, it extracts information such as implications or cliques from the model which can later be used, for example for branching or cutting plane separation.
SCIP implements a full set of primal and dual presolving reductions for MIP problems.
Restarts differ from the classical presolving methods in that they are not applied before the branch-and-bound search begins, but abort a running search process in

order to reapply other presolving mechanisms and start the search from scratch.

Primal Heuristics

Primal heuristics have a significant relevance as supplementary procedures inside a MIP solver: they help to find good feasible solutions early in the search process, which helps to prune the search tree by bounding and allows to apply more reduced cost fixing and other dual reductions that can tighten the problem formulation. Overall, there are 23 heuristics integrated into SCIP. They can be roughly subclassified into four categories:

- Rounding heuristics try to iteratively round the fractional values of an LP solution in such a way that the feasibility for the constraints is maintained or recovered by further roundings.
- Diving heuristics iteratively round a variable with fractional LP value and resolve the LP, thereby simulating a depth first search in the branch-and-bound tree.
- Objective diving heuristics are similar to diving heuristics, but instead of fixing the variables by changing their bounds, they perform "soft fixings" by modifying their objective coefficients.
- Improvement heuristics consider one or more primal feasible solutions that have been previously found and try to construct an improved solution with better objective value.

Cutting Plane Separators

Besides splitting the current subproblem Q into two or more easier subproblems by branching, one can also try to tighten the subproblem's relaxation in order to rule out the current solution and to obtain a different one. The LP relaxation can be tightened by introducing additional linear constraints that are violated by the current LP solution but do not cut off feasible solutions. Thus, the current solution is separated from the convex hull of integer solutions by the cutting plane.

4.2.2 Using SCIP

During the experimental phase of the work, tests on a series of models that differed in the number of variables, or temporal buckets, were carried out. This is because it has been noted that the behavior of the model is roughly the same until the number of buckets reaches a limit value, after which the efficiency of the algorithm decreases enormously. Through this tests we tried to detect possible aggregate branching decisions trying to individuate which elements represent a possible advantage for the B&B algorithm efficiency, and what the lack is in those cases where efficiency decreases. In this way, it's possible to guess how to improve the branching strategy limiting the depth of the search tree by taking such aggregate branching decisions.
In particular, the analysis was performed using SCIP Optimization Suite because

it has been noticed that, while using the default settings, solving specific types of models with OMP application,the resolution of the problem turns out to be inefficient, compared with the solution given by SCIP. Since SCIP permits switching the settings for all the parameters, the model resolution was tested with the different types of them. Subsequently, the results were compared by studying the statistics to understand the differences, advantages, and disadvantages.
SCIP was used as a pure CP/SAT solver by using the function $SCIPsetEmphasis()$. It's possible to change the behavior of SCIP switching the settings for all presolvers, heuristics, and separation plugins to three different modes via the `set{ presolving, heuristics, separating} emphasis` parameters in the interactive shell.
In detail, `off` turns off the respective type of plugins, while `fast` and `aggressive` respectively choose settings that lead to less ore more time spent in this type of plugins, making minimal or maximal their use or changing their frequency.
In this way, we could learn more about the presolve reasoning SCIP applies to the combinatorial optimization problem. Typing `display statistics` in the interactive shell it's possible to see which of the presolvers, propagators or constraint handlers performed the reductions.
Through the statistics it was possible to check which separators and heuristics are used, and the size of the search tree too, reported there as "nodes", and in particular the number of nodes that were processed during the search (see 4.2).
Furthermore to perform the analysis the application KDiff3 was used. It's a diff and merge program that compares or merges two or three text input files or directories and underlines the differences line by line and character by character. It also provides an automatic merge-facility and an integrated editor for comfortable solving of merge-conflicts. It prints of differences and gives an alignment of lines and has an intuitive graphical interface. It helped a lot to underline differences between models with different parameters settings.
Setting a parameter is done via the `set` command in the interactive shell:

```
set presolving/heuristics/separating emphasis
            off/aggressive/fast
```

To save statistics on files we used the following command, which saves only non-default params:

```
set diffsave <filename>.set
```

Once established the settings for the instances, we invoke SCIP directly with:

```
bin/scip -f <Modelname>.lp -s <filename>.set > <filename>.log
```

In the following figure (4.2) you can see the SCIP cycle resolution of an instance. First of all, SCIP is initialized by typing `SCIP` on the command window, and the model is read through `read <modelname>`. We have then the possibility to modify the parameters. After that, the model have to be presolved. Generally, without any specific command, SCIP automatically performs presolving and then moves directly to solving. However, if you want to control every step of the cycle, it's possible to

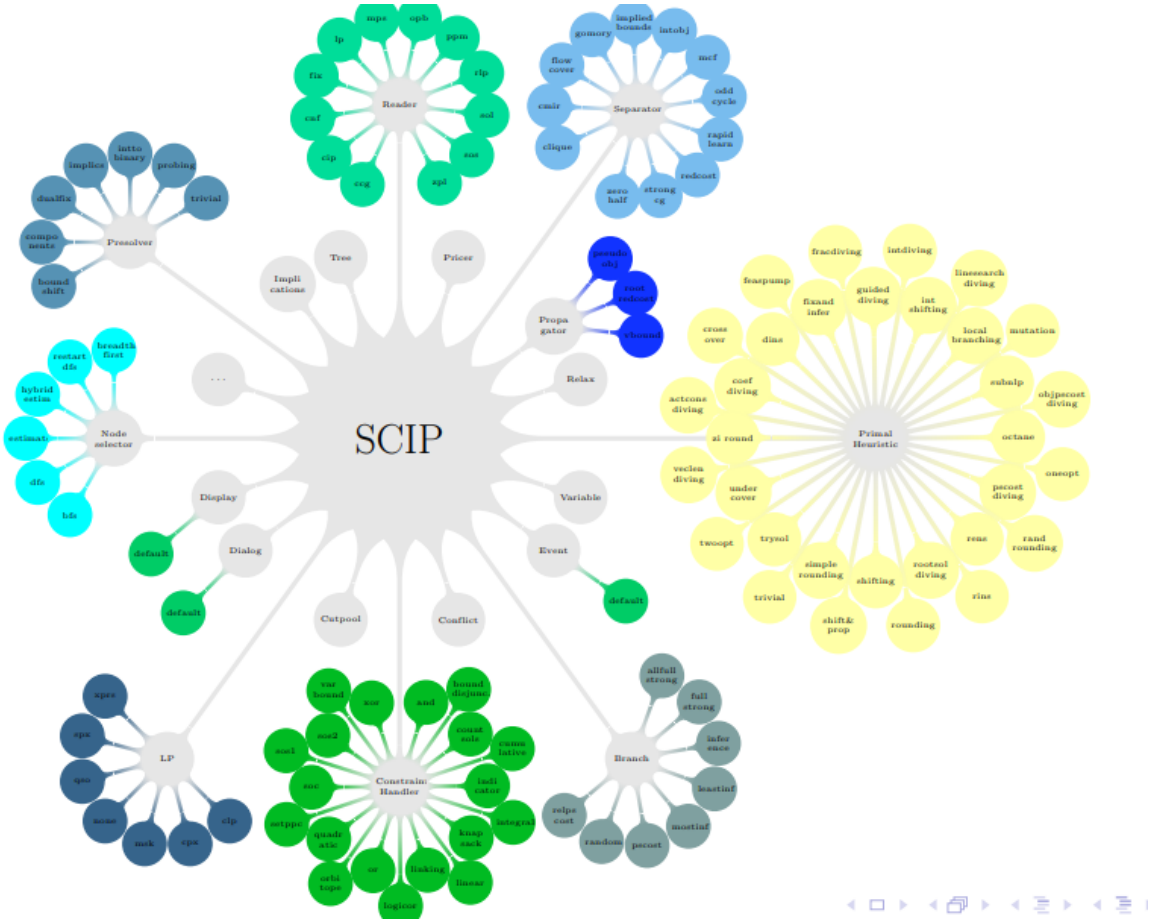


Figure 4.2: Graphical representation of all parameters viewable via SCIP

type `presolve` and view the results by `display statistics` and then initialize solving step with `solve`. Once reached the end of the cycle, the SCIP output consists of 3 relevant parts:

- Output for presolving progress
- Periodic status line during branch-and-bound in solving step
- Statistics to be displayed after solving was terminated/interrupted

The presolving output consists of the number of deleted/added variables and constraints etc. For each round, a problem summary at the end of presolving, and the presolving time.
The solving output consists of periodic status lines showing the elapsed time in seconds, the number of solved nodes, simplex iterations,the primal and dual bound, and the gap.
In particular, in this case we'll focus on presolving step, primal heuristics and separators.
There's also an analysis of how the program reacts to the change of parameters of

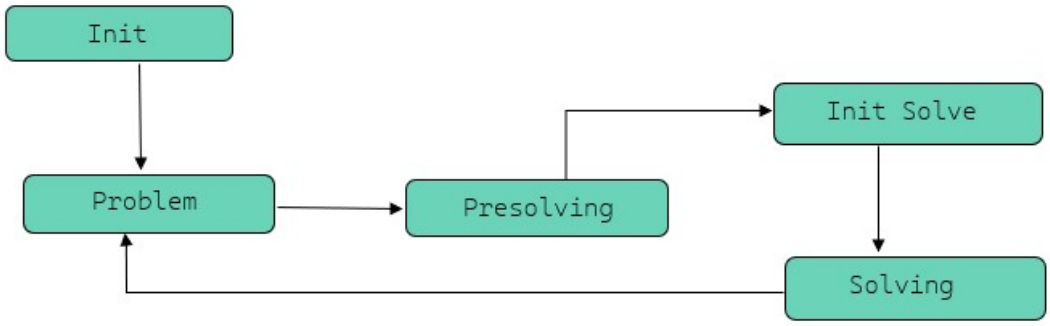


Figure 4.3: Graphic representation of SCIP cycle resolution of a problem

presolvers.

CHAPTER 5

Focusing on Primal Heuristics

In this chapter you can find a brief description with related algorithm of the primal heuristics of greatest impact in the analysis carried out. Knowledge of how they works is fundamental to be able to implement new heuristics and understand why they work well or not on the particular problem we are facing.

Generally speaking, primal heuristics play an important role in the solving of mixed integer programs as they often provide good feasible solutions early and help to reduce the time needed to prove optimality, and Branch-and-bound algorithm profits directly from finding good solutions as early as possible. On the one hand, these solutions originate from integral solutions to the linear programming (LP) relaxation, obtained by omitting the integrality restrictions, and is repeatedly solved for (sub-)problems during the branch-and-bound search to provide solution candidates and lower bounds. On the other hand, primal heuristics try to construct new feasible solutions or improve existing ones.

Searching for the best configuration, we noticed how some primal heuristics had more or less effect on the resolution of our problem. For this reason, it is interesting to delve deeper into them to try to understand what their functioning actually is and above all to understand why they work well in our case. In fact, through these analyses, studying the behavior of known heuristics on particular models, it is possible to have guidance on how to design even more specific heuristics according to one's needs.
Here are presented general functionalities and basic algorithms of some of the primal heuristics which have been protagonists of the research carried out on the Lot Sizing model through SCIP at OMP, that is, *Locks*, *Feaspump*, *Zirounding*, *Shift-and-Propagate*, *Intshifting*.

5.1 Locks

`locks` is one of the start heuristics that can be executed without previous knowledge of an LP solution or a previously found entire feasible solution. It uses global

structures, varible locks, available within MIP solvers to iteratively fix integer variables and propagate these fixings[1].

Variable locks are defined by the constraint matrix and take into account all given constraints. They are a measure of how many constraints may block an increase or decrease of the value of a variable.

The heuristic is motivated by greedy heuristics for set covering problems: Starting with an all-zero solution, one selects one variable which is contained in the highest number of constraints and fixes it to 1. By this, all these set covering constraints are fulfilled independently of the other variables' values. In subsequent steps, a variable is selected which is contained in the highest number of not-yet fulfilled constraints. It does not necessarily aim at fixing as many binary variables as possible. It constantly monitors how many of the constraints became redundant with respect to the tightened domains and stops the fixing phase as soon as all constraints became redundant. The values for all remaining variables can easily be determined by the subsequent LP solve. On the other hand, the variable-locks-driven fix-and-propagate heuristic is based on a strucure which covers the whole problem, in particular all binary variables, this means that it can always specify a fixing value for all of them.

How the variable-locks-driven fix-and-propagate heuristic translates this approach to general MIP is shown in the following algorithm on 5.1. In each iteration of the

input : - MIP $\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \mathcal{I})$
output: - index of binary variable x_k which should be fixed next; or -1 if no further fixings should be performed
- should x_k be fixed to it 0 (otherwise: 1)?
- result of the call: `continue` or directly `solve LP`

1 **begin**
  // get variable locks
2 **for** $i \in \mathcal{B}$ **do**
3 $\zeta_i^+ \leftarrow |\{r \in [1, \ldots, m] : a_{ri} > 0 \wedge \max\{A_r.x | \ell \leq x \leq u\} > b_r\}|$
4 $\zeta_i^- \leftarrow |\{r \in [1, \ldots, m] : a_{ri} < 0 \wedge \max\{A_r.x | \ell \leq x \leq u\} > b_r\}|$
  // select variable with highest sum of up- and down-locks
5 $k \leftarrow \arg\max_{i \in \mathcal{B}}\{\zeta_i^+ + \zeta_i^-\}$
  // no variable with locks left, stop fixing phase
6 **if** $\zeta_k^+ = \zeta_k^- = 0$ **then**
7 **return** *(0,FALSE,solve LP)*
  // fix variable to bound where it has fewer locks
8 **if** $\zeta_k^+ > \zeta_k^-$ **then**
9 **return** *(k,TRUE,continue)*
10 **else if** $\zeta_k^+ = \zeta_k^-$ **then**
11 **return** *(k,TRUE,continue)* with probability 33 %
12 **return** *(k,FALSE,continue)*

Figure 5.1: locks fixing algorithm

fixing process, a "high-impact" binary variable is selected, where the impact of a variable is decided based on the sum of its up- and down-locks, which corresponds to the number of constraints it is part of, cf. line 5. Then, the given variable is fixed

[1] Gamrath *et al.* [2019]

to the bound where it has the smaller number of locks, see lines 8 to 12. This aims at reaching feasibility fast and possibly ensuring that some constraints are already fulfilled after a few fixings, no matter how the values of the remaining variables in the constraint will be chosen within their updated bounds. If a variable has the same number of up- and down locks, a randomized approach is used to determine its fixing value. The variable is then fixed to 1 with a probability of 67%, where it showed a good performance. If a constraint is already fulfilled, its locks are disregarded (see lines 2–4), so that the impact and the fixing direction are always determined with respect to the not-yet fulfilled constraints only. Therefore, it may happen that all constraints are fulfilled already and none of the remaining variables has any locks left. In this case, the fixing procedure stops and returns that the LP should be solved directly in order to determine optimal values for the remaining variables, cf. lines 6–7.

5.2 Feasibility Pump

The `Feasibility Pump`[2] is probably the best known primal heuristic for mixed integer programming. The fundamental idea of all Feasibility Pump algorithms is to construct two sequences of points which hopefully converge to a feasible solution of a given optimization problem. One sequence consists of points which are feasible for a continuous relaxation, but possibly integer infeasible. The other sequence consists of points which are integral, but might violate some of the constraints. The next point of one sequence is always generated by minimizing the distance to the last point of the other sequence, by possibly using different distance measures in either cases.

The Feasibility Pump algorithm was originally introduced by Fischetti, Glover, and Lodi in 2005 for 0-1 mixed-integer linear programs, i.e., for the special case of MIPs in which $l_j = 0$ and $u_j = 1$ for all $j \in I$, where l_j and u_j are the lower and upper bound of the variable x_j and $I \subseteq N = \{1, ..., n\}$. The main idea is as follows. First, the LP relaxation of a MIP is solved. The LP optimum $\overline{x}$ is then rounded to the closest integral point

$$\overline{x} = \begin{cases} [\overline{x_j}] & if \quad j \in I \\ \overline{x_j} & if \quad j \notin I \end{cases} \tag{5.2.1}$$

where $[.]$ represent scalar rounding to the nearest integer. This part of the fp algorithm is called the *rounding step*. If $\overline{x}$ is not feasible for the linear constraints, the objective function of the LP is changed to the norm distance function

$$\Delta(x, \tilde{x}) := \sum_{j \in I} |x_j - \tilde{x_j}| = \sum_{j \in I : \tilde{x_j} = 0} x_j + \sum_{j \in I : \tilde{x_j} = 1} (1 - x_j) \tag{5.2.2}$$

on the set I of binary variables, and a new LP point $\overline{x}$ is obtained by minimizing $\Delta(x, \tilde{x})$ over the linear constraints of the MIP. The process is iterated until $\tilde{x} = \overline{x}$, which implies feasibility. The operation of obtaining a new $\overline{x}$ from $\tilde{x}$ is known as the *projection step*, as it consists of projecting $\tilde{x}$ to the feasible set of a continuous relaxation of the MIP along the direction $\Delta(x, \tilde{x})$. Two iterations of the algorithm

[2] Berthold *et al.* [2019]

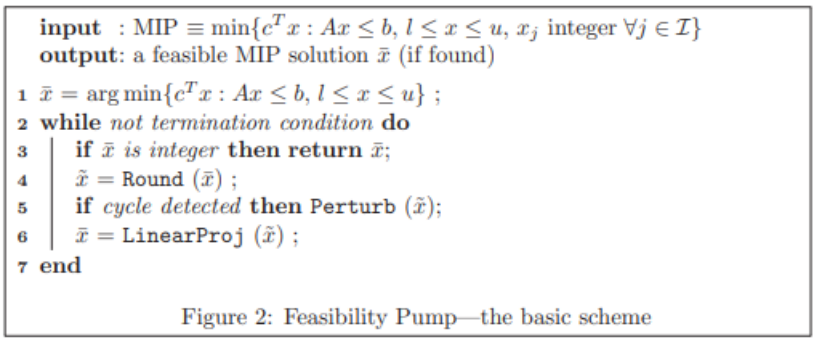


Figure 5.2: Feasibility Pump basic scheme

are illustrated for a simple example with a pseudocode description of the method is given in Figure 5.2.

The algorithm thus produces two sequences $\{\bar{x}^k\}_{k=1}^K$ and $\{\tilde{x}^k\}_{k=1}^K$for a finite K, which is either the iteration at which a feasible solution is found or some limit set to guarantee termination. All points of the sequence $\bar{x}^k$, with k denoting the iteration count of the FP, are feasible for the LP relaxation, all points $\tilde{x}^k$ are integral,i.e., $\tilde{x}^k \in \mathbb{Z}$ for all $j \in I$. Thus, $\tilde{x}^k = \bar{x}^k$ implies integrality and constraint-feasibility, which means that the corresponding point is feasible for the MIP.

5.3 Shift and Propagate

`Shift-and-Propagate`[3] is a pre-root primal heuristic that does not require a previously found LP solution. It applies domain propagation techniques to quickly drive a variable assignment towards feasibility. Computational experiments indicate that this heuristic is a powerful supplement of existing rounding and propagation heuristics.
The purpose of this primal heuristic is finding a feasible MIP solution at the very early stage of the solution process where no information about the root LP solution is available. In addition, it should be computationally cheap, using only domain propagation techniques.

The basic idea is as follows: in each iteration, the heuristic selects an unfixed variable $j \in K$ and a fixing value t_j^* within the domain of x_j , to which the variable is shifted. Then, domain propagation routines are called for this fixing. If domain propagation detects that fixing $x_j \to t_j^*$ is infeasible, a one-level backtrack-strategy is applied. Otherwise, the heuristic proceeds with the next unfixed variable. The goal of this heuristic is to find a good start solution, before the root node processing of a MIP solver starts, in particular prior to the first LP being solved. It might then serve as a reference point for improvement heuristics and for inferring further domain reductions.

[3] Berthold e Hendel [2015]

Input : MIP problem P
Output : a feasible solution of P, or **NULL** if search was not successful

$K \leftarrow \mathcal{I}$, $z \leftarrow 0$;
while $K \neq \emptyset$ **do**
  Select $j \in K$;
  Choose $t_j^* \in D_j$;
  Propagate $D_j \leftarrow \{t_j^*\}$;
  if *propagation detects infeasibility* **then**
    Apply backtrack strategy ;
  else
    $z_j \leftarrow t_j^*$;
    $K \leftarrow K \setminus \{j\}$;
  end
end
if z *is feasible for* P **then**
  return z ;
return NULL;

Figure 5.3: Basic Shift and Propagate algorithm

The general algorithm is described in 5.3.

5.4 Zirounding

`ZI Round`[4] is a pure integer rounding heuristic that attempts to round each fractional variable while using row slacks to maintain primal feasibility.
For integer variable x_j , define the fractionally of x_j as $ZI(x_j) = min\{x_j - \lfloor x \rfloor, \lceil x \rceil - x_j\}$. Also for solution x, define $ZI(x) := \sum_{i \in I} ZI(x_i)$, the integer infeasibility. The goal of ZI Round is to search the integer variables for ones that can be rounded to improve ZI until the integer infeasibility becomes zero at which point an integral solution has been found. ZI Round begin by calculating how much an integer variable can be moved within its bound while maintaining primal feasibility. To satisfy primal feasibility a variable shift of x_j must keep all slacks nonnegative.
So x_j can not be moved up more than ub, where $ub = min_i\{\frac{\bar{s}_i}{a_{ij}} : a_{ij} > 0\}$ for current slacks $\bar{s}$. Also x_j can not move past its upper bound, thus the shifting of x_j is limited by $UB := min\{ub, upperbound(x_j) - x_j\}$. Likewise x_j can not be shifted down more than $LB := min\{lb, x_j - lowerbound(x_j)\}$, where $lb = min_i\{\frac{-\bar{s}}{a_{ij}} : a_{ij} < 0\}$.
During ZI Round we first calculate UB and LB for variable x_j. We then move x_j to x_j + UB when $ZI(x_j + UB) < ZI(xj)$ or similarly we move x_j to $x_j - LB$ when $ZI(x_j - LB) < ZI(x_j)$. If x_j can be moved in both direction, we chose the direction which reduces $ZI(x_j)$ the most. If a tie occurs, i.e. both directions reduce $ZI(x_j)$ by the same amount, we round x_j in the direction which improves the objective function. We repeat this process until no more ZI improving shifts can be found. When ZI Round ends, if $ZI(x_j) = 0$ for all $j \in I$ then x is a feasible integer solution.
One detail that speeds up the ZI Round heuristic: stop calculating UB and LB if both fall below a predefined threshold. For x_j the threshold is a small positive number which we denote by ϵ. In practice we used $\epsilon = 0.00001$. Once UB < ϵ and LB < ϵ then there is no need to continue calculating UB and LB because $ZI(x_j)$ is limited by UB and LB and can change very little. Once a different variable has been rounded, the

[4] Wallace [2010]

ZI Round

Input Primal Feasible Point x
Repeat
  Loop For each fractional integer variable x_j
    1: Calculate (UB, LB, threshold $= \epsilon$)
      If $\mathrm{ZI}(x_j + \mathrm{UB}) = \mathrm{ZI}(x_j - \mathrm{LB})$ and $\mathrm{ZI}(x_j + \mathrm{UB}) < \mathrm{ZI}(x_j)$
        Then Update x_j to improve objective
            Update slacks $\hat{s}$
        Else If $\mathrm{ZI}(x_j + \mathrm{UB}) < \mathrm{ZI}(x_j - \mathrm{LB})$ and $\mathrm{ZI}(x_j + \mathrm{UB}) < \mathrm{ZI}(x_j)$
          Then Set $x_j := x_j + \mathrm{UB}$ and Update slacks $\hat{s}$
        Else If $\mathrm{ZI}(x_j - \mathrm{LB}) < \mathrm{ZI}(x_j + \mathrm{UB})$ and $\mathrm{ZI}(x_j - \mathrm{LB}) < \mathrm{ZI}(x_j)$
          Then Set $x_j := x_j - \mathrm{LB}$ and Update slacks $\hat{s}$
Until No more updates found

Figure 5.4: ZI Round

slacks change and it might be possible to round x_j . This is the reason ZI Round has two loops.

5.5 Intshifting

`Intshifting`[5] is a LP rounding heuristic that tries to recover from intermediate infeasibilities, shifts integer variables, and solves a final LP to calculate feasible values for continuous variables. The goal of rounding heuristics is to convert a fractional solution $\bar{\mathbf{x}}$ of the system $\mathbf{Ax} \leq \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$ into an integral solution, i.e., $x_j \in \mathbb{Z}\ \forall j \in I$. All rounding For a MIP, we call the number of positive coefficients $\psi^+ := \|\{i : a_{ij} > 0\}\|$ the up-locks of the variable x_j ; the number of negative coefficients is called the *down-locks* ψ^- of x_j.
`Rounding` heuristics performs roundings which potentially lead to a violation of some linear constraints, trying to recover from this infeasibility by further roundings later on. It takes up- and down-locks of an integer variable with fractional LP value $\overline{x_j}$ into account. As long as no linear constraint is violated, the algorithm iterates over the fractional variables and applies a rounding into the direction of fewer locks, updating the activities $\mathbf{A_i}\bar{\mathbf{x}}$ of the LP rows after each step, $\mathbf{A_i}$ being the i-th row of A. If there is a violated linear constraint, hence $\mathbf{A_i}\bar{\mathbf{x}} > \mathbf{b_i}$ for some i, the heuristic will try to find a fractional variable that can be rounded in a direction such that the violation of the constraint is decreased, using the number of up- and down-locks as a tie breaker. If no rounding can decrease the violation of the constraint, the procedure is aborted.
The `shifting` heuristic is similar to Rounding, but it tries to continue in the case that no rounding can decrease the violation of a linear constraint. In this case, the value of a continuous variable or an integer variable with integral value will be shifted in order to decrease the violation of the constraint. To avoid cycling, the procedure

[5] Achterberg *et al.* [2012]

terminates after a certain number of non-improving shifts. A shift is called non-improving, if it neither reduces the number of fractional variables nor the number of violated rows.

Part II

Experiments

CHAPTER 6

Analysis

The analyzes carried out on instances of the model under study described before, instances that differ in the number of temporal buckets and in the presence or absence of TotBin variable, used for aggregation, are presented. The idea is to study how the efficiency of the solver changes as the size of the model increases, and, if improvements are achieved without clustering, understand what they derive from.

Let's start from two first simple models with 10 variables, one of which makes use of clustering, and then continue with increasingly large models, respectively with 100 and 200 variables, until we reach a limit number of buckets, 1000, in which the efficiency of the problem decreases, and we find a different behavior from all the others.

To help the reader understand, an appendix containing definitions of various parameters is provided (Appendix B). Furthermore, for further curiosities about SCIP Optimization Suite you can consult the documentation[1].

For each model, an analysis of the program's behavior switching the settings for all presolvers, heuristics and separators was performed, in order to see how its efficiency could be changed by avoiding, strengthening, or making minimal their use.
Through SCIP, we are able to set the desired parameters, and, after running the program, we can save the results of the statistics on files to compare the different data more easily, using the application KDiff3 too.
In detail, the aim of all tests was to find the best configuration of each plugin in order to garantee the fastest end most efficient resolution of the problem considered.
First of all, for each plugin, the best configuration isolating it from all the others is evaluated. After that, we have to consider particular cases taking in consideration how the operations of the various plugins influence each other by working separately or in competition.
The following presentation of the results will be supported by the use of tables, provided with all the data taken directly from the analyzed statistics. On this Tables,

[1] Bestuzheva *et al.* [2023] on Bibliography

you can find the data provided for all plugins (`P=Presolving`, `H=Heuristics`, `S=Separating`) with different settings (`d=default`, `off=emphasis off`, `a=aggressive`, `f=fast`). In detail:

$$\begin{aligned} &\texttt{Psetting} \in \{\texttt{d, off, a, f}\} \\ &\texttt{Hsetting} \in \{\texttt{d. off, a, f}\} \\ &\texttt{Ssetting} \in \{\texttt{d, off, a, f}\} \end{aligned}$$

6.0.1 Presolving

To study the actual impact of the presolving process on problem resolution, we want to evaluate the results given by the presolvers by isolating them from all others parameters. We can do this by deactivating everything that could somehow alter the results given by presolving, i.e. heuristics and separators. These tests for all models are performed in the following way:

- `set presolving default/emphasis off`
  `set heuristics emphasis off`
  `set separating emphasis off`

and then the results will be compared.

Performing these tests for PingPong model (see page 29), we obtain interesting results, as shown on Table 6.1. PingPong is the model with 10 variables in which the integer variable `TotBin` is introduced. When presolving process is executed `TotBin` is deleted, so the program doesn't make use of aggregation. On the contrary when all presolvers are disabled, `strong branching` heuristic comes in to play, as one of the heuristics which can't be disabled by SCIP. As shown on Table 6.1 the number of nodes in this last case decreases respect to the 12 nodes of the first case. In this case SCIP benefits from the use of clustering strategy.

B&B tree	P=d,H=S=off	P=H=D=off
Number of runs	1	1
Nodes	**12**	**1**
Feasible leaves	**1**	**0**
Infeas.leaves	1	1
Objective leaves	0	0
Nodes (total)	**12**	**1**
Nodes left	0	0
Max depth	**10**	**0**
Max depth (total)	**10**	**0**
Backtracks	**9**	**0**
Early backtracks	0	0
Nodes exc. Ref.	0	0
Delayed cutoffs	**9**	**0**
Repropagations	**9**	**0**
Avg switch length	**4.83**	**2.00**
Switching time	0.00	0.00
Solving time	0.00	0.00

Table 6.1: Branch and Bound tree data analysing presolving on model PingPong

Regarding models which make no use of clustering, referring to Table 6.2, it is clear how the presolving process doesn't influence the resolution of the program. For exception of a small difference on model PingPong1000, the branch and bound tree on all other cases remains unchanged, with and without presolvers.

B&B tree	P=H=S=off				P=d.H=S=off			
	PP10	PP100	PP200	PP1000	PP10	PP100	PP200	PP1000
Number of runs	1	1	1	1	1	1	1	1
Nodes	12	102	202	**1997**	12	102	202	**1996**
Feasible leaves	1	1	1	**216**	1	1	1	**209**
Infeas.leaves	1	1	1	**681**	1	1	1	**687**
Objective leaves	0	0	0	100	0	0	0	100
Nodes (total)	12	102	202	**1997**	12	102	202	**1996**
Nodes left	0	0	0	0	0	0	0	0
Max depth	10	100	200	1000	10	100	200	1000
Max depth (total)	10	100	200	1000	10	100	200	1000
Backtracks	9	99	200	**889**	9	99	200	**895**
Early backtracks	0	0	0	0	0	0	0	0
Nodes exc. Ref.	0	0	0	0	0	0	0	0
Delayed cutoffs	9	99	199	**4**	9	99	199	**5**
Repropagations	9	99	200	**26494**	9	99	200	**20096**
Avg switch length	4.83	5.86	5.94	**51.27**	4.83	5.86	5.94	**43.40**
Switching time	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Solving time	0.00	0.00	0.00	**11.00**	0.00	0.00	0.00	**10.00**

Table 6.2: Branch and Bound tree data analysing presolving impact on the resolution of the problem

To know which presolvers are used for each case in all models, on Table A.1 you can find how many calls are done for each presolver. Note that, for all models, not only same presolvers are used, but also with the same number of calls.

In order to confirm this result, we have to study the impact of presolvers when they are part of the solving process together with primal heuristics or separators, performing the following tests:

- ```
  set presolving default/emphasis off
  set heuristics default
  set separating emphasis off
  ```

- ```
  set presolving default/emphasis off
  set heuristics emphasis off
  set separating default
  ```

So, maintaining primal heuristics or separators active, on the following tables (6.3, 6.4) you can find the results obtained on B&B tree switching on and off presolvers, for models without `TotBin` variable.

B&B tree	H=d, P=S=off		H=P=d, S=off	
	PPx	PP1000	PPx	PP1000
Number of runs	1	1	1	1
Nodes	1	**1612**	1	**1611**
Feasible leaves	0	0	0	0
Infeas.leaves	0	**627**	0	**626**
Objective leaves	**1**	1	**0**	1
Nodes (total)	1	**1612**	1	**1611**
Nodes left	0	0	0	0
Max depth	0	984	0	984
Max depth (total)	0	984	0	984
Backtracks	0	**404**	0	**417**
Early backtracks	0	0	0	0
Nodes exc. Ref.	0	0	0	0
Delayed cutoffs	0	**357**	0	**358**
Repropagations	0	**627**	0	**3692**
Avg switch length	2.00	**6.64**	2.00	**8.42**
Switching time	0.00	0.00	0.00	0.00
Solving time	0.00	**6.00**	0.00	**5.00**

Table 6.3: Branch and Bound tree data analysing presolving impact mantaining primal heuristics active. x $\in \{10, 100, 200\}$.

B&B tree	S=d, P=H=off				S=P=d, H=off			
	PP10	PP100	PP200	PP1000	PP10	PP100	PP200	PP1000
Number of runs	1	1	1	1	1	1	1	1
Nodes	**3**	102	202	**1997**	**5**	102	202	**1999**
Feasible leaves	1	1	1	**287**	1	1	1	**267**
Infeas.leaves	1	1	1	**600**	1	1	1	**625**
Objective leaves	0	0	0	**110**	0	0	0	**107**
Nodes (total)	**3**	102	202	**1997**	**5**	102	202	**1999**
Nodes left	1	0	0	0	0	0	0	0
Max depth	**1**	100	200	1000	**3**	100	200	1000
Max depth (total)	**1**	100	200	1000	**3**	100	200	1000
Backtracks	**0**	99	200	**827**	**3**	99	200	**844**
Early backtracks	0	0	0	0	0	0	0	0
Nodes exc. Ref.	0	0	0	0	0	0	0	0
Delayed cutoffs	**0**	99	199	**4**	**2**	99	199	**2**
Repropagations	**0**	99	200	**20754**	**3**	99	200	**24972**
Avg switch length	**2.00**	5.86	5.94	**42.47**	**3.60**	5.86	5.94	**51.36**
Switching time	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00
Solving time	0.00	0.00	**1.00**	**12.00**	0.00	0.00	**0.00**	**11.00**

Table 6.4: Branch and Bound tree data analysing presolving impact mantaining separators active

From Table 6.3, emerges that when heuristics are active, for models under 1000 variables, same B&B tree is obtained, while on PingPong1000 there is a small difference of one node between cases with and without presolvers. On the contrary, as shown on Table 6.4, when we maintain separators active but all heuristics are disabled, presolving process influences not only the resolution of PingPong1000 like the previous case, but also PingPong10. In fact, we can see that in absence of presolving process B&B tree presents less nodes making resolution more efficient.
For further data, on Table A.3 on page 96 of Appendix A, you can find which separators are used for each model performing this analysis.

When primal heuristics are active, in all cases, even if in PingPong1000, for the complexity of the problem, more primal heuristics come in to play, but solution is always found by `locks` heuristic[2] (see Table A.2), with and without presolving process.

Performing the same tests on PingPong model (Tables 6.5,6.6), different results are obtained activating heuristics and separators.
On the first case the presence of primal heuristics ensure that, even in the absence of presolvers, `TotBin` variable is not used. In detail, when presolving process is executed, `TotBin` variable is deleted, so that when solve operation is executed the model corresponds in structure and behavior to PingPong10 model, obtaining the same branch and bound tree. When presolvers are disabled, like in all other cases, solution is found by `locks`, and the tree presents one more leaf due to the necessity to pass also through the variable `TotBin`, but without benefiting from clustering. You can check this from the following Table 6.5.

B&B tree	H=d, P=S=off	H=P=d, S=off
Number of runs	1	1
Nodes	1	1
Feasible leaves	0	0
Infeas.leaves	0	0
Objective leaves	**1**	**0**
Nodes (total)	1	1
Nodes left	0	0
Max depth	0	0
Max depth (total)	0	0
Backtracks	0	0
Early backtracks	0	0
Nodes exc. Ref.	0	0
Delayed cutoffs	0	0
Repropagations	0	0
Avg switch length	2.00	2.00
Switching time	0.00	0.00
Solving time	0.00	0.00

Table 6.5: Branch and Bound tree data analysing presolving on model PingPong, mantaining primal heuristics active

Instead, when we put on separators but we disable primal heuristics, the situation is similar to that of Table 6.1. Note in particular that on tables 6.1,6.6 there are the same data. This is because even in this case in absence of heuristics, when presolving process is not ran and the variable `TotBin` can't be deleted, the program takes advantage from the use of clustering. In fact, as before, `strong branching` heuristic finds the solution.

[2]See Heuristic locks.h File Reference on Bibliography

B&B tree	S=d, P=H=off	S=P=d, H=off
Number of runs	1	1
Nodes	**1**	**5**
Feasible leaves	**0**	**1**
Infeas.leaves	1	1
Objective leaves	0	0
Nodes (total)	**1**	**5**
Nodes left	0	0
Max depth	**0**	**3**
Max depth (total)	**0**	**3**
Backtracks	**0**	**3**
Early backtracks	0	0
Nodes exc. Ref.	0	0
Delayed cutoffs	**0**	**2**
Repropagations	**0**	**3**
Avg switch length	**2.00**	**3.60**
Switching time	0.00	0.00

Table 6.6: Branch and Bound tree data analysing presolving on model PingPong, mantaining separators active

Drawing conclusions from previous analyses, in general, in models under 1000 variables, which don't present the integer variable `TotBin`, the presolving process does not influence the resolution of the problem at all. Only a little attention can be paid to the PingPong10 model.
On the contrary, in PingPong model, the absence of presolvers is decisive as it allows the use of clustering strategy, greatly improving the efficiency of the problem thanks to the intervention of `strong branching heuristic`.
In PingPong1000, although there are changes in B&B tree, they don't make a big difference in the structure since the difference in nodes is extremely small compared to the total of them. However, extra attention needs to be paid to this model as presolving process has an impact on resolution time. In particular, as shown on Tables 6.3,6.4,6.2, presolvers make the program faster, and consequently more efficient.

Consequentially, from now on, having completely different executions due to the use of aggregation, PingPong model will be excluded in the following tests, focusing only on the models without `TotBin` variable, being able to make a comparison between them.

Regarding these models, it is not enough to know that presolvers don't affect the resolution of the problem when set in default, because it's important to know what their best configuration is. So, in order to find it, it's necessary to make a comparison among results obtained running the program in default, and switching their setting with `presolving emphasis fast/aggressive`. As a result of the previous analysis, performing these additional tests, we can leave primal heuristics and separators in default setting.
Firstly, the program in default setting is ran, and the results of statistics are saved in a first file. After that, the problem is ran again after switching the setting for all presolvers and the results are saved in a second file. Then, the two files are compared to underline differences. Let's do this in the following way through command line:

```
scip
read <PingPongx>
set diffsave file1.set
quit
scip -f Ping-Pong.lp -s file1.set > file1.log
scip
read Ping-Pong.lp
set presolving emphasis off/aggressive/fast
set diffsave file2.set
quit
scip -f Ping-Pong.lp -s file2.set > file2.log
fc file1.log file2.log
```

Presolvers	P=H=S=d	H=S=d, P=a	H=S=d, P=f
	PPx	PPx	PPx
boundshift		1	
domcol	1	1	
dualagg		1	
dualcomp	1	1	1
dualinfer		1	
dualsparsify	1	1	
implics	1	1	1
inttobinary			
milp	1	1	1
redvub		1	
sparsify	1	1	
stuffing		1	
trivial	1	1	1
tworowbnd		1	
dualfix	1	1	1
probing	1	1	
symmetry	1	1	1
linear	2	2	2
components	1	1	

Table 6.7: Presolving behavior switching setting for all presolvers

On Table 6.7 you can see in detail which presolvers are called for each case analyzed. On the first column you can see all presolvers, the values represent the number of calls made to each of them, for each model, switching their setting. In detail, on the first multicolumn we have all presolvers used running the program in default setting, and these data are compared to those used setting `presolving emphasis fast/aggressive`, on the following multicolumns. There are no data for the setting `presolving emphasis off` as, although with SCIP is not possible to disable some presolvers, in this case these ones are not used. In all cases, stregthening the impact of presolving with `presolving emphasis aggressive`, because of the greater number of presolvers used, more operations during presolving process are done. On the contrary with `presolving emphasis fast` and `presolving emphasis off`, the process is less burdensome, if not absent, however bringing a series of disadvantages which will have an impact above all on the B&B tree.

Furthermore, note on Table 6.8 (Solving time), that switching setting for presolvers on model PingPong1000 influence solution time too. In particular, setting `presolving emphasis off/aggressive` slows down the resolution of the problem.

B&B tree	P=H=S=d		P=off/f, H=S=d		P=a, H=S=d	
	PPx	PP1000	PPx	PP1000	PPx	PP1000
Number of runs	1	1	1	1	1	1
Nodes	1	**1611**	1	**1612**	1	**1609**
Feasible leaves	0	**0**	0	**1612**	0	**0**
Infeas.leaves	0	**626**	0	**627**	0	**624**
Objective leaves	**0**	**1**	**1**	**627**	**0**	**1**
Nodes (total)	1	**1611**	1	**1612**	1	**1609**
Nodes left	0	0	0	0	0	0
Max depth (total)	0	984	0	984	0	984
Backtracks	0	**417**	0	**404**	0	**454**
Early backtracks	0	0	0	0	0	0
Nodes exc Ref	0	0	0	0	0	0
Delayed Cutoffs	0	**358**	0	**357**	0	**360**
Repropagations	0	**3692**	0	**627**	0	**2961**
Avg switch length	2.00	**8.42**	2.00	**6.64**	2.00	**8.11**
Solving Time	0.00	**5.00**	0.00	**6.00/5.00**	0.00	**6.00**

Table 6.8: Analysis of Branch and Bound tree switching the setting for all presolvers

On Table 6.8, you can find also how the B&B tree changes in each model, switching the setting for all presolvers. In detail, we find on the first column the list of the different parameters of the B&B tree to evaluate. Then we have three multicolumns where we find the default values of B&B tree, and on the subsequent, the values acquired by the parameters switching the setting, imposing `presolving emphasis off/aggressive/fast`, for each model. PPx is a parameter indicating models which make no use of aggregation under 1000 variables ($x \in \{10, 100, 200\}$). PP1000 is the model with 1000 variables and, as can be seen, shows a completely different behavior than all others ones, in which same results are obtained.
Using presolving emphasis fast the B&B tree turns out to be the same as in the case in which the presolvers are reset. For models under 1000 variables, comparing with results obtained running the program in default, we have a different value for the parameter `objective leaves`, the number of processed leaf nodes that hit LP objective limit (for more informations about B&B tree parameters go to Appendix B). This could be due to the lack of some presolvers in running the program with `presolving emphasis fast`, such as `domcol`, `dualsparsify`, `sparsify` , `probing`, `components` (referring to Table 6.7).
The optimal solution in all cases is the same and it's always found by `locks` heuristic. Since no particular differences occur when using `presolving emphasis aggressive`, using this setting doesn't bring particular advantages. On the contrary it makes the program less efficient, increasing the number of operations done during presolving process without useful results.

Completely different is PingPong1000 case. In fact, although the presolvers used are the same as in the previous cases, as evident in Table 6.7, the mechanism that in some way kept the size of the problem limited in previous models is missing here. Also in

this case the presolving process is unable to eliminate variables and constraints and from the results we already have 1611 nodes in the default case. The number of nodes worsens if we impose `presolving emphasis off/fast`, while it undergoes a slight increase by strengthening the use of presolvers, which however is insufficient to improve its efficiency because it slows down the resolution of the model (see on Table 6.8 Solving Time).

In order to understand what the difference in the first three models in the B&B tree is due to, we have to study in more detail the behavior of individual presolvers to know what their contribution is to this result.
In fact, if we recognize that one special plugin works poorly or well for my problem, we can change/disable some parameters of the problem in order to study their influence on it. In this case, the following code is used to disable presolvers:

```
SCIP > set presolvers <name of a presolver> maxrounds 0
```

and the result with its statistics are saved into files.
The value of `objective leaves` parameter is 1 only in `presolving emphasis off` and `presolving emphasis fast` cases, while it is equal to 0 in default and `heuritics emphasis aggressive` cases. We could therefore assume that in `presolving emphasis fast` case, there is an absence of presolvers that could make a difference in solving the problem. So, let's try to deactivate these presolvers in the original model, one by one, to understand how they influence the solution. It's possible to do it only with `domcol, dualsparsify, sparsify` presolvers. In fact with SCIP some of them like `probing` and `components` can't be disabled.
In all cases, no difference emerges in the B&B tree, while trying to disable one or two of these presolvers. This make think that the change could be due to the absence of all three presolvers taken into consideration or to the order in which they are used in the problem. Furthermore it could be due also to one of those presolving parameters which is not possible to modify. For example with `presolving emphasis fast` the probing cycle present in the default case is not carried out.

From the previous tests, since the small differences produced by switching the setting don't bring particular advantages, we can assume that the best configuration for the presolvers is the default setting. Consequently, from now on, to perform subsequent tests on primal heuristics and separators, presolvers setting won't be changed, leaving them in default.

6.0.2 Primal Heuristics and Separation configuration search

In order to understand the best configuration for primal heuristics and separators, first of all, we evaluate the behavior of the program by switching their setting, isolating them from everything else. In detail, as already seen, the presolving process does not influence the resolution of the problem, therefore we can leave the presolvers in default, but in order to be able to precisely evaluate the results given by primal heuristics or separators respectively, it is necessary that the one or the other are initially deactivated.
For primal heuristics, the following test are performed:

- `set presolving default`
  `set heuristics emphasis default/aggressive/fast`
  `set separators emphasis off`

On Table 6.9, you can find the results on B&B tree obtained for each model. Note that model PingPong1000 shows again a completely different behavior than all the others. Regarding models under 1000 variables, setting `heuristics emphasis aggressive` doesn't bring any advantage to the resolution. In fact, despite more operations are carried out, branch and bound tree and solution time remain unchanged. Setting `heuristics emphasis fast` the results are even worse, as there is an important increase of the number of nodes.
Therefore, it is clear how for these models the best configuration for the heuristics is the default.
On the contrary, model PingPong1000 presents a great improvement setting `heuristics emphasis fast`. In fact, B&B tree nodes decrease until 1151, against the 1611 and 1985 of the others cases, and through this configuration the fastest resolution is also obtained, of only 3.00 seconds.

B&B tree	P=H=d, S=off		P=d, H=a, S=off		P=d, H=f, S=off			
	PPx	PP1000	PPx	PP1000	PP10	PP100	PP200	PP1000
Number of runs	1	1	1	1	1	1	1	1
Nodes	**1**	**1611**	**1**	**1985**	**21**	**201**	**401**	**1151**
Feasible leaves	0	0	0	0	0	0	0	0
Infeas.leaves	**0**	**626**	**0**	**992**	**10**	**100**	**200**	**157**
Objective leaves	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**1**
Nodes (total)	**1**	**1611**	**1**	**1985**	**21**	**201**	**401**	**1151**
Nodes left	0	0	0	0	0	0	0	0
Max depth	**0**	**984**	**0**	**992**	**10**	**100**	**200**	**993**
Max depth (total)	**0**	**984**	**0**	**992**	**10**	**100**	**200**	**993**
Backtracks	**0**	**3**	**0**	**418**	**9**	**99**	**200**	**1002**
Early backtracks	0	**417**	0	**0**	0	0	0	**0**
Nodes exc. Ref.	0	0	0	0	0	0	0	0
Delayed cutoffs	0	**358**	0	**0**	0	0	0	**8.36**
Repropagations	**0**	**3692**	**0**	**1245**	**1**	**1**	**1**	**5805**
Avg switch length	**2.00**	**8.42**	**2.00**	**34.54**	**3.62**	**3.96**	**2.99**	**22.19**
Switching time	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Solving time	0.00	**5.00**	0.00	**22.00**	0.00	0.00	0.00	**3.00**

Table 6.9: Branch and Bound tree data analysing heuristics setting all separators off

In addition, if you want to know which heuristics[3] are used for each model for each

[3] consult the Bibliography for more about specific heuristics structure and files

setting you can check it on Table A.4 on page 97 of Appendix A.

Performing the same test on separators deactivating heuristics, in the following way:

- `set presolving default`
  `set heuristics emphasis off`
  `set separating emphasis default/aggressive/fast`

we find the results on B&B tree represented on Table 6.10 on page 62. Running the program setting separators default or imposing `separating emphasis aggressive` B&B tree remains unchanged, while setting `separating emphasis fast` there is an increment of number of nodes, and on PingPong1000 resolution is slower. In conclusion, the best configuration for separators seems to be the default.

However, we cannot jump to conclusions by looking only at previous results. In the case of presolvers we were able to conclude which was the best configuration because, by acting before the solving phase, they don't work in parallel with the plugins that operate later, but only deal with simplifying the problem, when possible, before moving on to the resolution.
On the contrary, with separators and heuristics it is necessary to consider the best combination of settings to find the ideal configuration for solving our problem. In fact, they influence each other by working in parallel and in competition. Consequently, although considering them in isolation we obtained useful results for a first general idea, it is possible that through further tests the process can be further improved, as demonstrated in the following paragraphs. In detail, setting presolvers in default, as shown in the following figure, we have to explore 4^2 configurations given by combinations of settings of heuristics and separators.

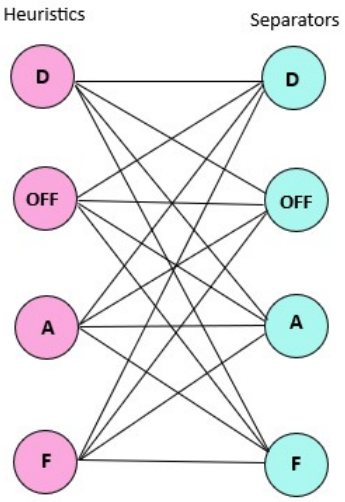


Figure 6.1: Configurations given by combinations of settings of heuristics and separators.

B&B tree	P=S=d, H=off				P=d, H=off, S=a				P=d, H=off, S=f			
	PP10	PP100	PP200	PP1000	PP10	PP100	PP200	PP1000	PP10	PP100	PP200	PP1000
Number of runs	1	1	1	1	1	1	1	1	1	1	1	1
Nodes	**5**	102	202	1999	**5**	102	202	1999	**7**	102	202	1999
Feasible leaves	1	1	1	**267**	1	1	1	**267**	1	1	1	**264**
Infeas.leaves	1	1	1	**625**	1	1	1	**625**	1	1	1	**629**
Objective leaves	0	0	0	**107**	0	0	0	**107**	0	0	0	**106**
Nodes (total)	**5**	102	202	1999	**5**	102	202	1999	**7**	102	202	1999
Nodes left	0	0	0	0	0	0	0	0	0	0	0	0
Max depth	**3**	100	200	1000	**3**	100	200	1000	**5**	100	200	1000
Max depth (total)	**3**	100	200	1000	**3**	100	200	1000	**5**	100	200	1000
Backtracks	**3**	99	200	**844**	**3**	99	200	**844**	**5**	99	200	**845**
Early backtracks	0	0	0	0	0	0	0	0	0	0	0	0
Nodes exc. Ref.	0	0	0	0	0	0	0	0	0	0	0	0
Delayed cutoffs	**2**	99	199	2	**2**	99	199	2	**4**	99	199	2
Repropagations	**3**	99	200	**24962**	3	99	200	**24962**	**5**	99	200	**20405**
Avg switch length	**3.60**	5.86	5.94	**51.36**	**3.60**	5.86	5.94	**51.36**	**4.29**	5.86	5.94	**41.73**
Switching time	0.00	0.00	0.00	**1.00**	0.00	0.00	0.00	**0.00**	0.00	0.00	0.00	**1.00**
Solving time	0.00	0.00	0.00	**11.00**	0.00	0.00	0.00	**11.00**	0.00	0.00	0.00	**12.00**

Table 6.10: Branch and Bound tree data analysing separators setting off all heuristics

Primal Heuristics

An additional analysis of primal heuristics is performed in the following way:

- `set presolving emphasis default`
  `set heuristics emphasis defaut/off/aggressive/fast`
  `set separating emphasis default`

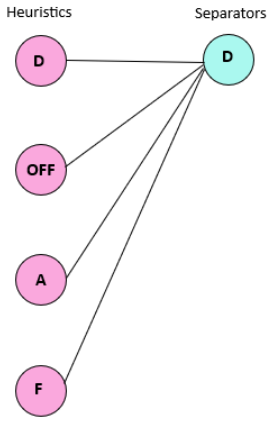


Figure 6.2: Configurations given by setting presolving and separating default, switching the setting for heuristics.

On Table 6.11 on page 64 it's possible to see which heuristics act in different cases and how they influence the resolution of the model. In detail, for each heuristic there are how many times it was called, if it found a feasible solution and if the solution found was the current best solution. Running the program in default and setting `heuristics emphasis aggressive` all models under 1000 variables presents the same situation (PPx : x $\in \{10, 100, 200\}$).

When we deactivate heuristics, the number of B&B nodes increases, bringing many disadvantages. Others constraint handlers are started such as `benderslp`, `integral`, `benders`, `countsols` (see for example Table A.13 on page 102 of Appendix A Additional Data). Go to Appendix B for detailled descriptions.
In PingPong1000, being primal heuristics insufficient to guarantee an efficient resolution, branching process is always activated, while in all other models it is present only when heuristics are deactivated, or their use is limited, but we will focus on this aspect in the next section. In particular, `relpcost` is used. This branching rule uses the notion of pseudo costs to measure the expected gain in the dual bound when branching on a particular variable. In the other cases there are no branching operations as the solution is easily found by heuristics.
Setting `heuristics emphasis fast` generates an increment in the operations of propagators, as shown in the following Table 6.12 on page 65. Go to Appendix B for detailed descriptions.

Primal Heuristics	P=H=S=d		P=S=d, H=a		P=S=d, H=f			
	PPx	PP1000	PPx	PP1000	PP10	P100	P200	PP1000
adaptivediving		1C		1C				6C, 1F, 1B
alns		5C		20C				
bound			1C	50C				
coefdiving		1C		1C				
conflictdiving				1C				4C
feaspump		1C		1C				
fracdiving		1C						
gins		3C		3C				
intshifting		3C		2C	1C, 1F, 1B	1C, 1F, 1B	1C, 1F, 1B	11C, 1F, 1B
locks	1C, 1F, 1B	1C, 1F, 1B	1C, 1F, 1B	50C, 1F, 1B				
mutation				3C				
oneopt	1C	1C	1C	1C	3C	3C	3C	3C
pscostdiving		1C		1C				
randrounding		50C		100C	9C	13C	22C	62C
rens		1C		20C				
rins		6C		6C				
rounding		389C		417C	11C	104C	156C	402C
shiftandpropagate					1C, 1F, 1B	1C, 1F, 1B	1C, 1F, 1B	
shifting		99C		167C		19C	1C, 1F, 1B	105C
trivial	2C	2C	2C	51C	2C	2C	2C	2C
trustregion				1C				
twoopt				1C				
zeroobj			1C, 1F, 1B	1C, 1F, 1B				
zirounding		984C		992C	3C, 1F, 1B	100C, 1F, 1B	200C, 1F, 1B	992C

Table 6.11: Heuristics behavior switching setting for all heuristics

Propagators	P=H=S=d				P=S=d, H=f			
	PP10	PP100	PP200	PP1000	PP10	PP100	PP200	PP1000
dualfix	propagate 10	propagate 11	propagate 11	propagate 11	propagate 19	propagate 19	propagate 19	propagate 19
pseudoobj	propagate 9, Dom-Reds 39	propagate 10, Dom-Reds 200	propagate 10, Dom-Reds 400	propagate 6968,cut-offs 6, domreds 1239580	propagate 41, Dom-Reds 41	propagate 340, Dom-reds 400	propagate 644, Dom-reds 800	propagate 5440, cut-offs 804, domreds 6000
redcost				propagate 1968	propagate 15	propagate 210	propagate 413	propagate 1997
rootredcost				propagate 1, Dom-Reds 8	propagate 2, Dom-Reds 20	propagate 2, Dom-Reds 200	propagate 2, Dom-Reds 400	propagate 2. Dom-Reds 2000
vbounds					propagate 1			

Table 6.12: Propagators increment setting heuristics emphasis fast

Switching the setting for primal heuristics also partially influence presolving process. In fact, while setting `heuristics emphasis off` and `heuristics emphasis fast`, there are no differences regarding presolvers, but a slight advantage is acquired in the case in which we strengthen the use of heuristics with `heuristics emphasis aggressive`, in which the presolved problem presents one less variable. On Table 6.13 it's possible to see the increment of presolvers while setting `heuristics emphasis aggressive`, compared with default case.

Presolvers	P=H=S=d	P=S=d, H=a		
	PPx	PP10	PP100/PP200	PP1000
domcol	1	1	1	
dualagg				**1**
dualcomp	1	1	1	1
dualinfer				**1**
dualsparsify	1	1	1	1
implics	**1**	1	**2**	1
inttobinary				
milp	1	1	1	1
sparsify	1	1	1	1
trivial	**1**	**10**	**3**	**2**
dualfix	**1**	**10**	**3**	**2**
probing	1	1	1	1
pseudobj		1	1	1
symmetry	1	1	1	1
linear	**2**	**11**	**4**	**3**
components	1	1	1	1

Table 6.13: Presolving increment setting heuristics emphasis aggressive

In particular there is an increment of `dualfix, trivial, linear, implics` presolvers. Multiple rounds of presolving are performed, 1 variable is eliminated, and an admissible solution is found immediately by the `zeroobj`. Separators are not used as branching process is not executed.

In PingPong200 we have also some tests performed to understand to which of all presolvers this improvement is due to. Some presolvers are disabled one by one and results are compared with the case `heuristics emphasis aggressive` in the following way:

```
set heuristics emphasis aggressive
set presolving <presolver> maxrounds 0.
```

Unfortunately, through this test we can't find interesting results and furthermore with SCIP it's impossible to deactivate `linear, dualfix, pseudobj` presolvers.

In default case and setting `heuristics emphasis aggressive`, optimal solution is found efficiently by `locks` heuristic. On the other hand in `heuristics emphasis fast`, in which `locks` heuristic is disabled, and disabling all heuristics, resolution is less efficient. The greater impact can be found in the B&B tree, by looking at how the values of parameters change consequently to the switch of the setting of heuristics, as you can see on Table 6.14 on page 67.

B&B tree	P=S=d, H=off				P=S=d, H=a				P=S=d, H=f			
	PP10	PP100	P200	PP1000	PP10	PP100	P200	PP1000	PP10	PP100	PP200	PP1000
Number of runs	1	1	1	1	1	1	1	1	1	1	1	1
Nodes	**5**	**102**	**202**	**1699**	**1**	**1**	**1**	**1985**	**7**	**201**	**401**	**1799**
Feasible leaves	**1**	**1**	**1**	**267**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Infeas.leaves	**1**	**1**	**1**	**625**	**0**	**0**	**0**	**992**	**3**	**99**	**199**	**805**
Objective leaves	**0**	**0**	**0**	**107**	**0**	**0**	**0**	**1**	**1**	**2**	**2**	**2**
Nodes (total)	**5**	**102**	**202**	**1999**	**1**	**1**	**1**	**1985**	**7**	**201**	**401**	**1799**
Nodes left	0	0	0	0	0	0	0	0	0	0	0	0
Max depth	**3**	**100**	**200**	**1000**	**0**	**0**	**0**	**992**	**3**	**100**	**200**	**992**
Backtracks	**3**	**99**	**200**	**844**	**0**	**0**	**0**	**418**	**3**	**98**	**199**	**1000**
Early backtracks	0	0	0	0	0	0	0	0	0	0	0	0
Nodes exc Ref	0	0	0	0	0	0	0	0	0	0	0	0
Delayed Cutoffs	**2**	**99**	**199**	**2**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**186**
Repropagations	**3**	**99**	**200**	**24972**	**0**	**0**	**0**	**1245**	**0**	**1**	**1**	**1792**
A.SwitchLength	**3.60**	**5.86**	**5.94**	**51.36**	**2.00**	**2.00**	**2.00**	**34.54**	**3.57**	**3.96**	**2.99**	**379.17**
Solving time	0.00	0.00	0.00	**11.00**	0.00	0.00	0.00	**21.00**	0.00	0.00	0.00	**4.00**

Table 6.14: Branch and Bound tree switching setting for all heuristics

For models with less than 1000 variables, there is no difference between the default case and `heuristics emphasis aggressive` case. Meanwhile, by imposing `heuristics emphasis off/fast`, the number of nodes increases, even in PingPong1000. That is, as we wanted to demonstrate, considering primal heuristics isolated than all the rest can lead to errors of evaluation. Disabling all separators, setting `heuristics emphasis fast`, appeared as the probable best configuration. On the contrary, in this case, although solution time lowers to 4.00s, there is a worsening of B&B tree, in which nodes reach the number of 1799 against the 1611 of default case. Furthermore, for PingPong1000, B&B tree size increase not only setting `heuristics emphasis fast/off`, but also with `heuristics emphasis aggressive`, with 1985 nodes. This clearly represent a disadvantage in the resolution as what we want is a branching tree that is as small as possible.

In order to understand which heuristic has a particular influence on the resolution, some tests are performed deactivating heuristics individually to study their behavior in detail. Except for `heuristics emphasis off/fast` cases, the final solution is always found by `locks` heuristic, that fixes variables based on their rounding locks, so that the attention is focused on it in the following tests.
We disable primal heuristics using the following command, and then analyzing the results through statistics.

```
SCIP > set heuristics <name of a heuristic > freq -1
```

In particular, in addiction to `locks`,in PingPong and PingPong10, we do this with `intshifting`, `shiftandpropagate`, `zeroobj`, `zirounding`, `oneopt`, `trivial`[4] in order to compare results with or without clustering. `Intshifting` tries to recover from intermediate infeasibilities, shifts integer variables, and solves a final LP to calculate feasible values for continuous variables. `Shiftandpropagate` is a pre-root heuristic to expand an auxiliary branch-and-bound tree and apply propagation techniques, while `zeroobj` is a pre-root heuristic to expand an auxiliary branch-and-bound tree and apply propagation techniques. `Zirounding` takes row slacks and bounds into account. This analysis are made because they are the heuristics which found a feasible solutions in the previous cases. We test `oneopt` and `trivial` too, that respectively try to improve setting of single integer variables and try some trivial solutions,because even if they do not find the solution, they are used in the default case and their absence could lead to slowdowns or a reduction in efficiency.

As expected, emerges that by deactivating the first mentioned group of heuristics the behavior turns out to be exactly the same as the default case and the solution is found by the `locks` heuristic, and the same is when we deactivate `oneopt`. When `trivial` is disabled, the resolution of the problem, by `locks`, requires less time and there are some differences in values of constraint handlers but it doesn't bring particularly important differences in the final solution.
Obviously more important results are obtained when `locks` heuristic is deactivated, tested in all models. On table 6.15 on page 69 you can see which heuristics come in to play disabling locks in default setting, in each model.

[4] see Heuristics File References on Bibliograpgy

Primal Heuristics	P=H=S=d		P=H=S=d,locks freq -1		
	PPx	PP1000	PP10/100	PP200	PP1000
adaptivediving		1C			1C
alns		5C	1C	1 C	5C
bound					
coefdiving		**1C**			
crossover					
conflictdiving		1C			1C
distributiondivin		1C			1C
feaspump		**1C**	**1C,1F,1B**	**1C,1F,1B**	**1C,1F,1B**
fracdiving		1C			1C
gins		**3C**			**2C**
intshifting		**3C**	**1C,1F,1B**	**1C,1F,1B**	**10C,1F,1B**
linesearchdiving					**1C**
locks	**1C,1F,1B**	**1C,1F,1B**			
mutation					
objcostdiving					**1C**
oneopt	1C	**1C**	1C	1C	**2C**
pscostdiving		1C			1C
randrounding		**50C**	**7C**	**11C**	**61C**
rens		1C	**1C**	**1C**	1C
rins		**6C**			**5C**
rounding		**389C**	**7C**	**11C**	**395C**
shiftandpropagate			**1C,1F,1B**	**1C,1F,1B**	**1C,1F,1B**
shifting		**99C**	**7C**	**11C**	**100C**
trivial	2C	2C	2C	2C	2C
trustregion					
twoopt					
zeroobj					
zirounding		**984C**	**1C**	**1C**	**970C,1F**

Table 6.15: Heuristics behavior disabling locks on each model, compared with results obtained running the program in default setting

In all cases the resolution of the problem proceeds more slowly, more propagators must be put into operation, as well as constraint handlers. In models with less than 1000 variables, branching process, which was not used in default setting, is started with the consequent use of separators, while in PingPong1000 their use is strenthened. In particular the separators used are `cut pool`, `aggregation`, `clique`, `gomory`, `impliedbounds`, `mcf`, `mixing` (for detailed informations go to Appendix B).
One more time from the B&B tree a great difference in PingPong1000 respect to all others models is found. In fact, while in the other models, although the lack of `locks` brought a decrease in efficiency, the branching tree remained unchanged, in PingPong1000 we find a different size. However, there is not a worsening of the tree as one would have expected, but rather an improvement, in fact the number of nodes decreases until 1121, for an improvement of about 500 of them. This demonstrates even more how this model has anomalous behavior compared to all the others. Nevertheless, note on Table 6.16, that there is a worsening of solving time of 2.00s.
That is, since the exploration of each node requires the use of a resolution algorithm, for example the Simplex Algorithm, the analysis time of each of them is not deterministic, but always varies. There are therefore cases like this in which, although the nodes explored are fewer, the total resolution time is higher. In this case the

choice depends on what we believe to be most important for the resolution of our specific problem. In most cases we associated the size of the tree with the yardstick to decide the best configuration, especially in models with few variables. However, it is essential to keep in mind that what we are looking for is always efficiency, inevitably linked to the solution time, since at the level of effectiveness, all possible configurations lead to the optimal solution.

In the following table you can see in detail the difference brougth to B&B tree disabling `locks` in PingPong1000, compared to that of default case.

B&B tree	P=H=S=d	P=H=S=d,locks freq -1
number of runs	1	1
feasible leaves	0	0
infeas.leaves	**626**	**150**
objective leaves	1	1
total nodes	**1611**	**1121**
max depth	**984**	**970**
backtracks	**417**	**978**
delayed cutoffs	**358**	**820**
repropagations	**3692**	**4805**
avg switch length	**8.42**	**27.96**
switching time	**1.00**	**0.00**
Solving time	**5.00**	**7.00**

Table 6.16: Branch and Bound tree improvement disabling locks heuristic in PingPong1000

Other heuristics like `feaspump`, `inshifting`, `shiftandpropagate` come in to play (see Table 6.15) improving the solution three times and, in particular, the best solution is found by `feaspump` heuristic.
In PingPong100, we perform further tests on single heuristics in order to understand which of them have more influence on the resolution, but setting `heuristics emphasis aggressive/fast`. That is, setting `heuristics emphasis aggressive` the solution was improved two times by `zeroobj` and `locks` heuristics. So, the following test are made comparing the results with the case in which we only impose `heuristics emphasis aggressive`:

```
set heuristics emphasis aggressive
set heuristics locks/zeroobj freq -1
```

Disabling `locks` results are similar to those of the previous test. The objective value is higher and also in this case branching process and separators are activated. Solution is improved three times by `feaspump`, `zeroobj` and `intshifting` heuristics (see Table 6.17)

Primal Heuristics	P=S=d, H=a, locks freq -1
alns	1C
feaspump	1C, 1F, 1B
intshifting	1C, 1F, 1B
proximity	1C
randrounding	7C
rens	1C
repair	1C
rounding	7C
shifting	7C
trivial	2C
twopt	1C
shifting	7C
zeroobj	1C, 1F, 1B
zirounding	1C

Table 6.17: PingPong100.lp heuristics behavior with heuristics emphasis aggressive and heuristics locks freq -1.

Disabling `zeroobj`, during the presolving process no variables are delated from the original model, this means that `zeroobj` has an important influence in the presolving process. The objective value is higher but it is found with only one improvement by `locks` heuristic (see solution values on table **??** on page **??** of Appendix A)

When we impose `heuristics emphasis fast` solution was improved three times by `intshifting, oneopt, zirounding` heuristics. Disabling `intshifting, shiftandpropagate` we don't find particular results, while interesting is the case in which we disable `zirounding`. In particular, important results are obtained on the B&B tree, as it can be seen in the following table.

B&B tree	P=S=d, H=f	P=S=d, H=f, zirounding freq -1
Number of runs	1	1
Nodes	**201**	**103**
Feasible leaves	**0**	**1**
Infeas.leaves	**99**	**1**
Objective leaves	**2**	**1**
Nodes (total)	**201**	**103**
Nodes left	0	0
Max depth	100	100
Max depth (total)	100	100
Backtracks	98	98
Early backtracks	0	0
Nodes exc. Ref.	0	0
Delayed cutoffs	**0**	**98**
Repropagations	**1**	**99**
Avg switch length	**3.96**	**5.81**
Switching time	0.00	0.00

Table 6.18: PingPong100.lp branch tree with heuristics emphasis fast and zirounding freq -1

The number of nodes decreases decisively, this means that in some way, `zirounding` heuristic makes a worsening in solving the problem.

Performing a different type of tests on `locks` in all models interesting results are found. In detail, the aim is to intensify its use in those cases that turned out to be the worst, in order to evaluate how it can really make the difference.

```
set heuristics emphasis off/fast
  set heuristics locks freq 1
```

You can see results of this analysis on B&B tree on Table 6.19 on page 73. On the first column there are all branch and bound tree parameters analyzed, while on the following two multicolumns the results obtained respectively when we disable heuristics and when we set `heuristics emphasis fast`, on each model. These results are compared to the cases in which we also strengthen the use of `locks` heuristics, on the last two multicolumns. Note that in these last cases, the same tree for models PingPong10, PingPong100, PingPong200 is obtained, so PPx : $x \in \{10, 100, 200\}$.
With the exception of PingPong1000, activating `locks` heuristic, there's again only 1 B&B node, less constraint handlers,propagator and total absence of separators, cutselectors and branching operations. The B&B tree comes back to the default form. Furthermore, this seems to be until more efficient than the default case as less constraint handlers and heuristics are used.
Clearly the best case remains the case in which only `locks` heuristics is active because setting `heuristics emphasis fast` there is a waste of time calling others heuristics that don't have an important impact on the resolution.
Totally different is the result given by PingPong1000. In fact, setting on `locks` heuristic, despite a small improvement in B&B tree, in both case we have an important worsening in solving time.

Given the inconsistency between the previous results given by PingPong1000 model and all the others, it is not possible to conclude in advance on the efficiency of `locks` heuristic. In order to find the best heuristic, it is necessary to test all the others one by one, then comparing the results in terms of B&B tree and times. Particular attention is given to PingPong1000 model, where having more variables, the changes are more evident, especially in terms of time.
In particular, referring to Tables 6.15,6.11, it is interesting to test those heuristics which come into play finding feasible solution in absence of `locks`, such as `feaspump`, `intshifting`, `shiftandpropagate`, `zirounding`, one to one, isolating them from all others heuristics. The mechanism of the test is always the same:

- ```
  set presolving default
  set heuristics emphasis off
  set heuristics <heuristic> freq 1
  set separating default
  ```

On Table 6.20 on page 74 you can see the results on B&B tree for each model.
Regarding models under 1000 variables, best results are obtained setting on `locks` and `feaspump` but solving time is the same between these two cases so it's not possible to establish the best one. On contrary, watching to solving times, it's clear how `feaspump` is more efficient than `locks`. Despite B&B tree for both cases is the same, setting on `feaspump` solving time decrease until 3.00s, against the 22.00 seconds of the other case.
  ```

B&B tree	P=S=d, H=off				P=S=d, H=f				P=S=d, H=off, locks freq 1		P=S=d, H=f, locks freq 1	
	PP10	PP100	P200	PP1000	PP10	PP100	P200	PP1000	PPx	PP1000	PPx	PP1000
Number of runs	1	1	1	1	1	1	1	1	1	1	1	1
Nodes	**5**	**102**	**202**	**1999**	**7**	**201**	**401**	**1799**	**1**	**1983**	**1**	**1625**
Feasible leaves	**1**	**1**	**1**	**267**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Infeas.leaves	1	1	1	625	3	99	199	805	0	991	0	633
Objective leaves	**0**	**0**	**0**	**107**	**1**	**2**	**2**	**2**	**0**	**1**	**0**	**1**
Nodes (total)	**5**	**102**	**202**	**1999**	**7**	**201**	**401**	**1799**	**1**	**1983**	**1**	**1625**
Nodes left	0	0	0	0	0	0	0	0	0	0	0	0
Max depth	**3**	**100**	**200**	**1000**	**3**	**100**	**200**	**992**	**0**	**991**	**0**	**991**
Backtracks	**3**	**99**	**200**	**844**	**3**	**98**	**199**	**1000**	**0**	**419**	**0**	**417**
Early backtracks	0	0	0	0	0	0	0	0	0	0	0	0
Nodes exc Ref	0	0	0	0	0	0	0	0	0	0	0	0
Delayed Cutoffs	**2**	**99**	**199**	**2**	**0**	**0**	**0**	**186**	**0**	**0**	**0**	**358**
Repropagations	**3**	**99**	**200**	**24972**	**0**	**1**	**1**	**1792**	**0**	**3187**	**0**	**2831**
A.SwitchLength	**3.60**	**5.86**	**5.94**	**51.36**	**3.57**	**3.96**	**2.99**	**379.17**	**2.00**	**15.27**	**2.00**	**7.20**
Solving time	0.00	0.00	0.00	**11.00**	0.00	0.00	0.00	**4.00**	0.00	**24.00**	0.00	**22.00**

Table 6.19: Branch and Bound tree improvement given by locks setting heuristics emphasis off and heuristics emphasis fast, for each model.

B&B tree	P=S=d,H=off, feaspump freq 1		P=S=d, H=off, intshifting freq 1				P=S=d,H=off. shiftandpropagate freq 1				P=S=d, H=off, zirounding freq 1			
	PPx	PP1000	PP10	PP100	PP200	PP1000	PP10	PP100	PP200	PP1000	PP10	PP100	PP200	PP1000
Number of runs	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Nodes	**1**	**1983**	**4**	**102**	**202**	**2001**	**6**	**103**	**203**	**2001**	**7**	**201**	**401**	**1992**
Feasible leaves	**0**	**0**	**1**	**1**	**1**	**294**	**1**	**2**	**2**	**36**	**0**	**0**	**0**	**266**
Infeas.leaves	**0**	**991**	**1**	**1**	**1**	**489**	**1**	**1**	**1**	**956**	**3**	**100**	**200**	**618**
Objective leaves	**0**	**1**	**0**	**0**	**0**	**218**	**0**	**0**	**0**	**9**	**1**	**1**	**1**	**108**
Nodes (total)	**1**	**1983**	**4**	**102**	**202**	**2001**	**6**	**103**	**203**	**2001**	**7**	**201**	**401**	**1992**
Nodes left	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Max depth	**0**	**991**	**2**	**100**	**200**	**1000**	**4**	**100**	**200**	**1000**	**3**	**100**	**200**	**1000**
Max depth (total)	**0**	**991**	**2**	**100**	**200**	**1000**	**4**	**100**	**200**	**1000**	**3**	**100**	**200**	**1000**
Backtracks	**0**	**419**	**2**	**99**	**200**	**708**	**4**	**98**	**199**	**999**	**3**	**99**	**200**	**844**
Early backtracks	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Nodes exc. Ref.	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Delayed cutoffs	**0**	**0**	**1**	**99**	**199**	**0**	**3**	**98**	**198**	**2**	**0**	**0**	**0**	**9**
Repropagations	**0**	**3187**	**2**	**99**	**200**	**1158**	**4**	**100**	**201**	**1954**	**1**	**1**	**1**	**20733**
Avg switch length	**2.00**	**15.27**	**3.00**	**5.86**	**5.94**	**111.95**	**4.00**	**6.66**	**6.34**	**499.57**	**2.57**	**3.96**	**2.99**	**53.35**
Switching time	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Solving time	0.00	**3.00**	0.00	0.00	0.00	**11.00**	0.00	0.00	0.00	**2.00**	0.00	0.00	0.00	**11.00**

Table 6.20: Branch and Bound tree data setting on heuristics one to one

In order to confirm this result, it is necessary to test these two heuristics in competition. In fact, until now, the best case, especially referring to PingPong1000, is the default one. However, it is important to note that when the heuristics are set to default, `feaspump` heuristic has no priority over `locks`. In detail `locks` has a priority of 3000 with a frequency of 0, while `feaspump` has priority and frequency set respectively to -10000 and 20. Consequently, there have previously been no cases in which they were found to operate at the same conditions.
We verify this performing the following test, for PingPong1000 model:

- ```
  set presolving default
  set heuristics emphasis off
  set heuristics locks freq 0
  set heuristics feaspump freq 1
  set heuristics feaspump advanced priority 3000
  set separating default
  ```

One more time solution is found by `locks` heuristic so we can't confirm the previous results about `feaspump` efficiency. Furthermore, running the following test:

- ```
  set presolving default
  set heuristics emphasis off
  set heuristics locks freq 0
  set separating default
  ```

we obtain the following results on Table 6.21:

B&B tree	P=H=d, locks freq 0	P=H=d, feaspump freq 0
number of runs	1	1
feasible leaves	0	0
infeas.leaves	991	991
objective leaves	1	1
total nodes	1983	1983
max depth	991	991
backtracks	419	419
delayed cutoffs	0	0
repropagations	3187	3187
avg switch length	15.27	15.27
switching time	1.00	0.00
Solving time	3.00	3.00

Table 6.21: Branch and Bound comparison activating locks and feaspump

Separators

With exception of PingPong1000, the use of separators occurs only in some cases. In particular, the separators, fundamental in the branching process, are not used in the default case as the solution is found directly by `locks` heuristic, nor when we modify the presolving setting. They act if the heuristics are deactivated, or if they are insufficient, i.e. all those times in which the branching process is activated. For this reason a more in-depth study of these parameters is carried out, trying to understand how they act during the resolution process. We therefore suppose to change the setting of the separators in those cases in which they participate in the resolution of the model.
On Table 6.22, you can check which separators are used for each model, in these mentioned cases. In detail, the values on the table stay for the number of calls effectuated for each separator, for models with less then 1000 variables, for which we perform a different analysis.

Separators	P=S=d, H=off			P=S=d, H=f			P=S=H=d, locks freq -1		
	PP10	PP100	P200	PP10	PP100	P200	PP10	PP100	PP200
cut pool	**15**	**28**	**38**	**16**	**27**	**43**	**13**	**13**	**21**
aggregation	**8**	**12**	**13**	**8**	**11**	**15**	**7**	**7**	**11**
cgmip				**1**					
clique	1	1	1	1	1	1	1	1	1
eccuts				**1**					
gomory	**8**	**12**	**13**	**8**	**11**	**13**	**7**	**7**	**10**
impliedbounds	**8**	**12**	**13**	**8**	**11**	**15**	**7**	**7**	**11**
mcf	1	1	1	1	1	1	1	1	1
mixing	**8**	**12**	**12**	**8**	**11**	**15**	**7**	**7**	**11**
rlt	**8**	**8**	**8**	**8**	**8**	**10**	**7**	**7**	**10**
zerohalf	**8**	**12**	**13**	**8**	**11**	**15**	**7**	**7**	**11**

Table 6.22: Separators behavior switching setting for heuristics

As in the previous cases, there is an analysis of the behavior of the separators by switching their setting using the `emphasis` command. In particular, in models with less of 1000 variables, we perform the analysis switching the setting of the separators in each of those cases in which they are activated. After that, for those cases which had given particular results, there is a deeper analysis performed by deactivating individual separators, to understand which of them can have a more important role in solving the problem.
We made this in the following way:

```
set heuristics /emphasis off/emphasis fast/locks freq -1
set separating emphasis off/ aggressive/ fast
```

running the following tests for models PingPong, PingPong10, PingPong100, PingPong200 :

1. When we impose `heuristics emphasis off` branching process is activated and consequentially also the use of separators. With this test we want to investigate how actually the separators influence the branching process when

we deactivate heuristics, changing their setting through `emphasis` command, in the following way:

```
set presolving default
set heuristics emphasis off
set separating emphasis default/off/ aggressive/ fast
```

Referring also to Table A.5 on page 98, note how these operation have greater

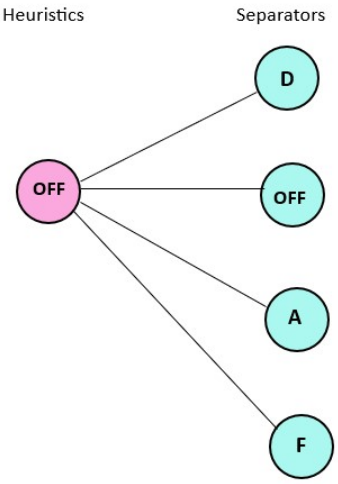


Figure 6.3: Configurations given by setting presolving default, heuristics off, switching the setting for separators.

impact only on the simplest model PingPong10, as you can see on the following Table 6.23

B&B tree	PingPong10			
	P=S=d, H=off	P=d, H=S=off	P=d, H=off,S=a	P=d, H=off,S=f
Number of runs	1	1	1	1
Nodes	**5**	**12**	**5**	**7**
Feasible leaves	1	1	1	1
Infeas.leaves	1	1	1	1
Objective leaves	0	0	0	0
Nodes (total)	**5**	**12**	**5**	**7**
Nodes left	0	0	0	0
Max depth	**3**	**10**	**3**	**5**
Max depth (total)	**3**	**10**	**3**	**5**
Backtracks	**3**	**9**	**3**	**5**
Early backtracks	0	0	0	0
Nodes exc. Ref.	0	0	0	0
Delayed cutoffs	**2**	**9**	**2**	**4**
Repropagations	**3**	**9**	**3**	**5**
Avg switch length	**3.60**	**4.83**	**3.60**	**4.29**
Switching time	0.00	0.00	0.00	0.00

Table 6.23: Ping-Pong.lp B&B tree changing the setting of separators when disabling heuristics

While in the first model the disadvantage is evident above all from the B&B tree as number of nodes is higher, in PingPong100 and PingPong200 the tree remains unchanged (Table A.5). As regards the solution, there are not differences and the final bound if found by `relaxations` in all cases.
However, for PingPong10, watching to the structure of the tree, where the number of nodes increases, this is sufficient to be able to state that deactivating the separators brings disadvantages to the resolution of the model, while for the others it might even be an advantage, ass without them less operations are performed.
When heuristics are deactivated and we strengthen the use of separators with `separating emphasis aggressive` (Table A.5), in all cases B&B tree remains exactly the same as the case in which we only deactivate the heuristics, forcing the program to perform more operations unnecessarily, without providing any advantage.

2. As in the previous case, because also imposing `heuristics emphasis fast` separators are active, let's repeat the last analysis in this case too:

```
set presolving default
set heuristics emphasis fast
set separating emphasis default/off/ aggressive/ fast
```

On Table A.6 on page 99 of Appendix A you can see the results on B&B

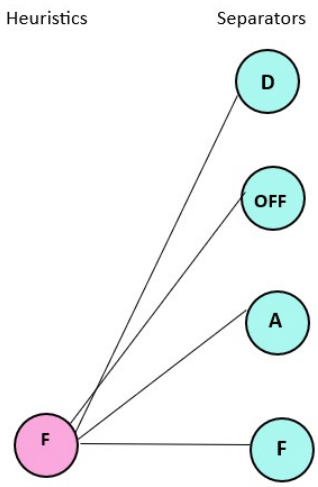


Figure 6.4: Configurations given by setting presolving default, heuristics fast, switching the setting for separators.

tree for each model.
When imposing `heuristics emphasis fast` and deactivating the separators or seting `separating emphasis fast`, like before, on PingPong10 there is an evident disadvantage seeing the B&B tree where in this case we have even 21 nodes. On the other hand we don't find particular differences on the others models as values are quite the same.
Interesting is the case in which we impose `separating emphasis aggressive`.

In fact, in PingPong10, and most of all in PingPong200 we have a clear improvement, so in this case changing the separator setting proves to be very useful. Strange is the case of PingPong100 where the tree remains almost unchanged.

B&B tree	P=S=d, H=f			P=s,H=f,S=a		
	PP10	PP100	P200	PP10	PP100	P200
Number of runs	1	1	1	1	1	1
Nodes	**7**	201	**401**	**1**	201	**1**
Feasible leaves	0	0	0	0	0	0
Infeas.leaves	**3**	99	**199**	**0**	99	**0**
Objective leaves	**1**	2	**2**	**0**	2	**1**
Nodes (total)	**7**	201	**401**	**1**	201	**2**
Nodes left	0	0	0	0	0	0
Max depth	**3**	100	**200**	**0**	100	**0**
Max depth (total)	**3**	100	**200**	**0**	100	**0**
Backtracks	**3**	98	**199**	**0**	98	**0**
Early backtracks	0	0	0	0	0	0
Nodes exc. Ref.	0	0	0	0	0	0
Delayed cutoffs	0	0	0	0	0	0
Repropagations	**1**	1	**1**	**0**	1	**0**
Avg switch length	**2.57**	**2.99**	**3.96**	**2.00**	**3.96**	**2.00**
Switching time	0.00	0.00	0.00	0.00	0.00	0.00

Table 6.24: Branch and Bound tree improvement strengthening separators when setting heuristics emphasis fast

As can be verified on Table 6.24, strengthening separators in PingPong and PingPong10, B&B tree come back to default form, but the improvement is especially noticeable in PingPong200, in which the operation has a great impact on presolving process too. In fact, resolution time increase a lot, but, restarting after 6 global fixings of integer variables, more cycles of presolving are effectuated obtaining 19 delated vars, 25 delated constraints, 15 tightened bounds, 1 implication. After that in the B&B tree only 1 solving node is obtained, against the 401 nodes of the case `heuristics emphasis fast`.
This makes me think that there is a highly useful separator that is used in this last case but not in the case in which the separators are left in a default situation. In order to study in more detail the separators that bring this improvement to the model, referring to Table A.7 on page 96 of Appendix A, we try to disable one by one those separators which increased setting `separating emphasis aggressive`, in the following way:

```
set heuristics emphasis fast set separating emphasis
aggressive set separating <name of a separator> freq -1
```

In particular, for PingPong10, this test is performed with `aggregation, closecuts, gomory, implied bounds, mixing, rlt, zerohalf` separators, while with PingPong200 we add `clique` and `mcf` too. It's impossible with SCIP to deactivate `cut pool` separator.
In all cases, when `closecuts,gomory, impliedbounds,mixing,rlt,zerohalf` are deactivated, B&B tree results

unchanged, so this means that they are not important in the resolution of the problem. On the contrary, whithout `aggregation`, the number of nodes increase significantly (see Table 6.25).This means that maybe a lack of `aggregation` has a certain impact in the resolution, since when it is disabled the efficiency of the problem decreases.

B&B tree	P=d,H=f,S=a		P=d,H=f,S=a, aggr. freq -1	
	PP10	P200	PP10	P200
Number of runs	1	**2**	1	**1**
Nodes	**1**	**1**	**19**	**401**
Feasible leaves	0	0	0	0
Infeas.leaves	**0**	**0**	**9**	**200**
Objective leaves	**0**	1	**1**	1
Nodes (total)	**1**	**2**	**19**	**401**
Nodes left	0	0	0	0
Max depth	**0**	**0**	**9**	**200**
Max depth (total)	**0**	**0**	**9**	**200**
Backtracks	**0**	**0**	**9**	**200**
Early backtracks	0	0	0	0
Nodes exc. Ref.	0	0	0	0
Delayed cutoffs	0	0	0	0
Repropagations	**0**	**0**	**1**	**1**
Avg switch length	**2.00**	**2.00**	**2.84**	**2.99**
Switching time	0.00	0.00	0.00	0.00

Table 6.25: Branch and Bound tree worsening disabling aggregation separator

3. We have the same analysis of separators behavior also in case in which `locks` heuristic is disabled:

```
set heuristics locks freq -1
set separating emphasis off/ aggressive/ fast
```

In all cases, through this test we don't find particular results. There are no differences in B&B tree and solution values.

On table 6.26, youcan see how separators operate for each setting of heuristics in PingPong1000, in terms of number of calls.

Separators	P=H=S=d	P=S=d, H=off	P=S=d, H=a	P=S=d, H=f
cut pool		**143**		**123**
aggregation	**5**	**14**	**5**	**16**
clique	1	1	1	1
gomory	**5**	**14**	**5**	**14**
impliedbounds	**5**	**14**	**5**	**16**
mcf	1	1	1	1
mixing	**5**	**16**	**5**	**16**
rlt	**1**	**8**	**1**	**10**
zerohalf	**5**	**14**	**5**	**16**

Table 6.26: PingPong1000.lp separators behavior switching the setting for all heuristics

Presenting PingPong1000 a different behavior then all others ones, we carry out the separators analysis for this model separately, in a different way.
As in this case branching process is executed also running in default setting, we try to switch the setting for all separators to see how they influence the resolution of the model. How separators operate for each setting is reported on Table A.8 on page 100 of Appendix A.
Disabling separators or setting `separating emphasis fast` there are no particu-

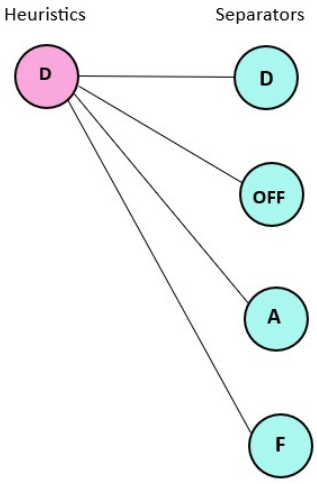


Figure 6.5: Configurations given by setting presolving default, heuristics default, switching the setting for separators.

lar differences respect to default result. When use of separators is increased, although solution time is higher, of 7.00 seconds, there's an improvement in the number of solving nodes, 1589 against the 1611 nodes of default case, while primal and dual bounds are the same. In the following table you can see how B&B tree is improved strengthening separators, compared with that of default case.

B&B tree	P=H=S=d	P=H=d, S=a
number of runs	1	1
nodes	**1611**	**1589**
feasible leaves	0	0
infeas.leaves	**626**	**605**
objective leaves	1	1
nodes (total)	**1611**	**1589**
nodes left	0	0
max depth	**984**	**983**
max depth (total)	**984**	**983**
backtracks	**417**	**435**
early backtracks	0	0
nodes exc ref	0	0
delayed cutoffs	**358**	**378**
repropagations	**3692**	**3353**
avg switch length	**8.42**	**8.59**
switching time	0.00	0.00
Solution time	**5.00**	**7.00**

Table 6.27: PingPong1000.lp branch tree improvement strengthening separating

On Table 6.28 you can find the results on B&B tree given by the last configurations to evaluate, represented in the following figure.

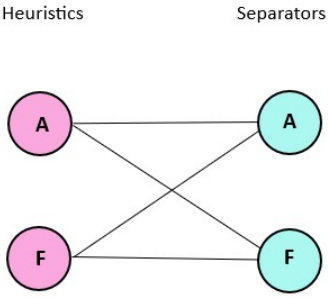


Figure 6.6: Additional Configurations evaluated for PingPong1000.

B&B tree	P=d,H=f,S=a	P=d,H=S=a	P=d,H=a,S=f	P=d,H=S=f
number of runs	1	1	1	1
nodes	**1799**	**1892**	**1985**	**1800**
feasible leaves	0	0	0	0
infeas.leaves	**805**	**899**	**992**	**806**
objective leaves	**2**	**1**	**1**	**2**
nodes (total)	**1799**	**1892**	**1985**	**1800**
nodes left	0	0	0	0
max depth	992	992	992	992
max depth (total)	992	992	992	992
backtracks	**417**	**419**	**418**	**1000**
early backtracks	**1000**	**0**	**0**	**0**
nodes exc ref	0	0	0	0
delayed cutoffs	**186**	**93**	**0**	**185**
repropagations	**1800**	**1775**	**1245**	**1188**
avg switch length	**378.42**	**51.10**	**34.54**	**114.04**
switching time	0.00	0.00	0.00	0.00
Solution time	**4.00**	**22.00**	**22.00**	**4.00**

Table 6.28: PingPong1000.lp branch and bound tree resulting from additional configurations

CHAPTER 7

Conclusions

In this chapter, by summarizing the various configurations analyzed and looking at their results, we reach a conclusion. It is possible to state which is the best configuration and, to confirm this, this latter is tested in a final model with numerous variables where the clear improvement obtained through it is highlighted compared to the results obtained leaving the default setting of SCIP.

Let's now take stock of the situation and draw conclusions from our analysis.
Summing up, what has been done is searching through manual parameter tuning for the best configuration to efficiently solve a particular lot sizing model. In particular, the attention was focused on the tuning of 3 parameters, *presolvers*, *heuristics* and *separators*, as they are those with the greatest impact on the results, in terms of size of the branching tree and solution time.
To do this, 5 instances of our model were tested, two of these with 10 variables, one of which containing the `TotBin` variable, and the others with 100, 200, 1000 variables respectively.
Starting from the search for the best configuration for the presolvers, subsequently, the analysis proceeded with the search for the best configuration for heuristics and separators, taking into account however that they could not be considered alone, working in parallel and in competition.

On the following Table 7.1 you can check the results obtained setting each model to all possible configurations analyzed.
As already mentioned previously, there are several factors to consider in order to establish which is the best one. Mainly the most important factors are the size of the branching tree, and the total resolution time. However, since the resolution time of each node is non-predictable, evaluating these two factors together as one depends on the other, what we want is a resolution that is as efficient as possible, especially in terms of time.
So, based on this, for models under 1000 variables we can say that the best configuration is the number 2 of the table, as it permits a reduction of the total of operations switching off all separators. However, referring to the particular test described on page 31, we can also consider to choose a configuration on which setting on only

`locks` heuristic, without spending time for additional operations with all others one.
Instead, watching to the results given by PingPong1000, the best configuration seems to be the number 14 with the best number of nodes, of 1151, and the best resolution time of 3.00 seconds.

Configuration		PP10		PP100		PP200		PP1000	
		size	time	size	time	size	time	size	time
1	P=H=S=d	1	0.00	1	0.00	1	0.00	1611	5.00
2	P=H=d,S=off	1	0.00	1	0.00	1	0.00	1611	5.00
3	P=H=d,S=a	1	0.00	1	0.00	1	0.00	1589	7.00
4	P=H=d,S=f	1	0.00	1	0.00	1	0.00	1611	6.00
5	P=S=d,H=off	5	0.00	102	0.00	202	0.00	1999	11.00
6	P=d,H=S=off	12	0.00	102	0.00	202	0.00	1996	10.00
7	P=d,H=off,S=a	5	0.00	102	0.00	202	0.00	1999	11.00
8	P=d,H=off,S=f	7	0.00	102	0.00	202	0.00	1999	12.00
9	P=S=d,H=a	1	0.00	1	0.00	1	0.00	1985	21.00
10	P=d,H=a,S=off	1	0.00	1	0.00	1	0.00	1985	22.00
11	P=d,H=S=a	1	0.00	1	0.00	1	0.00	1892	22.00
12	P=d,H=a,S=f	1	0.00	1	0.00	1	0.00	1985	22.00
13	P=S=d,H=f	7	0.00	201	0.00	401	0.00	1799	4.00
14	P=d,H=f,S=off	21	0.00	201	0.00	401	0.00	1151	3.00
15	P=d,H=f,S=a	1	0.00	201	0.00	1	0.00	1799	4.00
16	P=d,H=S=f	11	0.00	201	0.00	401	0.00	1800	4.00

Table 7.1: Results given by all possible configurations in terms of branch and bound tree size and time.

To verify the results obtained in the previous instances, in particular by referring to the PingPong1000 model, now let's verify a more complex model. In this way we can evaluate whether the configuration found, starting from 1000 variables, is constantly effective at increasing levels of computational complexity, or whether when the number of variables increase it loses efficiency or effectiveness. In detail, this test is performed on a model with 2000 variables.
Initially, we run the program with the default setting provided by SCIP, and then we compare these results with those given by which should the best configuration according to the previous analysis, comparing the files via the Kdiff application.

Primal Heuristics	PingPong2000	
adaptivediving	P=H=S=d 1C	P=d,H=f,S=off ET 5.00,4C,1F,1B
alns	ET 16.00,5C	
conflictdiving	ET 2.00,1C	ET 2.00, 5 Calls
feaspump	1C	
fracdiving	ET 2.00,1C	
gins	4C	
intshifting	3C	10C,1F,1B
locks	1C,1F,1B	
oneopt	1C	3C
pscostdiving	ET 2.00, 1C	
randrounding	100C	101C
rens	ET 2.00, 1C	
rins	ET 3.00,6C	
rounding	603C	584C
shiftandpropagate		1C,1F,1B
shifting	159C	200C
trivial	2C	2C
veclendiving	ET 1.00,1C	
zirounding	1000C	1000C

Table 7.2: Comparison between primal heuristics used running PingPong2000 in default configuration by SCIP and the best one

B&B tree	PingPong2000	
	P=H=S=d	P=d,H=f,S=off
Number of runs	1	1
Nodes	**3400**	**2098**
Feasible leaves	0	0
Infeas.leaves	**1405**	**99**
Objective leaves	1	1
Nodes (total)	**3400**	**2098**
Nodes left	0	0
Max depth	**1994**	**1998**
Max depth (total)	**1994**	**1998**
Backtracks	**724**	**2007**
Early backtracks	0	0
Nodes exc. Ref.	0	0
Delayed cutoffs	**589**	**1899**
Repropagations	**3490**	**10562**
Avg switch length	**989.02**	**18.76**
Solving time	**37.00**	**11.00**

Table 7.3: Comparison between results pn B&B tree and solving time running PingPong2000 in default configuration by SCIP and the best one

As expected, the configuration (P=d,H=f,S=off) brings a great improvement on the resolutions with more than 1300 nodes less on the B&B tree and 11.00s of solving time against the 37.00s of the default one. This attests to the validity of the results obtained previously.

Part III

Final Considerations

CHAPTER 8

Running on OMP application

In this chapter you can find brief notes about the results obtained by solving the previous instances with the OMP solver, and become aware of the differences found between the latter's resolution and that provided by SCIP. From this, it's possible to hypothesize possible future developments.

As already explained previously, the motivation that led to the project carried out through SCIP was to be able to provide useful information to then make improvements to the company solver. Indeed, it has been noted that in the context of the model and the instances analyzed, for certain quantities of variables, when OMP resorts to the use of clustering through the integer variable `TotBin` , the same problem is solved efficiently by SCIP without using it.
In fact, we noticed at the beginning of the experimental phase how aggregation for SCIP is the last alternative used, only when presolvers and heuristics are deactivated. When presolving is active, variable `TotBin` is discarded at the beginning, but the model is solved easily thanks to `locks` heuristic. For this reason we then discarded the first model from the following tests, as when the presolving does its job this model is reduced to the PingPong10 model.
In other cases, as showed below, the application developed by OMP prevails over SCIP, even without clustering support, thanks to the use of specific heuristics.
Below, with the support of the statistics provided by the OMP application, the results given by solving some of the instances analyzed via SCIP are shown, and the differences are highlighted.

On Fig. 8.1 you can see our first model PingPong on OMP Optimizer. We will obtain the rest of the instances by changing the range of the buckets between B01 to Bx : x$\in \{100, 200, 1000, 2000\}$. After being generated in their explicit form a series of solving tests are performed for each instance.
In detail, three type of solving tests are runned. Regarding the setting of branching strategies, the first test is performed given priority on binary variables, while the second using the clustering strategy. After that, an additional test is performed setting on `Feasibility Pump Heuristics`. In normal conditions `feaspump` is

```
** Model created from OMP Optimizer
VERSION=9
**
** Author      : DHU
** Date        : 6/5/2011 11:25:58
** Description:
**
SCENARIO=FormulationAndBranching

*/ Uncomment optimization direction
** MINIMIZE
** MAXIMIZE
MIN

** -- SETS

SET=bct : B01 TO B10

** -- SEGMENTATION AND AGGREGATION

** -- RELATIONS

** -- VARIABLES AND CONSTRAINTS

X=Produce.bct(&)=B
X=StockPos.bct(&)=C $1
X=StockNeg.bct(&)=C $10
X=TotBin=I #1

C=TotBin = TotBin = Produce.bct(S&)

C=Stock.bct(1) = StockPos.bct(1) - StockNeg.bct(1) = 10*Produce.bct(1) - 9.99
FOR T=2 TO bct(LAST)
C=Stock.bct(T) = StockPos.bct(T)- StockNeg.bct(T) = 10*Produce.bct(T) - 10 + StockPos.bct(T-1) - StockNeg.bct(T-1)
ENDFOR T



** -- DATA

** -- REPORT

END
```

Figure 8.1: PingPong model on OMP Optimizer

disabled in OMP solver as, by operating within the main thread, it can be cause slowdowns in the program, but in our case it's interesting to test it having found good results using it in our analysis through SCIP.

As can be observed on Fig. 8.2, representing the statistics printed by our solver for the first simple model of ten variables, giving priority to binary variables generate a B&B tree of 11 nodes, definitely worse than the solution given by SCIP thanks to `Locks` heuristic, of only one node. The same situation resulted from models with 100 and 200 buckets, in which respectively 101 and 201 nodes.

On the contrary, totally different results have been obtained for models with more variables, like in our case PingPong1000, PingPong2000. See on Fig. 8.3 the statistics about the results of PingPong1000. In this case OMP solver results to be extremely more efficient than SCIP. We manage to reach 149 nodes for PingPong1000 and, surprisingly, only 67 wheh we have 2000 variables, while with SCIP we haven't been able to go below 1000 nodes. In both cases the solution is found by `LP based objective`, an heuristic that, using the LP-value of the binaries as cost, it look for a MIP solution maximizing it.

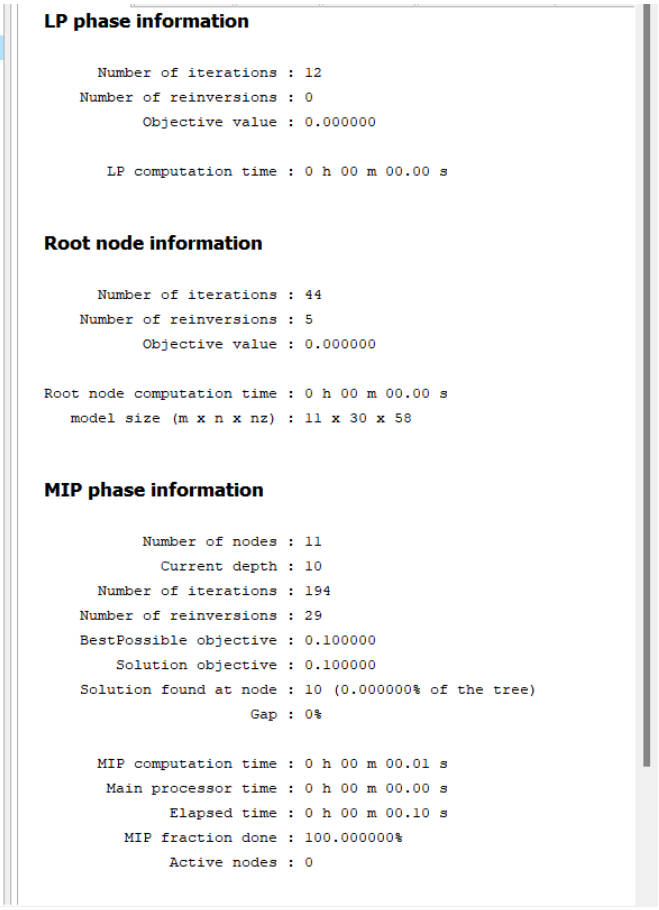


Figure 8.2: PingPong general statistics by OMP Optimizer

Parameter	n = 10	n = 100	n = 200	n = 1000	n = 2000
Number of nodes	11	101	201	149	67
Current depth	10	100	200	0	0
number of iterations	194	3872	8172	7478	8807
Source solution	MIP	MIP	MIP	LP based objective	Smart rounding

Table 8.1: Statistics data given by default setting of OMP solver

Let's go on with aggregation. Modifying the setting of branching strategies and taking away the priority on binary variables, I allow the program to execute the first branching on the integer variable `TotBin`, that is, the problem is solved applying clustering strategy.

See Fig. 8.4 on page 92. For model PingPong, but also for models with 100, 200, 1000, 2000 variables, we reach a Branch and Bound tree with only two nodes. This means, on critical cases such as models with less than 1000 variables, OMP solver manages to get good results compared to SCIP results only making use of aggregation. On the other hand, OMP application results better than SCIP for all other models and it's able to improve further their results thanks to clustering strategy.

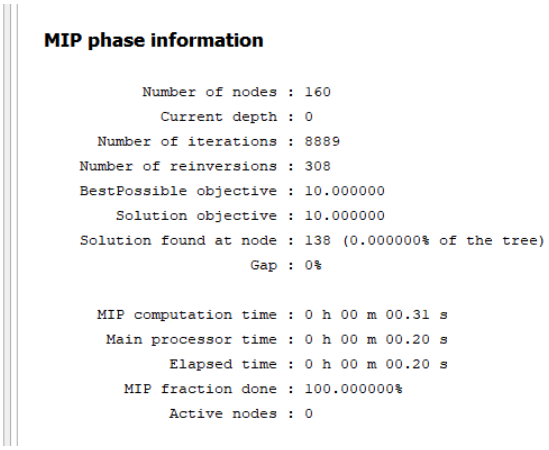


Figure 8.3: PingPong1000 statistics given by OMP solver on branch and Bound tree

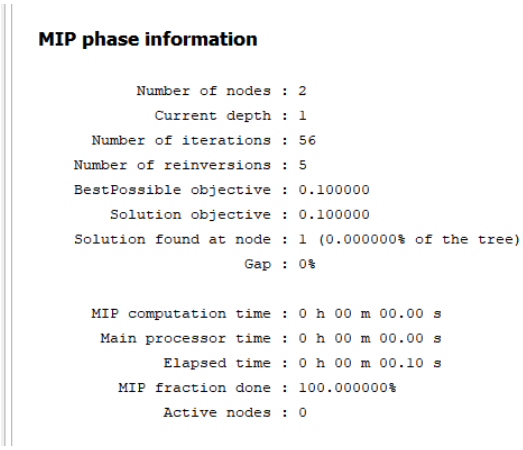
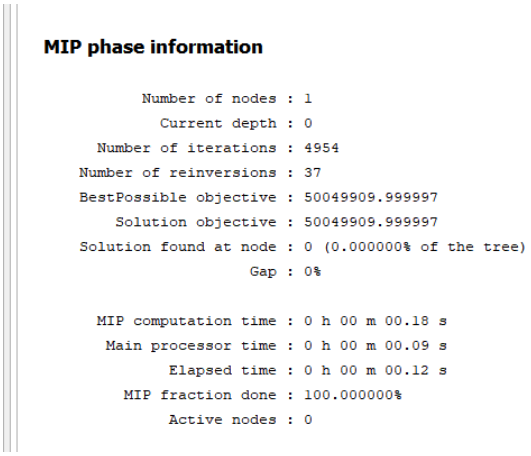


Figure 8.4: PingPong statistics using aggregation

Parameter	n = 10	n = 100	n = 200	n = 1000	n = 2000
Number of nodes	2	2	2	2	2
Current depth	1	1	1	1	1
number of iterations	56	317	517	2117	4117
Source solution	MIP	MIP	MIP	MIP	MIP

Table 8.2: Statistics data given by the use of aggregation

Extremely interesting results have been found putting into operation `Feasibility Pump Heuristic`, deactivated so far.
You can find on Fig. 8.5 on page 93 the results obtained on the model with 2000 variables. On this models, and in all the rest, the branching tree is reduced to just 1 node, demonstrating the enormous potential of this heuristic.



MIP phase information

```
          Number of nodes : 1
            Current depth : 0
   Number of iterations : 4954
Number of reinversions : 37
BestPossible objective : 50049909.999997
    Solution objective : 50049909.999997
Solution found at node : 0 (0.000000% of the tree)
                   Gap : 0%

  MIP computation time : 0 h 00 m 00.18 s
   Main processor time : 0 h 00 m 00.09 s
          Elapsed time : 0 h 00 m 00.12 s
     MIP fraction done : 100.000000%
          Active nodes : 0
```

Figure 8.5: PingPong2000 results found by FeasPump

Parameter	n = 10	n = 100	n = 200	n = 1000	n = 2000
Number of nodes	1	1	1	1	1
Current depth	0	0	0	0	0
number of iterations	43	459	949	4877	9786
Source solution	feas. pump	feas. pump	feas. pump	feas. pump	feas. pump

Table 8.3: Statistics data given by feaspump

Now, the question is, what would happen if we implemented `locks` heuristic on OMP solver?
`locks` played a central role throughout the analysis carried out on these same models with the SCIP solver. it is clear that, in the case of models with less than 1000 variables, it is thanks to this heuristic that SCIP manages to achieve excellent results even where OMP was forced to use clustering.
Furthermore, by testing `locks` in competition with `feaspump` through SCIP, we noticed how the solution was found by the first one, although not achieving in PingPong1000 and PingPong2000 the same good results obtained by OMP solver thanks to the second one.
This will certainly be the next step to address in the process of improving the company's solver.
The advice for the company is to delve deeper into this heuristic and possibly implement it, paying particular attention to smaller models, with less than a thousand variables.
In fact, such a number of variables should absolutely not be underestimated. In fact, as already explained previously, our variables represent time buckets for our company's production plan. A model of 200 variables can be translated in real terms to a production schedule of a product for almost one year, considering time buckets of one day, something regular for OMP. Furthermore, considering that OMP's production plans involve not just one, but hundreds of products, making our solver more efficient in terms of time could prove decisive for our planning.

APPENDIX A

Tables

A.1 Supporting Data

In this section you can find all tables used to carry on the data analysis.

Presolvers	PPx: H=S=off
boundshift	0
domcol	1
dualagg	0
dualcomp	1
dualinfer	0
dualsparsify	1
implics	1
inttobinary	0
milp	1
redvub	0
sparsify	1
stuffing	0
trivial	1
tworowbnd	0
dualfix	1
probing	1
symmetry	1
linear	2
components	1

Table A.1: Presolvers used setting off heuristics and separators. $x \in \{10, 100, 200, 1000\}$

Primal Heuristics	H=d, P=S=off		H=P=d, S=off	
	PPx	PP1000	PPx	PP1000
alns		ET 1.00, 5C		ET 2.00, 5C
conflictdiving		1C		1C
fracdiving		**ET 1.00, 1C**		
gins		3C		3C
intshifting		**2C**		**3C**
locks	1C,1F,1B	1C,1F,1B	1C,1F,1B	1C,1F,1B
oneopt	1C	1C	1C	1C
pcostdiving		1C		1C
randrounding		50C		50C
rens		**ET 1.00, 1C**		**1C**
rins		6C		6C
rounding		**437C**		**389C**
shifting		99C		99C
trivial	2C	2C	2C	2C
zirounding		984C		984C

Table A.2: Primal heuristics data analysing presolving impact mantaining primal heuristics active. $x \in \{10, 100, 200\}$

Separators	S=P=d, H=off				S=d, P=H=off			
	PP10	PP100	P200	PP1000	PP10	PP100	P200	PP1000
cut pool	**15**	**28**	**38**	**143**	**19**	**31**	**41**	**147**
aggregation	**8**	**12**	**13**	**14**	**10**	**13**	**14**	**15**
clique	1	1	1	1	1	1	1	1
gomory	**8**	12	13	14	**10**	12	13	14
impliedbounds	**8**	**12**	**13**	**14**	**10**	**13**	**14**	**15**
mcf	1	1	1	1	1	1	1	1
mixing	**8**	**12**	**13**	16	**10**	**13**	**14**	16
rlt	**8**	**8**	**8**	**8**	**10**	**10**	**10**	**10**
zerohalf	**8**	**12**	**13**	**14**	**10**	**13**	**14**	**15**

Table A.3: Separators used evaluating presolving effect on separators, maintaining heuristics disabled

Separators	P=S=d, H=f		P=d, H=f, S=a	
	PP10	P200	PP10	P200
cut pool	**16**	**43**	**19**	**51**
aggregation	**8**	**15**	**11**	**28**
clique	1	**1**	1	**2**
closecuts	**0**	**0**	**1**	**2**
gomory	**8**	**13**	**10**	**20**
impliedbounds	**8**	**15**	**11**	**28**
mcf	1	**1**	1	**2**
mixing	**8**	**15**	**11**	**28**
rlt	**8**	**10**	**10**	**20**
zerohalf	**8**	**15**	**11**	**28**

Table A.7: Separators behavior strengthening separators when setting heuristics emphasis fast, for models PingPong10,PingPong200

Primal Heuristics	P=H=d, S=off		P=d, H=a, S=off				P=d, H=f, S=off			
	PPx	PP1000	PP10	PP100	PP200	PP1000	PP10	PP100	PP200	PP1000
adaptivediving						ET 1.00, 1C				6C,1F,1B
alns		ET 2.00, 5C	1C	1C		ET 9.00, 20C				
bound					1C	50C				
coefdiving						1C				
conflictdiving		1C				ET 1.00,1C				ET 1.00, 4C
feaspump						1C				
fracdiving										
gins		3C				3C				
intshifting		3C				2C	1C,1F,1B	1C,1F,1B	1C,1F,1B	11C,1F,1B
localbranching						2C				
locks	1C,1F,1B	1C,1F,1B	1C,1F,1B	1C,1F,1B	1C,1F.1B	50C,1F,1B				
mutation						ET 1.00, 3C				
oneopt	1C	1C	1C	1C	1C	1C	3C	3C	3C	3C
proximity						ET 2.00, 1C				
pscostdiving		1C				EC 1.00, 1C				
randrounding		50C				100C	2C	6C	11C	51C
rens		1C				ET 4.00, 20C				
repair						1C				
rins		6C				6C				
rounding		389C				417C	10C	100C	150C	398C
shiftandpropagate							1C,1F,1B	1C,1F,1B	1C,1F,1B	1C,1F,1B
shifting		99C				167C	2C,1F	11C	21C	100C
trivial	2C	2C	2C	2C	2C	51C	2C	2C	2C	2C
trustregion						ET 1.00, 1C				
zeroobj			1C,1F,1B	1C,1F,1B	1C,1F,1B	1C,1F,1B				
zirounding		984C				992C	10C,1F,1B	100C,1F,1B	200C,1F,1B	993C

Table A.4: Primal heuristics evaluated setting off all separators. $x \in \{10, 100, 200\}$

B&B tree	P=S=d, H=off			P=d, H=S=off			P=d,H=off,S=a			P=d,H=off,S=f		
	PP10	PP100	P200	PP10	PP100	P200	PP10	PP100	PP200	PP10	PP100	PP200
Number of runs	1	1	1	1	1	1	1	1	1	1	1	1
Nodes	**5**	102	202	**12**	102	202	**5**	102	202	**7**	102	202
Feasible leaves	1	1	1	1	1	1	1	1	1	1	1	1
Infeas.leaves	1	1	1	1	1	1	1	1	1	1	1	1
Objective leaves	0	0	0	0	0	0	0	0	0	0	0	0
Nodes (total)	**5**	102	202	**12**	102	202	**5**	102	202	**7**	102	202
Nodes left	0	0	0	0	0	0	0	0	0	0	0	0
Max depth	**3**	100	200	**10**	100	200	**3**	100	200	**5**	100	200
Max depth (total)	**3**	100	200	**10**	100	200	**3**	100	200	**5**	100	200
Backtracks	**3**	99	200	**9**	99	200	**3**	99	200	**5**	99	200
Early backtracks	0	0	0	0	0	0	0	0	0	0	0	0
Nodes exc. Ref.	0	0	0	0	0	0	0	0	0	0	0	0
Delayed cutoffs	**2**	99	199	**9**	99	199	**2**	99	199	**4**	99	199
Repropagations	**3**	99	200	**9**	99	200	**3**	99	200	**5**	99	200
Avg switch length	**3.60**	5.86	5.94	**4.83**	5.86	5.94	**3.60**	5.86	5.94	**4.29**	5.86	5.94
Switching time	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table A.5: B&B tree changing the setting of separators when disabling heuristics

B&B tree	P=S=d, H=f			P=d,H=f,S=off			P=d,H=f,S=a			P=d,H=S=f		
	PP10	PP100	P200	PP10	PP100	P200	PP10	PP100	PP200	PP10	PP100	PP200
Number of runs	1	1	1	1	1	1	1	1	2	1	1	1
Nodes	**7**	201	**401**	**21**	201	**401**	**1**	201	**1**	**11**	201	**401**
Feasible leaves	0	0	0	0	0	0	0	0	0	0	0	0
Infeas.leaves	**3**	**99**	**199**	**10**	**100**	**200**	**0**	**99**	**0**	**5**	**99**	**199**
Objective leaves	**1**	**2**	**2**	**1**	**1**	**1**	**0**	**2**	**1**	**1**	**2**	**2**
Nodes (total)	**7**	201	**401**	**21**	201	**401**	**1**	201	**2**	**11**	201	**401**
Nodes left	0	0	0	0	0	0	0	0	0	0	0	0
Max depth	**3**	100	**200**	**10**	100	**200**	**0**	100	**0**	**5**	100	**200**
Max depth (total)	**3**	100	**200**	**10**	100	**200**	**0**	100	**0**	**5**	100	**200**
Backtracks	**3**	**98**	**199**	**9**	**99**	**200**	**0**	**98**	**0**	**5**	**98**	**199**
Early backtracks	0	0	0	0	0	0	0	0	0	0	0	0
Nodes exc. Ref.	0	0	0	0	0	0	0	0	0	0	0	0
Delayed cutoffs	0	0	0	0	0	0	0	0	0	0	0	0
Repropagations	1	1	1	1	1	1	0	1	0	1	1	1
Avg switch length	**2.57**	**2.99**	**3.96**	**3.62**	**3.96**	**2.99**	**2.00**	**3.96**	**2.00**	**2.73**	**3.96**	**2.99**
Switching time	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table A.6: Branch and Bound tree changing the setting of separators when imposing heuristics emphasis fast

Separators	P=H=S=d	P=H=d, S=a	P=H=d, S=f
cut pool		1	
aggregation	5	7	5
clique	1	1	1
gomory	5	6	5
impliedbounds	5	7	5
mcf	1	1	
mixing	5	7	5
rlt	1	5	1
zerohalf	5	7	5

Table A.8: PingPong1000.lp separators behavior switching the setting for all separators

A.2 Additional Data

In the following tables you can find additional informations and data about parameters like propagators, constraint handles, solution values. Although not of primary importance, also these data have been used to reach the conclusions of the analysis. It's possible to verify in this section how constraint handlers and propagators have been used switching setting of the program and solution values. That is, regarding solution values, how many times a feasible solution was found and improved, the value of the first solution and the gap between the fist solution and the optimal solution, and the value of final primal and dual bounds.
In addiction, which types of operations and how many operations of each type were performed by constraint handlers and propagators are provided, switching the setting for heuristics and separators, on models Ping-Pong.lp and PingPong10.lp

A.2.1 PingPong

Propagators	P=S=H=d	P=S=d, H=off	P=S=d, H=a	P=S=d, H=f
dualfix	propagate 9	propagate 1	propagate 10	propagate 19
pseudoobj	propagate 8, DomReds 39	propagate 7, Domreds 18	propagate 10, Domreds 40	propagate 41, Domreds 41
redcost rootredcost		propagate 1, Domreds 20		propagate 15 propagate 2, Domreds 20
vbounds				propagate 1

Table A.10: Ping-Pong.lp propagators behavior changing heuristics setting

Solutions	Ping-Pong.lp			
	P=S=H=d	P=s=d, H=off	P=S=d, H=f	P=S=d, H=a
Solutions found	1 (1 improvement)	1 (1 improvement)	3 (3 improvements)	2 (2 improvements)
First Solution	$9.99999999999979e^{-02}$ (in run 1, after 1 nodes, ti,e 0.00, depth 11, found by <locks>	$9.99999999999872e^{-02}$ (in run 1, after 4 nodes, time 0.00, depth 3, found by <relax-ation>)	$5.49900000000000e^{+03}$ (in run 1, after 1 nodes, time 0.00, depth 12, found by <shiftand-propagate>)	$5.39900000000052e^{+03}$ (in run 1, after 0 nodes, time 0.00, depth 0, found by <zeroobj>)
Gap First Sol	infinite	42.86%	infinite	infinite
Gap Last Sol	infinite	42.86%	42.86 %	infinite
Primal Bound	$9.99999999999979e^{-02}$	$9.99999999999872e^{-02}$	$1.00000000004796e^{-02}$ (in run 1, after 1 nodes, time 0.00, depth 11, found by <locks>)	$9.99999999976353e^{-02}$ (in run 1, after 1 nodes, time 0.00, depth 11, found by <locks>)
Dual Bound	$9.99999999999979e^{-02}$	$9.99999999999872e^{-02}$	$1.00000000004796e^{-02}$	$9.99999999976353e^{-02}$
Gap	0.00%	0.00%	0.00%	0.00 %

Table A.9: Ping-Pong.lp solutions changing the setting for all heuristics

Constraints	P=S=d, H=off	P=d, S=H=off	P=d, H=off, S=a	P=d, H=off, S=f
benderslp	enfolp 4	enfolp 11	enfolp 4	enfolp 6
integral	enfolp 4, Children 6	enfolp 11, Children 20	enfolp 4, Children 6	enfolp 6, Children 10
linear	number 10, MaxNumber 10, separate 8, enfolp 1, propagate 33, CutOffs 1, Domreds 22	number 10, MaxNumber 10, propagate 47, CutOffs 1, Domreds 24	number 10, MaxNumber 10,separate 9, propagate 33, enfolp 1,cutoffs 1, DomReds 22	number 10, MaxNumber 10, separate 6, propagate 38, cutoffs 1, domreds 22

Table A.11: Ping-Pong.lp use of constraint handlers changing the setting of separators when disabling heuristics

Propagators	P=S=d, H=off	P=d, H=S=off	P=d, H=off, S=a	P=d, H=off, S=f
dualfix	propagate 1	propagate 1	propagate 1	propagate 1
pseudoobj	propagate 7, DomReds 18	propagate 13, DomReds 37	propagate 7, DomReds 18	propagate 10, DomReds 18
rootredcost	propagate 1, Domreds 20	propagate 1, Domreds 20	propagate 1, Domreds 20	propagate 1, DomReds 20

Table A.12: Ping-Pong.lp use of propagators changing the setting of separators when disabling heuristics

A.2.2 PingPong10

Constraints	P=S=H=d	P=S=d, H=off	P=S=d, H=a	P=S=d, H=f
benderslp	check 10	enfolp 4, check 2	check 12	enfolp 3, check 22
integral	check 10	enfolp 4, check 2, children 6	check 12	enfolp 3, check 22, children 6
linear	number 10, MaxNumber 10, propagate 43, check 5,DomReds 21	number 10, MaxNumber 10, separate 8, propagate 32, enfolp 1,cutoffs 1,DomReds 20	number 10, MaxNumber 10, propagate 41, check 6, DomReds 24	number 10, MaxNumber 10, separate 8, propagate 71, check 16, cutoffs 3, domreds 40
benders	check 2	enfolp 1, check 1	check 4	check 4
countsols	check 2	enfolp 1, check 1	check 4	check 4

Table A.13: PingPong10.lp use of constraint handlers switching the setting for all heuristics

Propagators	P=S=H=d	P=S=d, H=off	P=S=d, H=a	P=S=d,H=f
dualfix	propagate 10	propagate 1	propagate 8	propagate 19
pseudoobj	propagate 9, DomReds 39	propagate 6, DomReds 19	propagate 8, DomReds 40	propagate 41, DomReds 41
redcost				propagate 15
rootredcost		propagate 1, Domreds 20		propagate 2, DomReds 20
vbounds				propagate 1

Table A.14: PingPong10.lp use of propagators changing the setting of heuristics

A.2.3 PingPong100

Solutions	PingPong100.lp		
	P=S=d, H=off	P=S=d, H=a	P=S=d, H=f
Solutions found	1 (1 improvement)	2 (2 improvements)	3 (3 improvements)
First Solution	$9.99999999999979e^{-01}$ (in run 1, after 101 nodes, time 0.00, depth 100, found by <relaxation>)	$5.04890000000004e^{+05}$ (in run 1, after 0 nodes, time 0.00, depth 0, found by <zeroobj>)	$5.0499000000000e^{+05}$ (in run 1, after 1 nodes, time 0.00, depth 102, found by <shiftandpropagate>)
Gap First Sol	1328.57%	infinite	infinite
Gap Last Sol	1328.57%	infinite	1328.57 %
Primal Bound	$9.99999999999979e^{-01}$	$9.99999999702249e^{-01}$ (in run 1, after 1 nodes, time 0.00, depth 101, found by <locks>)	$1.00000000046989e^{-00}$ (in run 1, after 101 nodes, time 0.00, depth 99, found by <zirounding>)
Dual Bound	$9.99999999999979e^{-01}$	$9.99999999702249e^{-01}$	$1.00000000046989e^{-00}$
Gap	0.00%	0.00%	0.00%

Table A.15: PingPong100.lp solutions changing the setting for all heuristics

Solutions	PingPong100.lp		
	P=S=H=d, locks freq -1	P=S=d,H=a, locks freq -1	P=S=d,H=a, zeroobj freq -1
Solutions found	3 (3 improvements)	3 (3 improvements)	1 (1 improvement)
First Solution	$5.0499000000000e^{+05}$ (in run 1, after 1 nodes, time 0.00, depth 102, found by <shiftandpropagate>)	$5.0499000000000e^{+05}$ (in run 1, after 1 nodes, time 0.00, depth 102, found by <zeroobj>)	$+9.99999999999979e^{-1}$ (in run 1, after 1 nodes, time 0.00, depth 102, found by <locks>)
Gap First Sol	infinite	infinite	infinite
Gap Last Sol	1566.67 %	1566.67 %	1566.67 %
Primal Bound	$9.99999999999979e^{-01}$ (in run 1, after 101 nodes, time 0.00, depth 99, found by <zirounding>)	$9.99999999999979e^{-01}$ (in run 1, after 101 nodes, time 0.00, depth 99, found by <feaspump>)	$9.99999999999979e^{-01}$ (in run 1, after 101 nodes, time 0.00, depth 99, found by <locks>)
Dual Bound	$9.99999999999979e^{-01}$	$9.99999999999979e^{-01}$	$9.99999999999979e^{-01}$
Gap	0.00%	0.00%	0.00%

Table A.16: PingPong100.lp solutions disabling single heuristics

A.2.4 PingPong200 and PingPong1000

Solutions	PingPong200.lp	
	P=S=H=d	P=S=d, H=a
Solutions found	1 (1 improvements)	2 (2 improvements)
First Solution	$1.9999999999996e^{-00}$ (in run 1, after 1 nodes, time 0.00, depth 201, found by <locks>)	$2.00988000000001e^{+06}$ (in run 1, after 0 nodes, 0.00 seconds, depth 0, found by <zeroobj>)
Gap First Sol	infinite	infinite
Gap Last Sol	infinite	infinite
Primal Bound	$1.99999999999996e^{-00}$	$1.99999999946212e^{+00}$ (in run 1, after 1 nodes, 0.00 seconds, depth 201, found by <locks>)
Dual Bound	$1.99999999999996e^{-00}$	$1.99999999946212e^{+00}$
Gap	0.00%	0.00%

Table A.17: PingPong200.lp solutions changing the setting for all heuristics

Solutions	PingPong1000.lp			
	P=S=H=d	P=S=d, H=off	P=S=d, H=a	P=S=d, H=f
Solutions found	1 (1 improvements)	267 (267 improvements)	2 (2 improvements)	3 (3 improvements)
First Solution	$1.9999999999979e^{-00}$ (in run 1, after 1 nodes, time 0.00, depth 1001, found by <locks>)	$+9.92997700000000e^{+04}$ (in run 1, after 217 nodes, 0.00 seconds, depth 1, found by <relaxation>)	$+5.00497999999995e^{+07}$ (in run 1, after 0 nodes, 0.00 seconds, depth 0, found by <zeroobj>)	$+5.00499000000000e^{+07}$ (in run 1, after 1 nodes, 1.00 seconds, depth 1002, found by <shiftandpropagate>)
Gap First Sol	infinite	124000862.50 %	infinite	infinite
Gap Last Sol	infinite	352.49 %	infinite	8990.91 %
Primal Bound	$1.99999999999979e^{-00}$ (in run 1, after 1 nodes, time 0.00, depth 201, found by <locks>)	$+9.99999999999979e^{00}$ (in run 1, after 1374 nodes,9.00 seconds,depth 1000, found by <relaxation>)	$+1.000000000307334e^{01}$ (in run 1, after 1 nodes,0.00 seconds,depth 1001, found by <locks>)	$+1.00000000360606e^{01}$ (in run 1, after 225 nodes,1.00 seconds,depth 1001, found by <adaptivediving>)
Dual Bound	$1.99999999999979e^{-00}$	$+9.99999999999979e^{00}$	$+1.000000000307334e^{01}$	$+1.00000000360606e^{01}$
Gap	0.00%	0.00%	0.00%	0.00%

Table A.18: PingPong1000.lp solutions changing the setting for all heuristics

APPENDIX B

Definitions

	B&B tree parameter	Description
1	feasible leaves	number of leaf nodes processed with feasible relaxation solution
2	infeasible leaves	number of infeasible leaf nodes processed
3	delayed cutoffs	bool parameter to indicate that treeCutoff() call was delayed because of diving and has to be executed
4	repropagations	cyclicly increased counter to create markers for subtree repropagation
5	depth	depth in the tree

	Solution parameter	Description
6	Primal Bound	gets global primal bound (objective value of best solution or user objective limit) for the original problem
7	Dual Bound	global dual bound
8	Solution Found	number of feasible primal solutions found so far
9	Gap	gets current gap $\|(primalbound - dualbound)/min(\|primalbound\|, \|dualbound\|)\|$ if both bounds have same sign, or infinity, if they have opposite sign

	Presolvers	Description
10	boundshift	presolver that converts variables with domain [a,b] to variables with domain [0,b-a]
11	domcol	This presolver looks for dominance relations between variable pairs. From a dominance relation and certain bound/clique-constellations variable fixings mostly at the lower bound of the dominated variable can be derived. Additionally it is possible to improve bounds by predictive bound strengthening

12	dualagg	This presolver looks for variables which could not be handled by duality fixing because of one up-/downlock. If the constraint which delivers the up-/downlock has a specific structure, we can aggregate the corresponding variable
13	dualcomp	This presolver looks for variables with i) objcoef >= 0 and exactly one downlock ii) objcoef <= 0 and exactly one uplock and fixes the variable in case i) at the lower bound and in case ii) at the upper bound if a combination of singleton continuous variables can compensate the downlock in case i) and the uplock in case ii)
14	dualinfer	This presolver does bound strengthening on continuous variables (columns) for getting bounds on dual variables y. The bounds of the dual variables are then used to fix primal variables or change the side of constraints. For ranged rows one needs to decide which side (rhs or lhs) determines the equality
15	dualsparsify	This presolver attempts to cancel non-zero entries of the constraint matrix by adding scaled columns to other columns
16	milp	Calls the presolve library and communicates (multi-)aggregations, fixings, and bound changes to SCIP by utilizing the postsolve information. Constraint changes can currently only be communicated by deleting all constraints and adding new ones
17	redvub	This presolver looks for dominating variable bound constraints on the same continuous variable and discards them
18	sparsify	This presolver attempts to cancel non-zero entries of the constraint matrix by adding scaled equalities to other constraints
19	stuffing	Investigate singleton continuous variables if one can be fixed at a bound
20	trivial	round fractional bounds on integer variables, fix variables with equal bounds
21	tworowbnd	Perform bound tightening on two inequalities with some common variables

	Propagators	**Description**
22	dualfix	to fix roundable variables to best bound
23	pseudobj	to propagate the objective function using the cutoff bound and the pseudo objective value
24	rootredcost	to globally propagate against the cutoff bound through the root reduced cost
25	redcost	to propagate the variables against the cutoff bound using the reduced cost of an optimal solved LP relaxation
26	vbounds	to deduce global and local bound changes thanks to information provided by scip

	Primal Heuristics	**Description**

27	alns	Adaptive large neighborhood search heuristic that orchestrates popular LNS heuristics
28	intshifting	LP rounding heuristic that tries to recover from intermediate infeasibilities, shifts integer variables, and solves a final LP to calculate feasible values for continuous variables
29	oneopt	improvement heuristic that alters single variable values
30	randrounding	randomized LP rounding heuristic which also generates conflicts via an auxiliary probing tree
31	rens	heuristic that finds the optimal rounding to a given point
32	rounding	rounding heuristic that tries to recover from intermediate infeasibilities
33	shifting	LP rounding heuristic that tries to recover from intermediate infeasibilities and shifts continuous variables

	Separators	**Description**
33	aggregation	flow cover and complemented mixed integer rounding cuts separator
34	mcf	multi-commodity-flow network cut separator
35	mixing	mixing/star inequality separator
36	rlt	separator for cuts generated by Reformulation-Linearization-Technique

Bibliography

(2000), «Progress in Linear Programming-Based Algorithms for Integer Programming: An Exposition», *INFORMS J. on Computing*, vol. 12 (1), p. 2–23.

ACHTERBERG, T., BERTHOLD, T. e HENDEL, G. (2012), «Rounding and Propagation Heuristics for Mixed Integer Programming», in KLATTE, D., LÜTHI, H.-J. e SCHMEDDERS, K., curatori, «Operations Research Proceedings 2011», p. 71–76, Springer Berlin Heidelberg, Berlin, Heidelberg. (Cited at page 46)

ACHTERBERG, T., BERTHOLD, T., KOCH, T. e WOLTER, K. (2008), «Constraint Integer Programming: A New Approach to Integrate CP and MIP», in «Integration of AI and OR Techniques in Constraint Programming», URL https://api.semanticscholar.org/CorpusID:6951394.

ATAMTÜRK, S. M. W. P., ALPER (2005), «Integer-Programming Software Systems», *Annals of Operations Research*, URL https://doi.org/10.1007/s10479-005-3968-2.

BERTHOLD, T. e HENDEL, G. (2015), «Shift-and-Propagate», *Journal of Heuristics*, vol. 21, p. 73–106, URL https://api.semanticscholar.org/CorpusID:209833456. (Cited at page 44)

BERTHOLD, T., LODI, A. e SALVAGNIN, D. (2019), «Ten years of feasibility pump, and counting», *EURO Journal on Computational Optimization*, vol. 7 (1), p. 1–14, URL https://www.sciencedirect.com/science/article/pii/S219244062100109X. (Cited at page 43)

BESTUZHEVA, K., BESANÇON, M., CHEN, W.-K., CHMIELA, A., DONKIEWICZ, T., VAN DOORNMALEN, J., EIFLER, L., GAUL, O., GAMRATH, G., GLEIXNER, A., GOTTWALD, L., GRACZYK, C., HALBIG, K., HOEN, A., HOJNY, C., VAN DER HULST, R., KOCH, T., LÜBBECKE, M., MAHER, S. J., MATTER, F., MÜHMER, E., MÜLLER, B., PFETSCH, M. E., REHFELDT, D., SCHLEIN, S., SCHLÖSSER, F., SERRANO, F., SHINANO, Y., SOFRANAC, B., TURNER, M., VIGERSKE, S., WEGSCHEIDER, F., WELLNER, P., WENINGER, D. e WITZIG, J. (2023), «Enabling Research through the SCIP Optimization Suite 8.0», *ACM Trans. Math. Softw.*, vol. 49 (2), URL https://doi.org/10.1145/3585516. (Cited at page 51)

BRUNO, G., GENOVESE, A. e PICCOLO, C. (2014), «The capacitated Lot Sizing model: A powerful tool for logistics decision making», *International Journal of Production Economics*, vol. 155, p. 380–390, URL https://www.sciencedirect.com/science/article/pii/S0925527314000887, celebrating a century of the economic order quantity model.

COOK, W. J., CUNNINGHAM, W. H., PULLEYBLANK, W. R. e SCHRIJVER, A. (1998), *Combinatorial optimization*, John Wiley & Sons, Inc., USA, URL https://onlinelibrary.wiley.com/doi/book/10.1002/9781118033142.

GAMRATH, G., BERTHOLD, T., HEINZ, S. e WINKLER, M. (2019), «Structure-driven fix-and-propagate heuristics for mixed integer programming», *Mathematical Programming Computation*, vol. 11 (4), p. 675–702. (Cited at page 42)

ILYAS HIMMIC, N. E. H. I. E. H. A. M. F. S., EL MEHDI ER RAQABI (2023), «MILPS: An Automatic Tuner for MILP Solvers», *Computers and Operations Research*. (Cited at page 33)

IOMMAZZO, G., D'AMBROSIO, C., FRANGIONI, A. e LIBERTI, L. (2020), «A Learning-Based Mathematical Programming Formulation for the Automatic Configuration of Optimization Solvers», in «Machine Learning, Optimization, and Data Science: 6th International Conference, LOD 2020, Siena, Italy, July 19–23, 2020, Revised Selected Papers, Part I», p. 700–712, Springer-Verlag, Berlin, Heidelberg, URL https://doi.org/10.1007/978-3-030-64583-0_61. (Cited at page 33)

J.T.LINDEROTH e M.W.P.SAVELSBERGH (1998), «A Computational Study of Search Strategies for Mixed Integer Programming», .

MARTIN, A. (2001), *General Mixed Integer Programming: Computational Issues for Branch-and-Cut Algorithms*, p. 1–25, Springer Berlin Heidelberg, Berlin, Heidelberg, URL https://doi.org/10.1007/3-540-45586-8_1.

NADDEF, D. e RINALDI, G. (2002), *3. Branch-And-Cut Algorithms for the Capacitated VRP*, p. 53–84, URL https://epubs.siam.org/doi/abs/10.1137/1.9780898718515.ch3.

WALLACE, C. (2010), «ZI Round, a MIP Rounding Heuristic», *Journal of Heuristics*, vol. 16 (5), p. 715–722, URL https://doi.org/10.1007/s10732-009-9114-6. (Cited at page 45)

Websites consulted

- Wikipedia – www.wikipedia.org
- SCIP Optimization Suite – https://scipopt.org/
- OMPartners – https://omp.com/solution

- Applied Mathematical Programming – http://web.mit.edu/15.053/www/AppliedMathematicalProgramming.pdf
- Branch-and-Cut algorithm – https://coral.ise.lehigh.edu/~ted/files/computational-mip/lectures/Lecture12.pdf

Heuristics File References

- Heuristic locks.h File Reference –https://www.scipopt.org/doc/html/heur__locks_8h.php
- Heuristic feaspump.h File Reference – https://www.scipopt.org/doc/html/heur__feaspump_8h.php
- Heuristic zirounding.h File Reference – https://www.scipopt.org/doc/html/heur__zirounding_8h.php
- Heuristic zeroobj.h File Reference – https://www.scipopt.org/doc/html/heur__zeroobj_8h.php
- Heuristic shiftandpropagate.h File Reference – https://www.scipopt.org/doc/html/heur__shiftandpropagate_8h.php
- Heuristic shifting.h File Reference – https://www.scipopt.org/doc/html/heur__shifting_8h.php
- Heuristic oneopt.h File Reference – https://www.scipopt.org/doc/html/heur__oneopt_8h.php
- Heuristic adaptivediving.h File Reference – https://www.scipopt.org/doc/html/heur__adaptivediving_8h.php
- Heuristic alns.h File Reference – https://www.scipopt.org/doc/html/heur__alns_8h.php
- Heuristic clique.h File Reference – https://www.scipopt.org/doc/html/heur__clique_8h.php
- Heuristic oneopt.h File Reference – https://www.scipopt.org/doc/html/heur__oneopt_8h.php
- Heuristic randrounding.h File Reference – https://www.scipopt.org/doc/html/heur__randrounding_8h.php
- Heuristic trivial.h File Reference – https://www.scipopt.org/doc/html/heur__trivial_8h.php
- Heuristic rounding.h File Reference – https://www.scipopt.org/doc/html/heur__rounding_8h.php