KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Technology

Decomposition-based algorithms for optimization problems

Túlio Ângelo Machado Toffolo

Supervisors:
Prof. dr. ir. G. Vanden Berghe
Prof. dr. P. De Causmaecker
Prof. dr. F. Spieksma

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Technology (PhD)

November 2017

Decomposition-based algorithms for optimization problems

Túlio Ângelo Machado TOFFOLO

Examination committee:
Prof. dr. ir. J. Ivens, chair
Prof. dr. ir. G. Vanden Berghe, supervisor
Prof. dr. P. De Causmaecker, co-supervisor
Prof. dr. F. Spieksma, co-supervisor
Prof. dr. ir. S. Vandewalle
Prof. dr. L. De Raedt
Prof. dr. R. Leus
Prof. dr. ir. D. Vigo
(Università di Bologna, Italy)
W. Tielemans
(Ordina, Belgium)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Technology (PhD): Computer Science

November 2017

Uitgegeven in eigen beheer, Túlio Ângelo Machado Toffolo, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

Acknowledgments

Developing and studying algorithms has long been one of my passions. It is thus by no means a coincidence that I have spent almost four years as a PhD researcher on this very topic, dedicating myself towards studying, researching and coding various approaches to address optimization problems. Luck has, however, played an important role in the course of my academic life. And lucky I certainly was, for encountering in my professional path amazing people like Marcone Souza and Greet Vanden Berghe. More than supervisors, they are friends who have my eternal gratitude for all their advice throughout my academic career. Advice which often went beyond the academic sphere and in diverse ways have contributed into making me the person I am today.

Concerning specifically the PhD, I would like to thank my supervisors Greet Vanden Berghe, Patrick De Causmaecker and Frits Spieksma for the unique opportunity of doing research at KU Leuven. It was an honor to have your support and guidance. Next, I thank Daniele Vigo, Stefan Vandewalle, Roel Leus, Luc De Raedt and Wouter Tielemans for being part of my examination committee and providing valuable feedback. Additionally, I would like to thank Jan Ivens for agreeing to chair my PhD defense.

Throughout the PhD I had the privilege of working in multiple problem domains, which provided me with the chance of collaborating with a large number of colleagues. To all co-authors I have worked with, I send my appreciation. Thank you all very much, in particular Thibaut Vidal and Haroldo Santos, who came all the way from Brazil to Belgium for our joint projects.

Moving to a country so far away from home often leaves one feeling isolated. It was, however, never the case for me. Alix, Jean-Pierre and Lynn received me as part of their family. Fleur's family further contributed to make Belgium feel more like home. It was a privilege to be with all of you. I would like to also thank Vanessa, a very special soul. I will always be grateful for your support. At work, even though I was the only international student for a long while at CODeS, the Belgian colleagues, in particular Tony, made it very easy for me to adapt.

Speaking of colleagues, I would like to dedicate at least one paragraph to those with whom I spent most of my time in Belgium. I would like to begin by thanking my dear friend Luke Connolly, who deserves more than a single mention. Luke went beyond providing editorial consultation, and indirectly taught me to see the English language in a different manner. I will miss our discussions together with Everton and conversations with Erik. I would like to also explicitly thank Everton, Tony, Jannes, Sam, Catherine, Sam Van Malderen, Federico, Thomas, Pieter, Wim, Evert-Jan, Jan, Eline, Thomas Sys, David, Bert, and the new colleagues Toni, Manos, Yihang, Reshma, Annelies, Farzaneh and Michiel. You all have made CODeS a very enjoyable working place.

I would like to also leave some words to my family and friends from Brazil. Thank you, grandma Gracinda, friends, cousins, ants and uncles, in particular uncle Ronaldo and Ângelo, for warmly welcoming me everytime I went home. To my beloved parents and brother, I send a special message. Even from so far away, you managed to be the most important people at all relevant moments. Your unwavering support and confidence in me throughout the last 32 years is more than I could have ever asked for.

My final words go to Fleur, who was there for me during the toughest moments of the PhD. You illuminated my days. Thank you for everything!

Túlio A. M. Toffolo Ghent, November 2017

Abstract

Despite the recent very significant progress concerning algorithms for combinatorial optimization problems, most large instances of $\mathcal{NP}$-Hard problems remain intractable by general solvers, motivating the development of problem-specific (meta)heuristic algorithms. While often resulting in acceptable results, the development of problem-specific algorithms is time-consuming and traditionally very expensive. In response to these shortcomings, this manuscript investigates decomposition-based algorithms as an alternative for addressing combinatorial optimization problems. These algorithms decompose a problem into multiple subproblems so as to efficiently approach it. Generally, such subproblems are much easier to solve than the entire problem, thereby enabling one to address large problems by employing, for example, general solvers.

Decomposition-based algorithms may be categorized as follows: those which optimally solve the original problem and those which address it heuristically with the goal of producing high-quality solutions. Both algorithm classes are addressed within this thesis, whose primary focus concerns decomposition-based heuristic algorithms. However, can these algorithms replace state-of-the-art problem-specific (meta)heuristics? To answer this question, six different problems are investigated throughout this manuscript, ranging from scheduling to logistic problems. All six problems are associated with challenging benchmark instances extensively studied in the literature, which permits a comparison of the developed methods against several other algorithms. Multiple decomposition strategies are investigated, employing the problems' structure, the decisions associated with them or simple strategies seeking to reduce their size. Several decomposition-based algorithms are proposed and thoroughly analyzed, with some of them proven general despite the various individual problem characteristics. A general framework is proposed, successfully addressing not only some of the problems studied throughout the thesis but also an additional

one from the literature.

Computational experiments validate the proposed algorithms, which result in several improvements over the state-of-the-art for all six investigated problems. This thesis not only contributes towards various individual problem domains, but also to the future of decomposition-based methodologies. Its findings, combined with the practical advantages of decomposition-based algorithms, culminate in a discussion concerning the role of decomposition within state-of-the-art algorithms for combinatorial optimization problems. Furthermore, in the spirit of reproducible science, whenever possible the source code produced was made publicly available online together with instance and solutions files.

Beknopte samenvatting

Ondanks significante doorbraken in de combinatorische optimalisatie blijken algemene *solvers* nog altijd tekort te schieten om grote instanties van $\mathcal{NP}$-harde problemen op te lossen. Voor deze open optimalisatievraagstukken biedt de ontwikkeling van probleemspecifieke (meta)heuristieken een uitweg. Dergelijke algoritmen, hoewel meestal erg performant, vergen een aanzienlijke ontwikkeltijd en zijn bijgevolg bijzonder duur.

Dit manuscript onderzoekt hoe decompositie-algoritmen een alternatief kunnen bieden voor zowel algemene solvers als probleemspecifieke heuristieken voor combinatorische optimalisatieproblemen. Een decompositie-algoritme benadert een probleem efficiënt door het op te splitsen in een aantal deelproblemen, die zelf gewoonlijk veel gemakkelijker zijn dan het oorspronkelijke optimalisatieprobleem. Met deze methodologie worden grote problemen alsnog oplosbaar met, bijvoorbeeld, algemene solvers.

Men kan decompositie-algoritmen als volgt categoriseren: algoritmen die het originele probleem optimaal oplossen en heuristische alternatieven die goede benaderende oplossingen berekenen. Terwijl dit onderzoek hoofdzakelijk decompositie-gebaseerde heuristieken bestudeert, komen de beide categorieën aan bod in het manuscript. De centrale onderzoeksvraag luidt 'Kunnen deze decompositie-algoritmen de huidige probleemspecifieke (meta)heuristieken vervangen?' Om deze vraag te beantwoorden onderzoekt dit werk zes verschillende logistieke problemen, gaande van *scheduling* tot rittenplanning. Bij elk van deze problemen horen uitdagende instanties, uitgebreid bestudeerd in de wetenschappelijke literatuur. Deze referentie-instanties vergemakkelijken een kwantitatieve performantievergelijking met bestaande algoritmen.

Het ontwerp van de verschillende decompositiestrategieën is ofwel geïnspireerd op de probleemstructuur of op veel eenvoudiger methoden die de probleemdimensies

trachten te verminderen. Een grondige analyse van deze nieuwe decompositiebenaderingen bewijst de optimaliteit en de algemene toepasbaarheid van enkele algoritmen, die niettemin individuele probleemkarakteristieken aankunnen.

Het algemene raamwerk ontwikkeld tijdens dit onderzoek overstijgt de problemen in dit manuscript. Zonder noemenswaardige inspanning werd ook een ander optimalisatieprobleem uit de wetenschappelijke literatuur succesvol opgelost.

Uitgebreide computationele experimenten bevestigen de kwaliteit van de voorgestelde algoritmen, die talrijke nieuwe beste oplossingen laten noteren voor elk van de zes bestudeerde problemen. Deze thesis levert niet enkel een bijdrage tot elk individueel probleemdomein maar, bovenal, tot de decompositiemethoden van de toekomst.

De wetenschappelijke realisaties en praktische voordelen van decompositiegebaseerde algoritmen komen bijeen in een discussie over de rol van decompositie in actuele algoritmen voor combinatorische optimalisatieproblemen. In de geest van reproduceerbare wetenschap is de ontwikkelde broncode publiek beschikbaar, samen met de instanties en hun oplossingsbestanden.

Abbreviations

APD	Average Project delay
BFS	Best First Search
BKS	Best Known Solution
B&S	Balas & Simonetti Neighborhood
CP	Constraint Programming
CPD	Critical Path Duration
CVRP	Capacitated Vehicle Routing Problem
DFS	Depth First Search
ESICUP	EURO Special Interest Group on Cutting and Packing
GA	Genetic Algorithm
GPSP	Generalized Project Scheduling Problem
HBSS	Heuristic-Biased Stochastic Sampling
HC	Hill Climbing
ILS	Iterated Local Search
INRC-1	First International Nurse Rostering Competition
IP	Integer Programming
LAHC	Late Acceptance Hill-Climbing
LB	Lower Bound

LP	Linear Programming
MCLP	Multiple Container Loading Problem
MISTA	Multidisciplinary International Scheduling Conference: Theory and Applications
MPSP	Multi-Project Scheduling Problem
MSSCSP	Multiple Stock-Size Cutting Stock Problem
NRP	Nurse Rostering Problem
OSI	Open Solver Interface
PSP	Project Scheduling Problem
RCPSP	Resource-Constrained Project Scheduling Problem
SAT	Boolean Satisfiability Problem
SBVRP	Swap-Body Vehicle Routing Problem
SGS	Serial Generation Scheme
SMPTSP	Shift Minimization Personnel Task Scheduling Problem
TMS	Total MakeSpan
TPD	Total Project Delay
TSP	Traveling Salesman Problem
TUP	Traveling Umpire Problem
UB	Upper Bound
UHGS	Unified Hybrid Genetic Search
VeRoLog	EURO Working Group on Vehicle Routing and Logistics
VND	Variable Neighborhood Descent
VRP	Vehicle Routing Problem

Contents

List of Algorithms

Chapter 6: Capacitated Vehicle Routing Problem

Chapter 7: Swap-body Vehicle Routing Problem

Chapter 8: Multiple Container Loading Problem

List of Figures

Chapter 3: Nurse Rostering Problem

Chapter 4: Project Scheduling Problem

Chapter 5: Towards a general solver

Introduction to Part II

Chapter 6: Capacitated Vehicle Routing Problem

Chapter 7: Swap-body Vehicle Routing Problem

Chapter 8: Multiple Container Loading Problem

List of Tables

Chapter 2: Traveling Umpire Problem

Chapter 3: Nurse Rostering Problem

Chapter 4: Project Scheduling Problem

Chapter 6: Capacitated Vehicle Routing Problem

Chapter 7: Swap-body Vehicle Routing Problem

Chapter 8: Multiple Container Loading Problem

Chapter 1

Introduction

Throughout the last several decades there has been remarkable progress within the *Operational Research* (OR) field. The amount of progress is particularly significant when considering fundamental optimization problems such as the Traveling Salesman Problem. However, these advances are not limited to traditional problems. General solvers, employing *Mixed Integer Programming* (MIP) and *Constraint Programming* (CP) for example, have also evolved at an impressive rate. CPLEX, one of the leading commercial MIP solvers, has become approximately 90 times faster between the years 1998 and 2012 (Achterberg and Wunderling, 2013). These advances together with recent gains in computational power have resulted in breakthroughs and the solving of several previously unsolved problem instances.

Despite all the recent progress, large instances of most $\mathcal{NP}$-Hard problems remain intractable by general solvers. This is primarily due to the fact that these solvers employ exponential time complexity algorithms, in addition to them often being unable to explore key problem characteristics. This situation is even more evident when considering real-world problems from industry, given these problems generally exhibit a large number of constraints and consist of multiple interconnected problems. Such large problems are predominantly addressed by problem-specific *heuristics*, which often obtain satisfactory results within short computational time. However, developing such problem-specific algorithms is time-consuming and thus very expensive. Additionally, these algorithms do not take advantage of recent progress within general solvers.

This research seeks to combine both heuristic and exact algorithms, taking advantage of principles from both paradigms. Note, however, that this integration is by no means an idea here proposed. It is in fact an established research theme, with such integration typically implemented by decomposition methods.

Decomposing a problem or system consists in breaking it into smaller parts. In our context these constituent parts represent subproblems. Each subproblem addresses only a subset of the problem components, being generally much easier to solve than the original problem. Subproblem solutions are then combined to generate a solution for the entire original problem. Note that even MIP solvers themselves implement decomposition approaches, where subproblems are mostly defined by either the soft-fixation of variables, as in *Local Branching* (Fischetti and Lodi, 2003, Hansen et al., 2006) and similar methods, or by hard-fixation of variables, as in *Relaxation Induced Neighborhood Search* (Danna et al., 2003).

There are many advantages in applying decomposition, specially when complex, difficult-to-solve problems are considered. One of these advantages is the possibility of employing general solvers to handle subproblems. The generation of reusable code represents another potential advantage.

This thesis studies decomposition approaches that result in both *exact* and *heuristic* algorithms, which may or may not employ general solvers. In a *decomposition-based exact algorithm*, the problem is decomposed to result in an algorithm that ultimately produces an optimal solution for the original problem. Moreover, problems can be decomposed resulting in a relaxation of the original problem. These relaxations can provide strong bounds, which may for instance be used within a branch-and-bound framework, deriving an exact algorithm. It is the case of the decomposition schemes proposed by Dantzig and Wolfe (1960) and Benders (1962) for linear programming models, for example. The main drawback of decomposition-based exact algorithms is their exponential worst-case time complexity. While often being capable of solving larger problems than MIP and CP general solvers (as in Toffolo et al., 2017a), their exponential time complexity remains an issue for many real-world applications.

Decomposition-based heuristic algorithms divide the problem heuristically into subproblems, which are generally solved to optimality. This class of algorithms represents an alternative when addressing problems for which exact algorithms result in unreasonably long computational times, which is the case for a large number of real-world applications. However, one drawback is clear: combining (optimal) solutions for heuristically-defined subproblems does not necessarily

result in an optimal solution for the original problem. In fact, there is often no information concerning optimality. Despite this drawback, decomposition-based heuristic algorithms symbolize a growing trend in combinatorial optimization (Maniezzo et al., 2010), and represent the main theme of this thesis.

When looking at the literature concerning decomposition-based algorithms, it is noticeable how many pervasive terms are employed, such as heuristic decomposition, subproblem optimization, matheuristic, and very large neighborhood search. These terms, although often employed in different contexts, generally refer to algorithms sharing similar characteristics. Heuristic decomposition and subproblem optimization are terms usually employed to allude to problem size reduction strategies, as in Hansen et al. (2001), Brunner (2010) and Krüger et al. (2016) for example. The term matheuristic refers to algorithms which employ mathematical programming methods, such as MIP, to produce heuristics (Boschetti et al., 2009), while very large neighborhood search refers to the employment of exponentially-large neighborhoods within local search, which are generally explored by resolving subproblems (see Ahuja et al., 2002).

The approaches proposed and studied throughout this thesis go beyond matheuristics and very large neighborhood search. In fact, a MIP formulation is not always a requirement and subproblem solutions are often employed outside local search. Although MIP formulations are employed to solve subproblems in many cases, and also within local search methods, we prefer the term *decomposition-based heuristic* given its generality and contextual appropriateness.

In total, this thesis addresses six different optimization problems, for which different decomposition strategies are proposed and evaluated. For all six problems, the decomposition-based methodologies here proposed represent the current state-of-the-art concerning solution quality, outperforming previous approaches on benchmark instances. Some methods are also state-of-the-art in terms of computational performance, obtaining very strong results within limited runtimes.

Among the work developed during the PhD, we have decided to include only those which consider benchmarked and challenging data within this thesis. By benchmarked data we mean data consistently utilized by various authors to evaluate algorithms, and by challenging data we refer to difficult, currently unsolved problem instances. On the one hand, this decision forced us to leave some interesting work developed throughout the PhD outside the thesis, such as the decomposition approaches proposed for the *Sport Teams Grouping Problem*

(Toffolo et al., 2017a) and for the *Leather Nesting Problem*[1]. On the other hand, however, considering only problems with benchmarked and challenging instances strengthens our conclusions concerning the superior performance of the decomposition-based algorithms proposed when compared against state-of-the-art algorithms.

Structure of the thesis

The remainder of this thesis is divided into two parts plus the final conclusions. The main difference between these parts lies in how subproblems are solved: while in Part I decomposition-based algorithms enforce optimal subproblem solutions, Part II evaluates algorithms which permit heuristic (possibly non-optimal) subproblem solutions. Each part investigates decomposition-based algorithms for three different problems. Part I focuses on scheduling and timetabling problems while Part II addresses logistics problems concerning routing and packing, considering both classic and real-world problems defined by industry. Figure 1.1 outlines this thesis by presenting an organizational chart correlating all nine chapters. Note that the research conducted during the PhD is not presented in chronological order. Instead, the manuscript was structured as to best support and explain the main thesis' idea.

Part I encompasses four chapters, beginning with Chapter 2 and ending with Chapter 5. Three different problems are addressed: the *Traveling Umpire Problem* (TUP), *Nurse Rostering Problem* (NRP) and *Project Scheduling Problem* (PSP). These problems have one common characteristic: the clear definition of a *time* dimension. This similarity is exploited to derive both exact and heuristic decomposition-based methodologies. In all of them, subproblems are solved to optimality. In total, up to four different approaches are proposed and evaluated for these problems, namely: (*i*) compact MIP formulation, (*ii*) Dantzig-Wolfe decomposition, (*iii*) exact algorithm employing decomposition-based bounds and (*iv*) decomposition-based heuristic. The resulting algorithms improve upon many results from the literature, redefining the state-of-the-art for these problems. These results motivate combining the primary principles from these algorithms in an attempt of producing a general decomposition-based solver, which is discussed in Chapter 5.

[1] An academic report concerning our work on the *Leather Nesting Problem* will be published online at a later date.

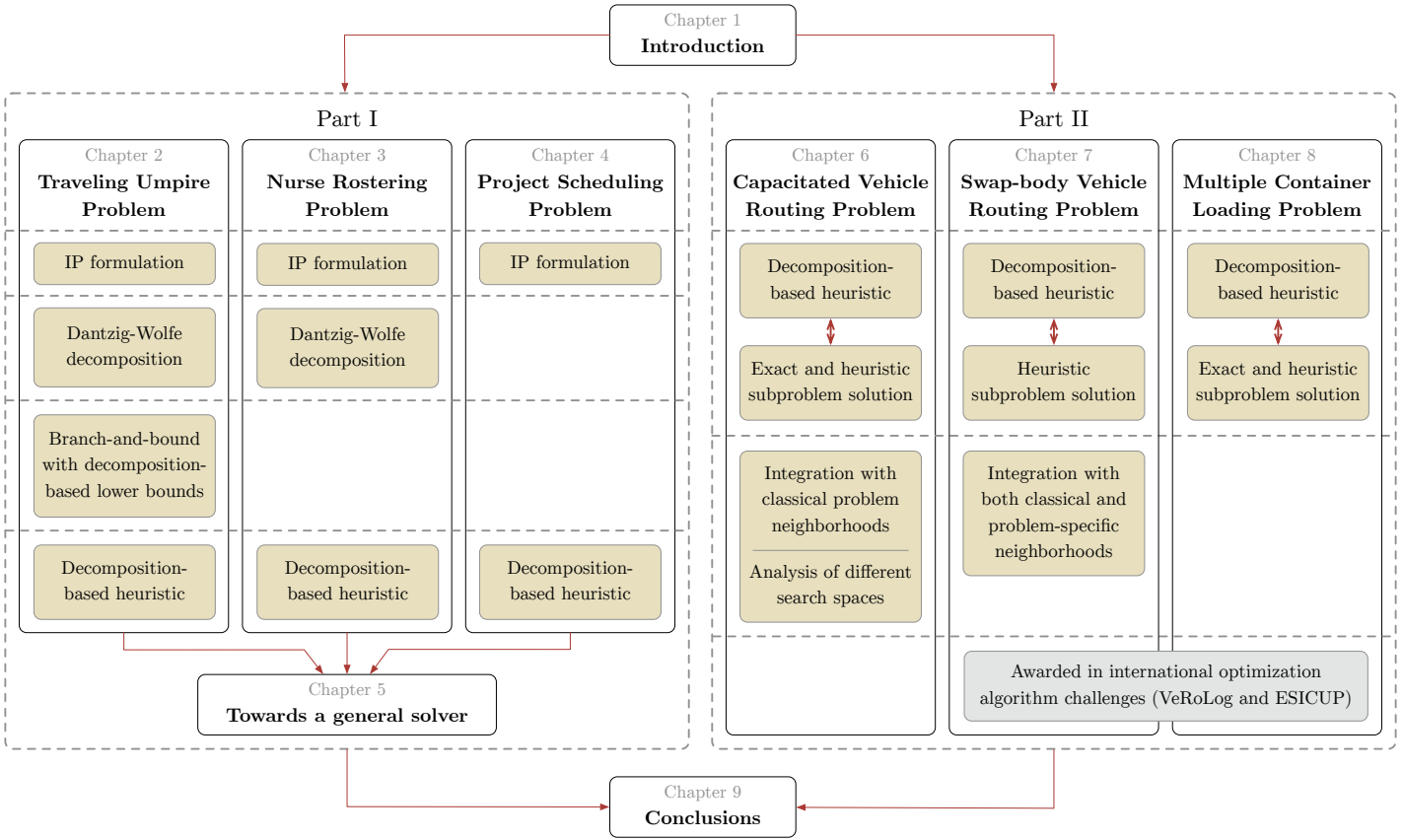


Figure 1.1: Thesis outline

Part II focuses on another three problems, all related to logistics: the *Capacitated Vehicle Routing Problem* (CVRP), the *Swap-Body Vehicle Routing Problem* (SBVRP), and the *Multiple Container Loading Problem* (MCLP). In contrast to Part I, only decomposition-based heuristics are considered. Different methodologies are proposed and evaluated, also resulting in competitive and state-of-the-art algorithms. Also differently from Part I, Part II evaluates decomposition-based heuristics in which subproblems are not necessarily solved to optimality. This results in algorithms which can be applied even when the permitted runtime is very restricted. Such was the case, for instance, for the problems discussed throughout Chapters 7 and 8. These two chapters present award-winning approaches that competed in international algorithm challenges, where a tight runtime limit was enforced.

Chapter 9 finishes this thesis by presenting the final conclusions and future research directions, while conducting a discussion concerning the role of decomposition approaches within algorithms for combinatorial optimization problems.

In the spirit of reproducible science (Kendall et al., 2016), whenever possible the source code associated with elements of this thesis was made available online, together with all instance and solution files.

Part I

Optimal subproblem solutions

Introduction to Part I

Part I addresses three problems that share one characteristic: they all contain the *time* dimension. Naturally, *time* is a dimension present in any timetabling or scheduling problem, defining a unique structure, which we exploit to decompose the problem deriving both exact and heuristic algorithms. In addition to time-structure based decompositions, classical decomposition methods are also evaluated for two of these problems as means of providing lower bounds to further evaluate the efficacy of the proposed decompositions. In all cases, subproblems resulting from the decompositions are optimally solved.

The three next chapters present both published and unpublished content. Rather than simply reproducing the published content, the notation, organization and nomenclature throughout the chapters were unified. Given that similar strategies were applied to the problems addressed by these chapters, a unified notation facilitates identifying the commonalities among the different decomposition approaches. Experiments including *Integer Programming* (IP) solvers were re-executed with the latest versions of the two best-performing[2] IP solvers currently: CPLEX and Gurobi. Results are also reported differently than within the papers, as to enable a sequential presentation of the primary principles behind the algorithms. Moreover, the chapters were crafted so that some independence is kept between them, with special attention to avoiding needless repetition.

Figure I presents the organization of the next chapters.

Chapter 2 investigates the Traveling Umpire Problem (TUP). Three different decomposition-based algorithms are proposed and evaluated: (*i*) a branch-and-price algorithm applying Dantzig-Wolfe's decomposition, (*ii*) a branch-and-bound algorithm employing decomposition-based lower bounds and, finally, (*iii*)

[2]According to the benchmark experiments reported by Mittelmann (2017).

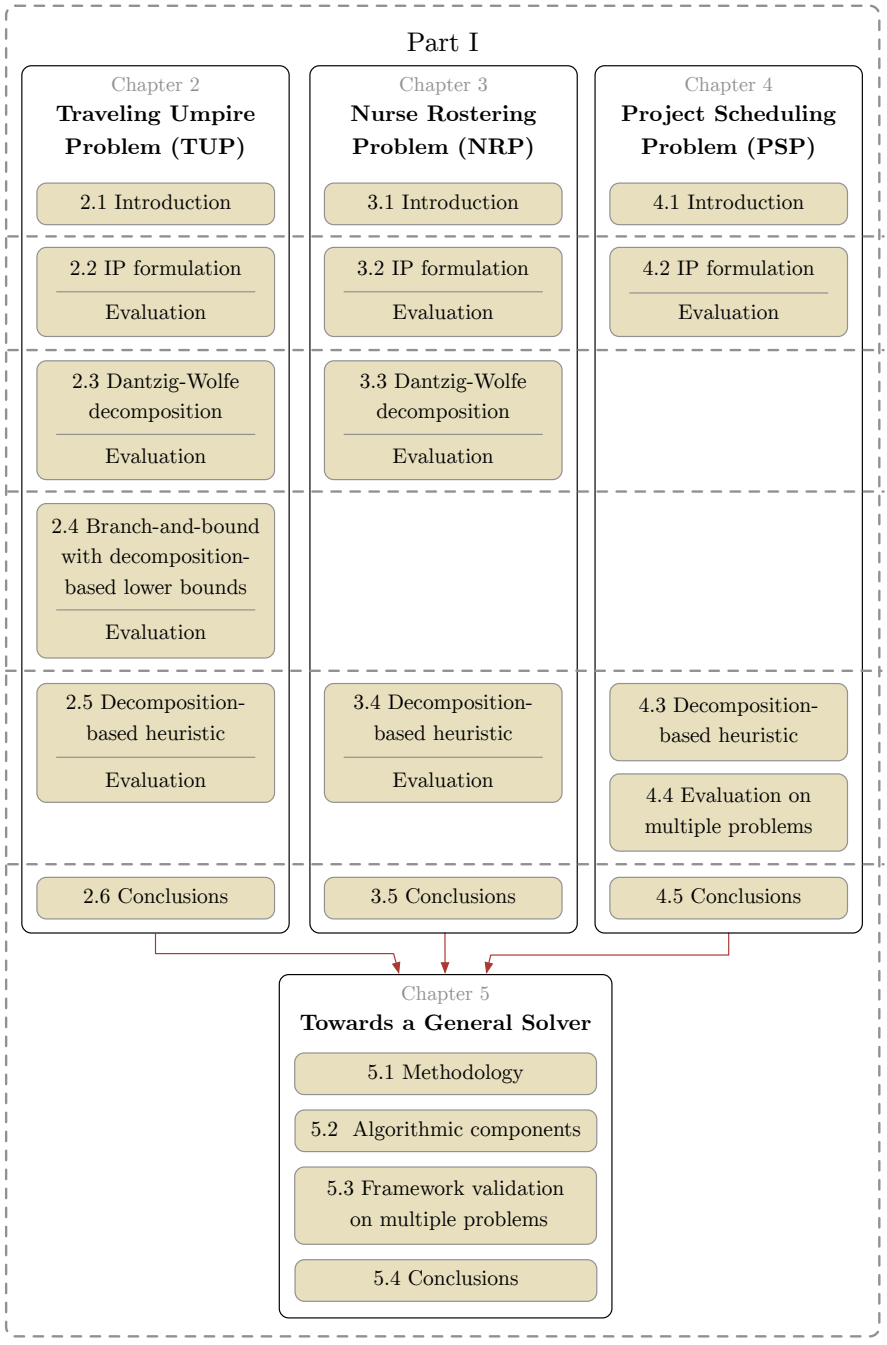


Figure I: Organization of Part I

a decomposition-based heuristic approach. These approaches resulted in various improved bounds and solutions, many of which proven optimal.

Chapter 3 focuses on the Nurse Rostering Problem (NRP). An exact decomposition-based algorithm and a local search method employing three decompositions are evaluated. Multiple benchmark instances had their best known solutions and bounds improved, evidencing the efficiency of the employed decompositions.

Chapter 4 investigates multiple versions of the Project Scheduling Problem (PSP). Constructive and local search decomposition-based heuristics are proposed and evaluated for the problem. Computational experiments considering two benchmark sets show the efficiency of the proposed approaches. Additionally, challenges imposed by specific problem properties are discussed together with strategies to circumvent them.

Finally, Chapter 5 proposes a general framework built upon the knowledge produced by Chapters 2, 3 and 4. The primary challenges are discussed and it is shown how the algorithms proposed by these chapters can be generalized. The general framework is validated with experiments considering not only the problems studied throughout Part I but also an additional one from the literature.

Chapter 2

Traveling Umpire Problem

This chapter addresses the *Traveling Umpire Problem* (TUP), an optimization problem in which umpires must be assigned to games in a double round robin tournament. The objective is to obtain a solution with minimum total travel distance over all umpires, while respecting hard constraints concerning assignment and sequencing.

As with all the problems addressed by this thesis, the TUP represents a challenging optimization problem. Up till this research, no general or dedicated algorithm was capable of solving all instances with 12 and 14 teams. This challenging status, combined with the problem's properties, such as its time structure, make the TUP the ideal test subject for the decomposition strategies we propose.

This chapter is the result of two publications – Toffolo et al. (2014)[1] and Toffolo et al. (2016c)[2] – combined with some more recent, unpublished research. It begins with an introduction and literature overview concerning the TUP, presented in Section 2.1. The problem is then formally described by an IP model in Section 2.2. Next, three different decomposition-based algorithms are introduced and evaluated for the problem: a classical approach, an exact

[1] **Toffolo, T. A. M.**, Van Malderen, S., Wauters, T., and Vanden Berghe, G. (2014). Branch-and-price and improved bounds to the traveling umpire problem. In *Proceedings of the 10th international conference on practice and theory of automated timetabling (PATAT 2014)*, pages 420–432, York, UK.

[2] **Toffolo, T. A. M.**, Wauters, T., Van Malderen, S., and Vanden Berghe, G. (2016). Branch-and-bound with decomposition-based lower bounds for the traveling umpire problem. *European Journal of Operational Research*, 250(3), 737–744.

one exploiting the problem's time structure and, finally, a heuristic algorithm. Section 2.3 presents the Dantzig-Wolfe's reformulation of the IP model presented in Section 2.2. The resulting formulation is solved by a tailor-made branch-and-price algorithm. Details of the methodology are described and then analyzed via computational experiments. In the quest of obtaining better bounds and solutions, Section 2.4 explores the TUP's time structure to derive a competitive branch-and-bound with decomposition-based lower bounds. All algorithmic components are evaluated and several new optimal solutions are obtained within short runtime. Rather than requiring more than 24 hours of computational time, as previously published approaches in the literature did, the proposed algorithm is capable of solving some 14-team instances in only a few seconds. However, larger instances prove challenging and remain unsolved. A decomposition-based heuristic is proposed throughout Section 2.5 to address these larger instances. Constructive and local search procedures derived from the decomposition approach are presented, resulting in improved solutions for large instances. Finally, Section 2.6 concludes this chapter by evaluating the different decomposition methods and highlighting our contributions, which include redefining the state-of-the-art for the TUP.

2.1 Introduction

The TUP is a sports timetabling problem concerning the scheduling of umpires (sport referees). The goal is to assign umpires to the matches of a tournament whose schedule is predetermined.

A double round robin tournament is considered, with $2n$ teams playing twice against each other – once at their home venue and once away. This results in a competition with $4n - 2$ rounds, each consisting of n matches. Such a tournament requires assigning n umpires to the games, with the objective to minimize their total travel distance. In order to obtain a fair schedule, hard Constraints $(a) - (e)$ are imposed:

(a) every match in the tournament is officiated by exactly one umpire;

(b) every umpire must work in every round;

(c) every umpire must visit the home venue of every team at least once;

(d) no umpire may be assigned to the same venue more than once in any q_1 consecutive rounds;

(*e*) no umpire may officiate games of the same team more than once in any q_2 consecutive rounds. This constraint is similar to the previous one, but also takes the 'away team' into consideration.

The values q_1 and q_2 range from 1 to n and 1 to $\lfloor \frac{n}{2} \rfloor$[3], respectively.

Since the introduction of the TUP by Trick and Yildiz (2007), who specifically address the Major League Baseball tournament, many exact and heuristic approaches have been developed. This initial work was extended (Trick and Yildiz, 2011) by a Benders' cuts guided large neighborhood search. Both papers also provided Integer Programming (IP) and Constraint Programming (CP) formulations for the problem. A greedy matching heuristic and a simulated annealing approach employing a two-exchange neighborhood were described by Trick et al. (2012). Trick and Yildiz (2012) developed a Genetic Algorithm (GA) with a locally optimized crossover procedure. A stronger IP formulation and a relax-and-fix heuristic were proposed by de Oliveira et al. (2014), who improved both lower and upper bounds. Wauters et al. (2014) improved solutions and lower bounds using an enhanced iterative deepening search with leaf node improvements (IDLI), an iterated local search (ILS) and a new lower bound methodology. Xue et al. (2015) presented two exact approaches to the TUP: a branch-and-bound algorithm relying on a Lagrangian relaxation for obtaining lower bounds and a branch-and-price-and-cut algorithm. The latter approach, which builds upon the branch-and-price algorithm we propose (Section 2.3), enabled two 14-team instances to be solved within the runtime limit of 48h.

Next, we formally present the TUP by means of an integer programming formulation.

2.2 Integer programming formulation

This section presents a flow formulation for the TUP based on the formulations presented by Trick and Yildiz (2007) and de Oliveira et al. (2014). A graph $G = (V, E)$ is given, in which each node represents a game and directed edges connect the nodes (games) of round r to the nodes of round $r + 1$. G also contains:

[3]Trick and Yildiz (2007) originally presented the parameters d_1 and d_2 such that $q_1 = n - d_1$ and $q_2 = \lfloor \frac{n}{2} \rfloor - d_2$, with $0 \leq d_1 < n$ and $0 \leq d_2 < \lfloor \frac{n}{2} \rfloor$.

- a *source node*, f, and directed edges connecting f to the nodes representing games of the first round;
- a *sink node*, l, and directed edges connecting the nodes representing games of the last round to l.

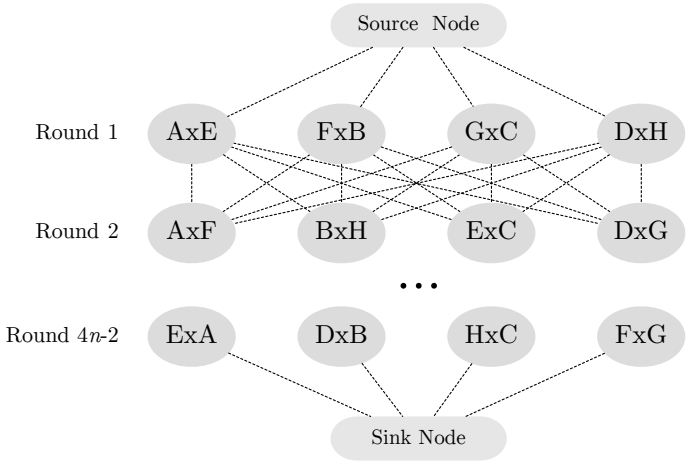


Figure 2.1: Graph $G = (V, E)$ representing an 8-team TUP instance

Figure 2.1 presents an example of this graph for an 8-team (or 4-umpire) instance. The formulation considers the following input data:

d_e : distance of directed edge e;

I : set of teams $\{1, ..., 2n\}$;

H_i : set of nodes where team i plays at home;

R : set of rounds $\{1, ..., 4n - 2\}$;

$Q'_{i,r}$: set of nodes (games) of team i playing at home in rounds $R \cap \{r, ..., r + q_1 - 1\}$;

$Q''_{i,r}$: set of nodes (games) of team i (home or away) in rounds $R \cap \{r, ..., r + q_2 - 1\}$.

U : set of umpires $\{1, ..., n\}$.

And the following variables:

$$x_{e,u} = \begin{cases} 1 & \text{if edge } e \text{ is selected for umpire } u \\ 0 & \text{otherwise} \end{cases}$$

Finally, let $\delta(I)$ and $\omega(I)$ denote the sets of edges that enter and exit the nodes in I, respectively. The problem's formulation is given by Equations (2.1)-(2.7).

Minimize:

$$\sum_{e \in E} \sum_{u \in U} d_e x_{e,u} \tag{2.1}$$

Subject to:

$$\sum_{e \in \delta(j)} \sum_{u \in U} x_{e,u} = 1 \qquad \forall j \in V \backslash \{\text{source, sink}\} \tag{2.2}$$

$$\sum_{e \in \delta(j)} x_{e,u} - \sum_{e \in \omega(j)} x_{e,u} = \begin{cases} -1 & \text{if } j \text{ is the source} \\ +1 & \text{if } j \text{ is the sink} \\ 0 & \forall j \in V \backslash \{\text{source, sink}\}, \end{cases}$$

$$\forall u \in U \tag{2.3}$$

$$\sum_{e \in \delta(H_i)} x_{e,u} \geq 1 \qquad \forall i \in I, u \in U \tag{2.4}$$

$$\sum_{e \in \delta(Q'_{i,r})} x_{e,u} \leq 1 \qquad \forall i \in I, r \in R, u \in U \tag{2.5}$$

$$\sum_{e \in \delta(Q''_{i,r})} x_{e,u} \leq 1 \qquad \forall i \in I, r \in R, u \in U \tag{2.6}$$

$$x_{e,u} \in \{0, 1\} \qquad \forall e \in E, u \in U \tag{2.7}$$

The objective, given by Equation (2.1), is to minimize the total distance traveled by the umpires. Constraints (2.2) ascertain that each game is officiated by exactly one umpire. Constraints (2.3) are flow preservation constraints, and together with the graph structure ensure that every umpire officiates exactly one game per round. If an umpire is at the location of a team in round r, the umpire must leave from this location to go to the next location in round $r + 1$.

This is also guaranteed by the flow preservation constraints. Constraints (2.4) state that every umpire must visit every location at least once during the season. Constraints (2.5) and (2.6) specify that every umpire must wait $q_1 - 1$ days to revisit the same home location and $q_2 - 1$ days to revisit the same team, respectively. Finally, Constraints (2.7) specify that the variables considered are binary.

2.2.1 Computational experiments

Formulation (2.1)-(2.7) was evaluated using the state-of-the-art solvers CPLEX 12.7 and Gurobi 7.5 on an Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz computer with 128Gb of RAM memory running Linux Ubuntu 16.04.2 LTS. Table 2.1 details the results obtained by these solvers. The characteristics of the model – number of variables (Vars), constraints (Cons) and non-zeros (NZs) – are presented together with the results obtained by the solvers: final lower (LB) and upper (UB) bounds, the computed optimality gap $\frac{UB-LB}{UB} \times 100$ and required runtime (Time). For compactness, runtimes are presented in different units. Moreover, note that the computational runtime was restricted to three hours.

Small instances with up to 10 teams were quickly solved by the solvers. The solvers were, however, unable to solve instances with more than 10 teams employing Formulation (2.1)-(2.7). On the one hand, CPLEX was capable of producing feasible solutions for most instances, while Gurobi could not find any feasible solution for instances with 16 teams. On the other hand, Gurobi produced better lower bounds on average for the evaluated instances than CPLEX.

Additional experiments concerning compact IP formulations for the TUP are reported by de Oliveira et al. (2014). They recently extended their formulation by adding valid inequalities and cutting planes, resulting in a branch-and-cut algorithm (de Oliveira et al., 2016). Despite improving lower bounds for large instances, IP formulations remain incapable of solving reasonable-sized TUP instances.

Table 2.1: Experiments with IP formulation

Instance	Model Dimensions			CPLEX 12.7				Gurobi 7.5			
	Vars	Cons	NZs	LB	UB	Gap	Time	LB	UB	Gap	Time
6 - 3,1	213	504	2019	14077	14077	0.0	0.1s	14077	14077	0.0	0.0s
6A - 3,1	213	504	2019	15457	15457	0.0	0.1s	15457	15457	0.0	0.0s
6B - 3,1	213	504	2019	16716	16716	0.0	0.1s	16716	16716	0.0	0.0s
6C - 3,1	213	504	2019	14396	14396	0.0	0.0s	14396	14396	0.0	0.0s
8 - 4,2	448	1244	5472	34311	34311	0.0	0.3s	34311	34311	0.0	0.2s
10 - 5,2	1325	2440	17650	48941	48942	0.0	22.6s	48938	48942	0.0	48.1s
10A - 5,2	1325	2440	17650	46548	46551	0.0	26.4s	46551	46551	0.0	29.8s
10B - 5,2	1325	2440	17650	45609	45609	0.0	5.0s	45609	45609	0.0	6.1s
10C - 5,2	1325	2440	17650	43149	43149	0.0	43.6s	43145	43149	0.0	3.2m
12 - 5,3	3096	4224	47592	91819	93965	2.3	3.0h	88426	94190	6.1	3.0h
14 - 7,3	6223	6755	106575	148668	177359	16.2	3.0h	151032	186184	18.9	3.0h
14 - 6,3	6223	6755	101675	150058	170196	11.8	3.0h	150370	170534	11.8	3.0h
14 - 5,3	6223	6755	96530	150165	164584	8.8	3.0h	149935	169260	11.4	3.0h
14A - 7,3	6223	6755	106575	141338	172515	18.1	3.0h	144259	188194	23.3	3.0h
14A - 6,3	6223	6755	101675	141813	163707	13.4	3.0h	143557	166542	13.8	3.0h
14A - 5,3	6223	6755	96530	143208	163840	12.6	3.0h	143765	159287	9.7	3.0h
14B - 7,3	6223	6755	106575	140568	170031	17.3	3.0h	144605	172994	16.4	3.0h
14B - 6,3	6223	6755	101675	142868	165740	13.8	3.0h	144052	164620	12.5	3.0h
14B - 5,3	6223	6755	96530	142075	157806	10.0	3.0h	143466	167170	14.2	3.0h
14C - 7,3	6223	6755	106575	140623	167983	16.3	3.0h	143048	190189	24.8	3.0h
14C - 6,3	6223	6755	101675	140097	162294	13.7	3.0h	142541	164291	13.2	3.0h
14C - 5,3	6223	6755	96530	142233	161820	12.1	3.0h	142421	159458	10.7	3.0h
16 - 8,4	11304	9568	224968	150938	-	-	3.0h	154988	-	-	3.0h
16 - 8,2	11304	9568	182568	142637	190640	25.2	3.0h	147950	-	-	3.0h
16 - 7,3	11304	9528	195280	143094	193649	26.1	3.0h	149468	-	-	3.0h
16 - 7,2	11304	9528	173696	137950	167491	17.6	3.0h	143474	-	-	3.0h
16A - 8,4	11304	9568	224968	161467	-	-	3.0h	159627	-	-	3.0h
16A - 8,2	11304	9568	182568	157054	187014	16.0	3.0h	159553	-	-	3.0h
16A - 7,3	11304	9528	195280	158235	207098	23.6	3.0h	163100	-	-	3.0h
16A - 7,2	11304	9528	173696	154095	177248	13.1	3.0h	157299	-	-	3.0h
16B - 8,4	11304	9568	224968	160253	-	-	3.0h	168404	-	-	3.0h
16B - 8,2	11304	9568	182568	156309	215056	27.3	3.0h	160977	-	-	3.0h
16B - 7,3	11304	9528	195280	156471	209344	25.3	3.0h	160750	-	-	3.0h
16B - 7,2	11304	9528	173696	154216	185852	17.0	3.0h	158604	-	-	3.0h
16C - 8,4	11304	9568	224968	158674	-	-	3.0h	167002	-	-	3.0h
16C - 8,2	11304	9568	182568	156446	195605	20.0	3.0h	160367	212078	24.4	3.0h
16C - 7,3	11304	9528	195280	158469	224646	29.5	3.0h	162645	-	-	3.0h
16C - 7,2	11304	9528	173696	155009	180532	14.1	3.0h	158982	194055	18.1	3.0h

2.3 Dantzig-Wolfe decomposition

The IP model presented during the previous section is reformulated by applying the Dantzig-Wolfe decomposition (Dantzig and Wolfe, 1960). The original problem is decomposed into a master problem and n pricing problems, one per umpire.

Figure 2.2 visualizes the structure of the *Linear Program* (LP) for a 6-team TUP instance considering the formulation presented in Section 2.2. This figure presents the coefficient matrix of the original LP (left image) and the same LP after sorting the rows and columns by umpire (right image). The dots indicate non-zero coefficients in the constraint matrix. The required block structure for the Dantzig-Wolfe decomposition is easily identifiable in the right image, where each square block forms a pricing problem containing the constraints and variables corresponding to a single umpire.

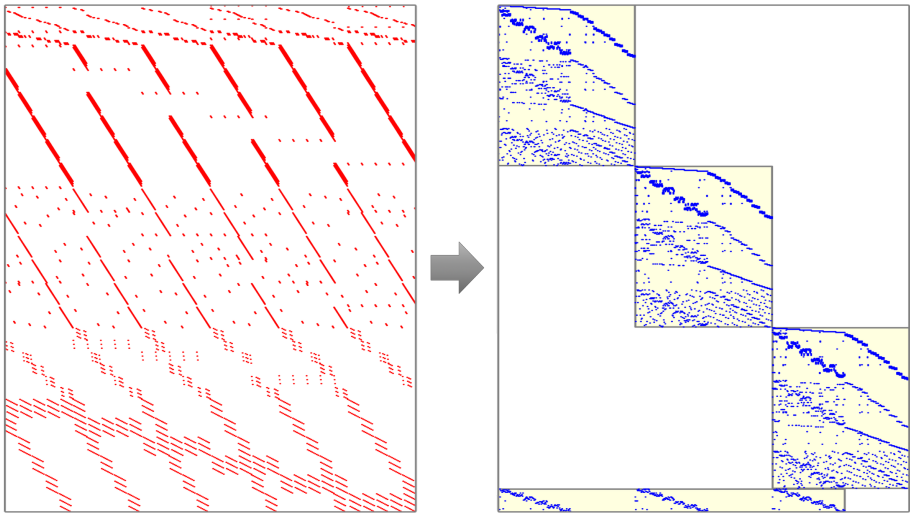


Figure 2.2: Representation of Dantzig-Wolfe's decomposition for the TUP

In Formulation (2.1)-(2.7), Constraints (2.3)-(2.7) are umpire-oriented and form the pricing problems. The remaining constraints, given by Equation (2.2), are the coupling (or linking) constraints. These correspond to the wide block at the bottom of the sorted LP in Figure 2.2.

The pricing problem essentially constitutes finding the optimum schedule for one umpire with the consideration of dual costs.

The master problem is a set partition problem whose formulation is given by Equations (2.8)-(2.11). Within this formulation, G is the set of games ($G = V \backslash \{\text{source, sink}\}$), Ω is the set of columns (possible schedules for the umpires), Ω_u represents the subset of Ω containing all columns of umpire $u \in U$, d_s is the cost (travel distance) of column $s \in \Omega$, λ_s is a binary variable indicating whether column $s \in \Omega$ is selected or not and, finally, $a_{j,s}$ is a binary coefficient

denoting whether game $j \in G$ is officiated in schedule (column) $s \in \Omega$ or not. Constraints (2.9) guarantee that only one column is selected per umpire while Constraints 2.10 are the coupling constraints inherited from the original problem (2.2), which ensure that each game in each round is officiated by exactly one umpire.

Minimize:

$$\sum_{s \in \Omega} d_s \lambda_s \tag{2.8}$$

Subject to:

$$\sum_{s \in \Omega_u} \lambda_s = 1 \qquad \forall u \in U \tag{2.9}$$

$$\sum_{s \in \Omega} a_{j,s} \lambda_s = 1 \quad \forall j \in G \tag{2.10}$$

$$\lambda_s \in \{0, 1\} \qquad \forall s \in \Omega \tag{2.11}$$

2.3.1 Column generation

The column generation approach (Lübbecke and Desrosiers, 2005, Vanderbeck and Wolsey, 2010) is applied iteratively. The linear relaxation of the master problem is solved first. At each iteration, the pricing problems are solved to obtain columns with negative reduced cost. A negative reduced cost column for umpire u is a column $s \in \Omega_u$ for which $v_u + \sum_{j \in G} a_{j,s} w_j > d_s$, where v_u and w_j represent the dual values corresponding to Constraints (2.9) and (2.10), respectively. If such columns are found, they are added to the master problem, which is subsequently re-solved. The algorithm continues until no column with negative reduced cost exists, whereupon the relaxation of the reduced master problem is solved.

Symmetry breaking

In order to speed up the pricing solver, the games assigned to the umpires in the first round are preallocated. This strategy, proposed by Yildiz (2008), reduces symmetry in the original problem, as otherwise the umpires would have similar coefficients in the constraint matrix. Preallocation is enforced by

adding Constraints (2.12) to Formulation (2.1)-(2.7). In these constraints, $I(i)$ is employed to represent the edge connecting the source node to the i-th game in the first round, with the games in lexicographic order.

$$x_{e,u} = 1 \qquad \forall u \in U,\ e = I(u) \tag{2.12}$$

Constraints (2.12) are umpire-oriented and may be included in the pricing problems of the column generation scheme. Including these constraints reduces the pricing problem size by one round.

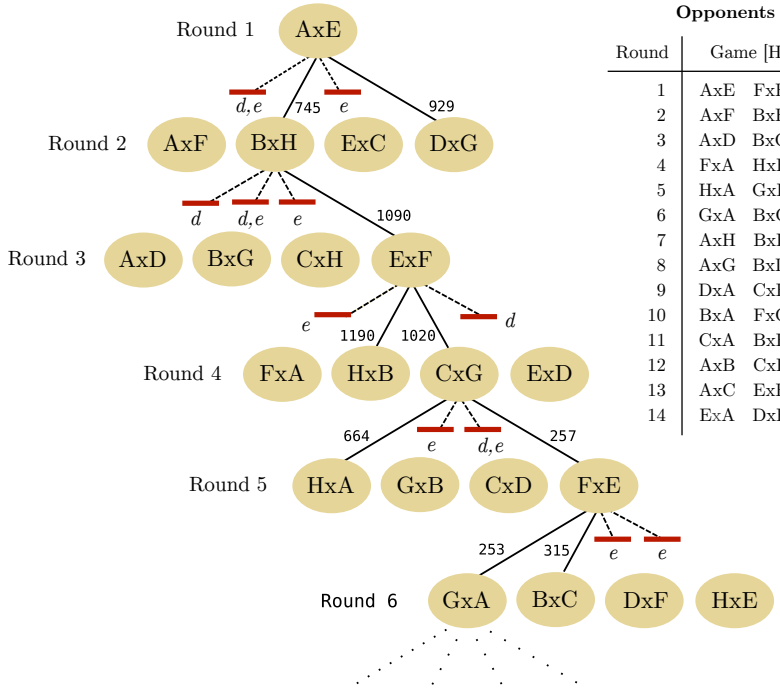
2.3.2 Solving the pricing problems

A branch-and-bound pricing solver is employed to produce columns with negative reduced cost. Beginning in the first round, the algorithm assigns games to the umpire, round after round until the last round. An assignment of a game to an umpire in a round is feasible if (i) the umpire did not visit the same location in the previous $q_1 - 1$ rounds and (ii) the umpire did not officiate any of the teams playing the game during the previous $q_2 - 1$ rounds. Whenever multiple games can be assigned in a round, the algorithm selects the assignment incurring the smallest increase in travel distance.

Figure 2.3 shows a snapshot of the branch-and-bound tree during its traversal for an 8-team (4-umpire) problem instance. The table inside the figure details the considered game schedule (opponents matrix). The example considers the pricing problem for the first umpire using parameter values $q_1 = 4$ and $q_2 = 2$. As detailed earlier within Section 2.3.1, the assignment for the first round is fixed.

The umpire may neither officiate game AxF nor game ExC in the second round due to Constraint (e) (presented in Section 2.1), since a game played by teams A and E has already been officiated by the umpire during the first round. Moreover, the umpire must not officiate game AxF due to Constraint (d), since the home location of team 1 has already been visited in the previous round. The only possibilities left in round two are games BxH and DxG. The branch-and-bound assigns the umpire to game BxH because the travel distance between the home location of teams A and B is smaller than the distance between the home locations of teams A and D.

If no valid assignment is possible in a certain round, the procedure returns to the previous round and instead selects the game with the second-lowest



Opponents Matrix

Round	Game [Home x Away]			
1	AxE	FxB	GxC	DxH
2	AxF	BxH	ExC	DxG
3	AxD	BxG	CxH	ExF
4	FxA	HxB	CxG	ExD
5	HxA	GxB	CxD	FxE
6	GxA	BxC	DxF	HxE
7	AxH	BxF	DxC	GxE
8	AxG	BxD	CxF	ExH
9	DxA	CxB	ExG	FxH
10	BxA	FxC	DxE	HxG
11	CxA	BxE	HxD	GxF
12	AxB	CxE	GxD	HxF
13	AxC	ExB	FxD	HxG
14	ExA	DxB	HxC	FxG

Figure 2.3: Pricing solver example for an 8-team TUP instance

travel distance. This procedure continues until a valid assignment has been found for the tournament's last round. If the resulting solution does not violate Constraint (c), it is feasible and its distance serves as an upper bound for pruning when exploring the remainder of the search tree.

Multiple strategies exist to prune unfavorable parts of the search tree. First, the branch-and-bound algorithm prunes the parts of the search tree where no optimum solution can reside based on the travel distance lower and upper bounds. Once the branch-and-bound algorithm has obtained a feasible solution, it serves as an upper bound concerning the minimum travel distance of the umpire.

For each game in every round, a shortest path exists to any of the games in the last round. The shortest path serves as a lower bound for the branch-and-bound procedure. When attempting to assign a game in a round, the algorithm evaluates whether the current travel distance together with the lower bound exceeds the currently best known upper bound. If so, the branch-and-bound

need not consider that assignment anymore, since it will not improve the current upper bound.

It is impossible to evaluate Constraint (c) before a complete path has been generated for the umpire. Nevertheless, a second pruning strategy is possible. If the number of unvisited home locations exceeds the remaining number of rounds within a certain round, it is impossible to obtain a solution satisfying Constraint (c), given the assignments throughout the previous rounds. The branch-and-bound algorithm should therefore return to a previous round and explore alternative assignments.

2.3.3 Branch-and-price

Section 2.3.1 presented the column generation scheme. Since this algorithm only solves the LP-relaxed version of the problem, it may be necessary to branch on fractional variables to obtain an integer solution. When this situation occurs, branch-and-price (Barnhart et al., 1998) is applied, which is a variant of branch-and-bound where the relaxation is solved by column generation in each node of the search tree.

The branch-and-price algorithm branches on the variables $x_{e,u}$ from the original formulation. Since the branching tree is too large, we consider two different strategies for branching, each one pursuing a different goal. The first strategy seeks to provide good lower bounds by conducting a *Best-First Search* (BFS) in the branching tree. The second strategy executes *Depth-First Search* (DFS) and focuses on obtaining integral solutions.

In each iteration, the BFS strategy selects variables for branching based on the following criterion: variables with the most fractional value concerning the earliest available round are selected first. Fixing variables of the earliest available round impacts the performance of the pricing solver considerably. The pricing solver constructs the solution from the first to the last round, in lexicographical order. Hence, if a variable from the last round was selected first, runtime could be wasted exploring infeasible subtrees. Since the fixation of a variable renders several subtrees infeasible, it is better to detect the infeasibility as soon as possible during the branching process. Otherwise the detection of infeasible subtrees is delayed, consuming a considerable amount of processing time.

The DFS strategy aims to obtain feasible solutions as soon as possible. Therefore,

in each node the variable with the least fractional value of the earliest possible round is selected to be branched first. By proceeding in a depth-first search manner, the fixations are directed to iteratively build a feasible solution employing the information provided by the column generation.

2.3.4 Computational experiments

The developed approach employs SCIP/GCG (Achterberg, 2009). This open source framework provides a well-structured platform for developing branch-and-price algorithms. The branching scheme, node rules and pricing solver were coded in Java, using Java Native Interface (JNI) to exchange information between Java and C. CPLEX was employed to solve the linear relaxation of the Restricted Master Problem.

The experiments were executed on an Intel(R) Xeon(R) CPU E5-2650 @ 2.60GHz computer with 128Gb of RAM memory running Linux Mint 16. CPLEX version 12.6 and Java Virtual Machine 1.7 were used.

We developed an automated benchmark website for TUP[4] where the considered instances are available, together with all solutions and bounds produced. Results are here compared against the best known solution values in the literature at the time of the experiment, which includes bounds and solutions from Trick and Yildiz (2007, 2011, 2012, 2013), Trick et al. (2012), de Oliveira et al. (2014) and Wauters et al. (2014). Note that Xue et al. (2015) builds upon the research here presented and is therefore not considered in this comparison. Their results are mentioned in the next sections.

The discussion of experiments focuses on two evaluations: the dual bounds obtained by the BFS branching scheme in the branch-and-price and the feasible solutions obtained by the DFS branching strategy. The results obtained by both strategies are presented in Table 2.2. Instance names are abbreviated, such that '12-7,2' represents instance umps12 with $q_1 = 7$ and $q_2 = 2$. The table details:

- the best lower bounds (LB^*) and solution values (UB^*) found in the literature, followed by the derived optimality gap;
- the lower bounds (LB_0) obtained by column generation (in the root node) and the required runtime in seconds;

[4]https://benchmark.gent.cs.kuleuven.be/tup

- the lower bounds (LB) obtained with the BFS strategy, followed by the total runtime to obtain the bound;
- the solution values (UB) obtained with the DFS strategy, followed by the runtime to obtain the solution;
- and the resulting (often improved) optimality gap for each instance.

Note that gaps are calculated as $\frac{UB-LB}{UB} \times 100$.

The results obtained with Formulation (2.1)-(2.7) are not competitive and were not included in the table. Additionally, the BFS strategy could not generate feasible solutions for any instance with more than 10 teams within the runtime limit and its solution values were consequently omitted. Results for smaller instances are not reported either, since they were easily solved in a few seconds.

Table 2.2 shows that the column generation (root node) alone already improved 8 best known lower bounds. By applying the branch-and-price with BFS, 15 other instances had their best known dual bound improved. This result corroborates the expected strong bounds from column generation.

Table 2.2 also enables assessing the influence of the pricing solver on the total processing time of the column generation. Consider, for example, the difference in time required for solving the column generation in the root bound (given by column LB_0) for instances '16A-7,2' and '16A-7,3'. Column generation for instance '16A-7,2' required much more computation time than for '16-A-7,3'. This is primarily due to the value $q_2 = 2$ for the first instance, which is less constrained than the second one, within which $q_2 = 3$. Small q_2 values negatively impact the pricing solver's performance, since it provides fewer pruning opportunities.

The DFS strategy within the branch-and-price improves upon five best known solutions, despite the total available runtime being only three hours. Considering that the developed approach tends to perform better on more constrained instances, one would presumably expect better results for the highly-constrained '16-8,4', '16A-8,4', '16B-8,4' and '16C-8,4' instances. It is important to note that the bounds (and solutions) for these benchmark instances have been repeatedly updated over the years. The branch-and-price was unable to find any feasible solution after three hours of processing time. This result, together with the fact that there are no known solution for these instances, motivated us to investigate the strong indication that they may be infeasible. The infeasibility of these instances is proven in Section 2.4.5.

Table 2.2: Branch-and-price results for the TUP

Inst.	Best results in literature			Column generation		BFS strategy		DFS strategy		Best gap
	LB*	UB*	Gap	LB_0	Time(s)	LB	Time(s)	UB	Time(s)	
14 - 7,3	159797	164440	2.8%	156439.3	42	157812.8	10500	166942	547	2.8%
14 - 6,3	156551	159505	1.9%	154439.9	41	155570.4	10560	159808	2462	1.9%
14 - 5,3	153066	155439	1.5%	152941.3	50	153759.6	10740	155392	1215	1.1%
14A - 7,3	153199	158760	3.5%	149992.7	41	151243.5	10740	160856	7500	3.5%
14A - 6,3	150998	153216	1.4%	148168.7	46	149285.4	10680	154637	2814	1.4%
14A - 5,3	148299	149331	0.7%	147097.5	48	147966.4	10620	150386	4110	0.7%
14B - 7,3	151059	157884	4.3%	149767.0	44	151165.8	10620	162677	1560	4.3%
14B - 6,3	149267	152740	2.3%	148243.9	50	149208.6	10620	155817	5662	2.3%
14B - 5,3	147534	149621	1.4%	146846.2	56	147638.3	10800	149866	1579	1.3%
14C - 7,3	151581	154913	2.2%	148613.2	45	150101.6	10380	159815	6072	2.2%
14C - 6,3	148728	150858	1.4%	146774.6	48	147820.0	10320	152696	6877	1.4%
14C - 5,3	146764	149662	1.9%	145794.4	50	146622.1	10620	149482	9219	1.8%
16 - 8,4	185939	-	-	184187.6	172	193457.1	10260	-	-	-
16 - 8,2	151481	160705	5.7%	155045.2	7092	155045.2	7092	161999	9919	3.5%
16 - 7,3	158480	168860	6.1%	158257.4	10500	158586.0	10500	170293	7800	6.1%
16 - 7,2	147138	153978	4.4%	148341.8	10102	148341.8	10102	-	-	3.7%

(continued on next page)

Table 2.2 continued: Branch-and-price results for the TUP

Inst.	Best results in literature			Column generation		BFS strategy		DFS strategy		Best gap
	LB*	UB*	Gap	LB_0	Time(s)	LB	Time(s)	UB	Time(s)	
16A - 8,4	185119	-	-	198969.7	172	200648.5	10260	-	-	-
16A - 8,2	162788	172966	5.9%	166575.5	5403	166624.1	10410	171882	8017	3.1%
16A - 7,3	172964	179960	3.9%	170575.1	371	172420.1	10560	187686	2980	3.9%
16A - 7,2	161640	164620	1.8%	161571.2	7476	161571.2	7476	165766	9759	1.8%
16B - 8,4	208418	-	-	207505.4	202	209346.5	10440	-	-	-
16B - 8,2	167768	180888	7.3%	169363.4	5162	170092.6	10162	180728	10717	5.9%
16B - 7,3	173023	181565	4.7%	170632.5	880	172058.0	10560	186429	1378	4.7%
16B - 7,2	164012	170194	3.6%	163539.7	9021	163649.6	11298	-	-	3.6%
16C - 8,4	188561	-	-	200682.6	234	205643.8	10380	-	-	-
16C - 8,2	166001	180221	7.9%	168783.6	7380	168783.6	7380	179939	9286	6.2%
16C - 7,3	171377	184181	7.0%	171216.0	449	171767.6	10740	187310	2235	6.7%
16C - 7,2	163305	169184	3.5%	163850.8	10578	163850.8	10578	-	-	3.2%

2.4 Branch-and-bound with decomposition-based lower bounds

The Dantzig-Wolfe decomposition proposed earlier within Section 2.3 for the TUP was improved by Xue et al. (2015), who included cuts and employed a labeling algorithm to solve the pricing problem. Consequently, and for the first time, a 14-team instance was solved. However, only the two least constrained 14-team instances were solved, requiring a prohibitive runtime in practice (around 48 hours). The question remains: how to solve such instances in a reasonable amount of time?

In this section, we propose a completely different algorithm that decomposes the TUP into subproblems by exploring the time structure of the problem. Subproblem solutions are combined to produce strong bounds, which are employed within a branch-and-bound algorithm. The resulting algorithm solved all 14-team instances. Rather than requiring over one day of runtime (Xue et al., 2015), the 14-team instances were solved within a few minutes, or even seconds, of runtime. This corroborates our hypothesis that algorithms which employ decomposition approaches exploiting straightforward structural problem characteristics can outperform classical heuristic and exact algorithms.

2.4.1 Branch-and-bound

Building on the branch-and-bound procedure established by Land and Doig (1960), we introduce a specialized decomposition-based algorithm to the TUP. This algorithm considers the same graph $G = (V, E)$ presented for the integer programming formulation in Section 2.2. Beginning from the first round, the branch-and-bound algorithm assigns games to umpires, one at a time and round by round, until the sink node is reached. An assignment of a game to an umpire in a round is feasible if (i) the umpire did not visit the same location in the previous $q_1 - 1$ rounds and (ii) the umpire did not officiate any of the teams during the previous $q_2 - 1$ rounds. Whenever it is possible to assign multiple games to a single umpire in one round, the algorithm greedily selects the assignment incurring the smallest increase in travel distance. When ties occur, games are sorted lexicographically (using home team names, for instance). Note that a similar strategy was employed within the branch-and-bound algorithm described in Section 2.3.2.

If no valid assignment exists for an umpire in a certain round, the procedure backtracks to the previous allocation and selects the next game in the ordered list of games in the round. This procedure continues until the sink node is reached for all umpires. If the resulting solution does not violate Constraint (c), it is feasible and its total distance serves as an upper bound. This upper bound is, together with the calculated lower bounds, employed to prune the parts of the search tree where no optimum solution can reside.

Whenever a new feasible solution is obtained, a local search procedure is applied to improve its quality. Even if the obtained solution is infeasible, meaning it does not satisfy Constraint (c), a local search algorithm is executed which attempts to first restore feasibility and then to improve the quality of the resulting solution.

Figure 2.4 presents an example of the branch-and-bound optimizing an 8-team instance where $q_1 = 3$ and $q_2 = 2$. It illustrates the algorithm currently defining which game Umpire 1 will officiate after game BxH. The games AxD, BxG and CxH are cut from the search tree in the current stage, as they would lead to infeasible solutions. The first two games would violate Constraint (d) while the second and third would violate Constraint (e). Thus, the only option for Umpire 1 in the next round is to officiate game ExF.

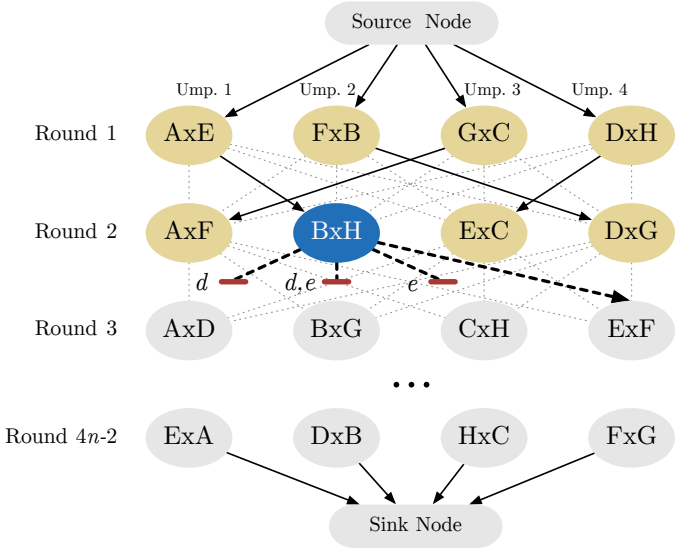


Figure 2.4: Branch-and-bound illustration for an 8-team TUP instance

Symmetry breaking

The symmetry breaking strategy discussed in Section 2.3.1 is also applied here: game assignments of the first round are fixed, as otherwise the umpires would be identical and introduce redundant subtrees. Therefore, Constraints (2.12) are also considered within the branch-and-bound algorithm.

Preprocessing the graph

Another way to speed up the branch-and-bound is by removing edges which violate one of the constraints (de Oliveira et al., 2014). If $q_1 > 1$, then all edges connecting games in the same venue are removed. Likewise, if $q_2 > 1$ then edges connecting games of the same team are also removed. For instance, the edges connecting games BxH to BxG and BxH to CxH in Figure 2.4 would be removed by this preprocessing procedure.

Additional pruning rules

Constraint (c) – every umpire should visit the home of every team at least once – is employed as an additional pruning rule. If the number of unvisited home locations for an umpire in a certain round exceeds the remaining number of rounds, given the assignments in the previous rounds, it is impossible to obtain a solution satisfying Constraint (c). The branch-and-bound algorithm should consequently backtrack and explore alternative assignments. This pruning strategy is irrelevant for the last round, however, because the maximum number of unvisited home locations for an umpire would be one. In this case, the local search heuristic can be applied to restore feasibility, potentially resulting in an improved upper bound.

Local search procedure

A local search procedure is applied to feasible and infeasible solutions obtained by the branch-and-bound to quickly improve the upper bound. The local search performs a steepest descent search employing a matching neighborhood. The neighborhood calculates the matching for each round in the solution. With a view to minimizing infeasibility, invalid assignments incur additional costs to the matching problems. For every umpire u and every game g in a given round

r, the matching cost $C_{u,g}$ for assigning game g to umpire u is a combination of two deltas presented by Equation (2.13):

$$C_{u,g} = \Delta d_{u,g} + \rho \ \Delta v_{u,g} \tag{2.13}$$

where $\Delta d_{u,g}$ is the difference between the distance of the new and the current assignment, $\Delta v_{u,g}$ is the difference between the number of hard constraint violations in the new and the current assignment, and ρ is a value sufficiently large such that any variation of $\Delta v_{u,g}$ is more significant than any possible value for $\Delta d_{u,g}$.

Pseudo-code of the branch-and-bound algorithm

The pseudo-code of a recursive version of the branch-and-bound algorithm is presented in Algorithm 2.1. This algorithm should initially be executed as BranchBound($\emptyset, 1, 1$), thereby receiving the following parameters: (i) an empty solution, (ii) the first umpire and (iii) the first round. Initially, the umpire and round to be analyzed in the next iteration are determined (lines 1-2) and a sorted list $\mathcal{L}$ of possible allocations for umpire u in round r is constructed (line 3). The algorithm then iterates through list $\mathcal{L}$ (line 4), pruning the allocation when possible (line 5) or adding it to the solution (line 6). If other umpire-game allocations remain unexplored, the procedure is recursively executed for the next umpire and/or round (lines 7-8). Once the solution is complete (line 9), meaning all the games have umpires assigned, the local search procedure described in Section 2.4.1 is executed (line 10). If the resulting solution S' is better than the best found (S^*), then S^* is updated (lines 11-12). Finally, the current allocation is removed in line 13.

2.4.2 Decomposition-based lower bounds

A good lower bound is a basic requirement for an efficient branch-and-bound minimization algorithm. The branch-and-bound developed for the TUP employs a decomposition approach for quickly calculating strong lower bounds. Initially, the problem is decomposed into $|R|-1$ subproblems. Each of these subproblems consists of exactly two consecutive rounds, which enables calculating a lower bound for these rounds. Next, the decomposition is modified by iteratively increasing the size of the subproblems by one round.

Algorithm 2.1: Branch-and-bound algorithm

```
Let S* be a global variable representing the best solution, initialized as S* ← ∅
Input: Solution S, umpire u and round r
BranchBound(S, u, r)
1   u+ ← (u mod n)+1 // next umpire to analyze
2   r+ ← r + 1 if u = n and r otherwise // next round to analyze
3   L ← sorted list of feasible allocations in S for umpire u in round r
4   foreach a ∈ L do
5       if allocation a cannot be pruned away then
6           S ← S ∪ {a}
7           if S is not complete then
8               BranchBound(S, u+, r+)
9           else
10              S' ← LocalSearch(S)
11              if S* = ∅ or S' is better than S* then
12                  S* ← S'
13          S ← S \ {a}
```

Initial lower bounds

The first subproblems contain exactly two consecutive rounds and consist of finding a set of trips (edges) for the umpires to officiate the games in these rounds. The objective thus is to find a feasible edge set which connects the subproblem's rounds. This subproblem is a simple assignment problem and can be solved efficiently using the Hungarian Algorithm (Munkres, 1957). Constraint (c) is ignored in the subproblems.

Figure 2.5 shows an example of a subproblem with two rounds, r and $r+1$. Four games are to be officiated by four umpires in each round. The solution is a *matching*. Note that edges violating Constraints (d) and (e) were removed from the graph. The preprocessing procedure presented in Section 2.4.1 avoids analyzing these infeasible connections.

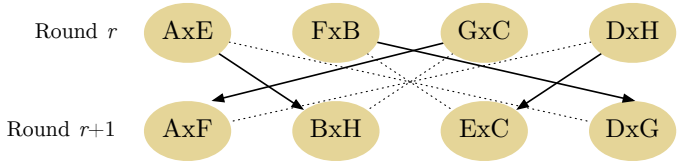


Figure 2.5: Example of a (solved) TUP subproblem

The sum of the distances of all $|R|-1$ matchings is a valid lower bound for the problem. It equals the minimum-cost flow with node capacity (equal to 1) for the original problem. This network flow problem is a relaxation of the TUP, obtained by removing Constraints (2.4)-(2.6) from Formulation (2.1)-(2.7).

The lower bound obtained is employed by the branch-and-bound procedure for pruning. Let m_r be the value of the matching between the consecutive rounds r and $r+1$. The initial lower bounds LB_{r_1,r_2} for the cost between rounds r_1 and r_2, $r_1 < r_2$, are given by Equation (2.14).

$$LB_{r_1,r_2} = \sum_{r \in R: r_1 \leq r < r_2} m_r \tag{2.14}$$

Strengthening the lower bounds

The matchings provide valid, but relatively weak lower bounds. In order to improve the quality of the bounds, subproblem sizes are incremented. The main principle is that subproblems with more rounds consider more constraints and, therefore, the obtained bounds tend to be stronger. However, by increasing the number of rounds, the subproblems become considerably more difficult to solve. For instance, the subproblem with $|R|$ rounds is equivalent to the original TUP without Constraint (c).

Subproblems with three or more rounds are solved by the very same branch-and-bound presented in Section 2.4, except for the evaluation of Constraint (c), which here is irrelevant. Therefore, the pruning rules presented in Section 2.4.1 are not considered.

Lower bounds computed previously are employed for pruning incrementally larger subproblems. Figure 2.6 shows an example of a subproblem containing four rounds of an 8-team instance with $q_1 = 3$ and $q_2 = 2$. While solving this subproblem, the bounds obtained from smaller subproblems, with two and three rounds, are utilized to prune the search tree. Note that the algorithm ensures that smaller subproblems, which may provide bounds, are solved before the enclosing 'larger' subproblems. For instance, in the example of Figure 2.6, the subproblems with rounds $\{r+2, r+3\}$ and $\{r+1, r+2, r+3\}$ are solved before the subproblem with rounds $\{r, r+1, r+2, r+3\}$.

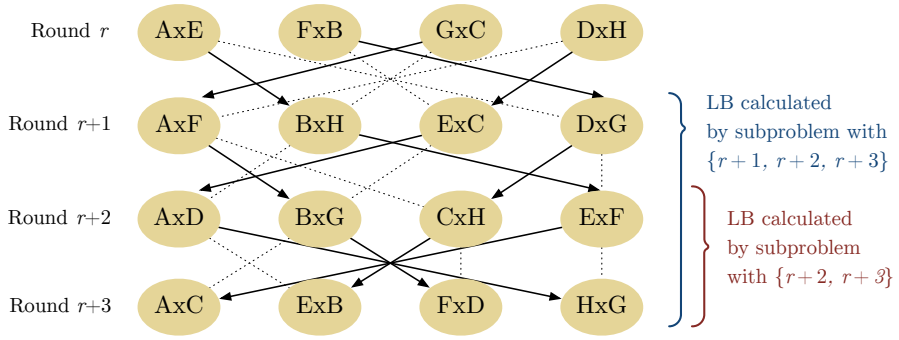


Figure 2.6: Lower bounds example for a TUP subproblem with four rounds

Lower bounds propagation

One of the key advantages of the decomposition approach presented is how the solution of one subproblem may be employed to strengthen a number of lower bounds. Strengthening is achieved with a simple bound propagation procedure.

Consider the subproblem illustrated by Figure 2.6, which includes rounds r, $r+1$, $r+2$ and $r+3$. The solution distance of this subproblem provides a new bound, $LB^*_{r,r+3}$. This bound is applied to improve all values of LB_{r_1,r_2} with $r_1 \leq r$ and $r_2 \geq r+3$. Equation (2.15) shows how these bounds are improved. In this equation, k represents the difference between the subproblem first and last rounds ($k = 3$ in the example of Figure 2.6). Note that for any r, $LB_{r,r} = 0$.

$$LB_{r_1,r_2} = \max(LB_{r_1,r_2}, LB_{r_1,r} + LB^*_{r,r+k} + LB_{r+k,r_2}) \tag{2.15}$$

Equation (2.15) is applied to all pairs of rounds (r_1, r_2), with $r_1 \in \{1, ..., r\}$ and $r_2 \in \{r+k, ..., |R|\}$, possibly improving several bounds.

Pseudo-code of the lower bounds computation algorithm

Algorithm 2.2 presents the lower bounds computation procedure. The algorithm begins by setting all values of the matrices S and LB to zero (lines 1-2). The first for-loop (lines 3-6) calculates the initial lower bounds for all pairs of rounds using the values of the matchings between every two consecutive rounds. The next for-loop (line 7) is responsible for solving subproblems with more than two rounds. The difference between the subproblem first and last rounds (k) begins

at 2 and increases until $|R|-1$, thereby implying the subproblem size begins at 3 and increases until it equals $|R|$. Line 8 specifies the first round of the current subproblem (r). The subproblems are solved in the while-loop (line 9). Some subproblems require that lower bounds are calculated beforehand. Lines 10-11 guarantee this requirement, by first solving subproblems beginning in round $r' = r + k - 2$ and decrementing until round $r' = r$. To avoid recalculation, a subproblem with rounds $\{r', ..., r + k\}$ is solved only if $S_{r',r+k} = 0$ (line 10). The new bounds are subsequently propagated to all pairs of rounds which can benefit from the improved values (lines 12-13). Finally, the first round r of the next subproblem is updated (line 14).

Algorithm 2.2: Lower bounds computation algorithm

```
   Let S be an |R| × |R| matrix with solution values for the subproblems
   Let LB be an |R| × |R| matrix with lower bounds for all pairs of rounds
   CalculateLBs()
1  |  S ← 0_{|R|×|R|}
2  |  LB ← 0_{|R|×|R|}
3  |  foreach r ∈ {|R| − 1, ..., 1} do
4  |  |  S_{r,r+1} ← value of matching between rounds r and r + 1
5  |  |  foreach r_2 ∈ {r + 1, ..., |R|} do
6  |  |  |  LB_{r,r_2} ← S_{r,r+1} + LB_{r+1,r_2}
7  |  foreach k ∈ {2, ..., |R| − 1} do
8  |  |  r ← |R| − k
9  |  |  while r ≥ 1 do
10 |  |  |  foreach r' ∈ {r + k − 2, ..., r} | S_{r',r+k} = 0 do
11 |  |  |  |  S_{r',r+k} ← solution value of subproblem {r', ..., r + k}
12 |  |  |  |  foreach r_1 ∈ {r', ..., 1}, r_2 ∈ {r + k, ..., |R|} do
13 |  |  |  |  |  LB_{r_1,r_2} ← max(LB_{r_1,r_2}, LB_{r_1,r'} + S_{r',r+k} + LB_{r+k,r_2})
14 |  |  |  r ← r − k
```

Algorithm 2.2 is executed in parallel during the branch-and-bound procedure. Two threads are employed by the final algorithm: one to calculate lower bounds (Algorithm 2.2) and another to compute upper bounds (Algorithm 2.1). Executing both algorithms sequentially would require solving all the subproblems in advance, potentially leading to a waste in computational time. Addressing both lower and upper bounds in parallel avoids this situation, since the algorithm stops whenever optimality is proven, a situation which may be achieved before all subproblems are solved. One possible disadvantage is that the algorithm's execution is not deterministic, since information is exchanged between threads.

2.4.3 Pruning strategies

The branch-and-bound procedure prunes away nodes and reduces the search tree based on the lower bounds. Assume a feasible solution with cost UB is given, and that the branch-and-bound is analyzing the node corresponding to the allocation of a specific game to an umpire in round r. Let $LB_{r,|R|}$ be the lower bound for all allocations after round r and let C be the sum of the distances of all the allocations in the current solution plus the distance of the allocation currently being analyzed. The search tree derived from the current allocation may be pruned if $C + LB_{r,|R|} \geq UB$.

This strategy, however, has one drawback. If remaining umpires are to be assigned in round r, the number of pruning opportunities may be limited because bound $LB_{r,|R|}$ only considers allocations of rounds after r, while allocations for round r itself are pending. The following procedure is proposed to address this drawback and improve the pruning strategy:

1. A subgraph is derived containing:
   - the games of round $r-1$ of umpires not yet allocated in round r,
   - the games of round r with allocations pending,
   - the edges connecting games of these two sets.
2. A matching problem defined by the subgraph is solved.

This "partial" matching provides a value m that may be employed to improve the lower bound, enabling the pruning away of a branch whenever $C+LB_{r,|R|}+m \geq UB$.

Figure 2.7 illustrates this procedure. The allocation of game CxH to Umpire 2 is being considered for round 3. Note that game ExF of round 3 was already assigned to Umpire 1. In this case, the "partial" matching problem consists of games AxF, ExC, AxD and BxG and the edges connecting these games. Let m be the solution cost of this matching problem. The allocation of CxH to Umpire 2 in the current solution is ignored if $C + LB_{3,|R|} + m \geq UB$, where C is the total allocation distance in the current solution plus the cost of allocating game CxH to Umpire 2 after game DxG.

The "partial" matching procedure introduces considerable computational overhead to the branch-and-bound algorithm. To circumvent this issue, a memoization scheme (Michie, 1968) is employed to avoid the recalculation of

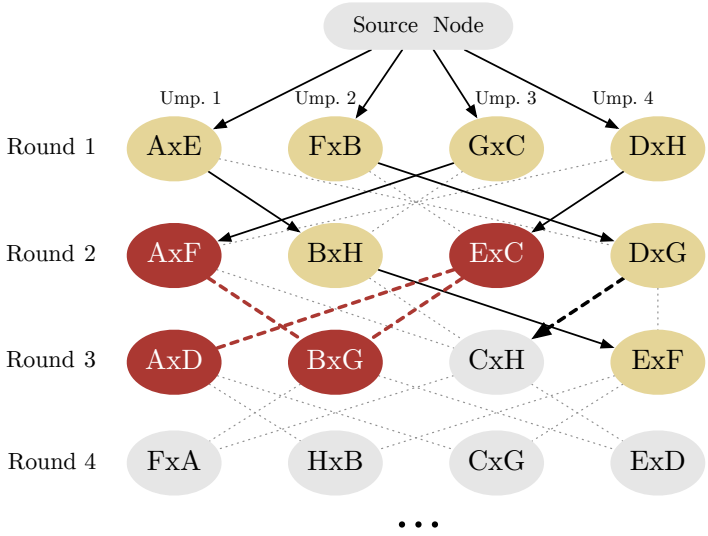


Figure 2.7: "Partial" matching problem example

previously-solved matching problems, reducing this computational overhead to an acceptable level.

2.4.4 Parallelization

The branch-and-bound previously described already employs two threads: one to calculate lower bounds and another to compute upper bounds. However, the algorithm contains properties which enable additional parallelization:

1. most subproblems solved during the computation of lower bounds are independent and may be processed in parallel;

2. multiple subtrees of the branch-and-bound algorithm may be analyzed in parallel.

With these two properties in mind, a parallel version of the algorithm is also considered. Rather than employing two threads, an arbitrary number $\mathcal{T} \geq 2$ of CPUs is utilized to execute the branch-and-bound algorithm. The additional CPUs are initially divided among computing lower bounds and exploring the branch-and-bound tree. Here many division strategies may be employed, but to prioritize simplicity we opted for dividing the resources equally between

the two processes. Whenever there are fewer independent subproblems to be solved than the number of available CPUs, CPUs are re-allocated to explore the branch-and-bound tree. Ultimately, after all lower bounds have been computed, every CPU is assigned to search the tree.

Employing multiple CPUs to compute the lower bounds is straightforward. However, when it comes to searching the branch-and-bound tree in parallel, many design decision must be made. Gendron and Crainic (1994) and Crainic et al. (2006) present surveys on parallel branch-and-bound algorithms, indicating different strategies for parallelizing the search. We opted for a simple strategy. A certain number of nodes are initially stored in a centralized pool. The subtrees of each of these nodes are subsequently expanded by different CPUs. Each CPU retrieves the next node in the pool and performs the depth first search explained in Section 2.4.1. Whenever a subtree exploration finishes, the idle CPU takes the next node available. If a CPU is idle and the pool is empty, a flag is activated and new nodes are added to the pool by the CPUs currently working. Note that information concerning bounds are shared among the CPUs. One of the primary advantages of this strategy is the reduced (controlled) memory usage.

Experiments comparing the 2-threaded against the fully parallelized versions of the algorithm are presented in the following section.

2.4.5 Computational experiments

The branch-and-bound algorithm was coded in Java 8 and the experiments were executed on an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz computer with 16GB of RAM memory running Ubuntu Linux 12.04 LTS. In the spirit of reproducible science, the source code and all the solution files are publicly available at github[5] and at our automated TUP benchmark website[6], respectively.

This section is organized as follows. First the results obtained by the presented approach are compared against the best known results from the literature, namely those of Trick and Yildiz (2007, 2011, 2012, 2013), Trick et al. (2012), de Oliveira et al. (2014), Wauters et al. (2014), and Xue et al. (2015). Next, the impact of the presented branch-and-bound components is discussed. Finally, the benefits of parallelizing the proposed algorithm are analyzed.

[5] https://github.com/tuliotoffolo/tup
[6] https://benchmark.gent.cs.kuleuven.be/tup

Results on benchmark instances

Table 2.3 details the results obtained by the algorithm with two threads for the benchmark instances from 12 to 32 teams. The table presents, for each instance:

- the best known results: the runtime (in hours), when available, for obtaining the best known lower bound and the best solution, as well as the values of the best lower (LB) and upper bounds (UB), collected from different papers and including results reported in Section 2.3.4;
- the results obtained by the presented branch-and-bound: the runtime (in different units for compactness: seconds (s), minutes (m) or hours (h)), number of explored nodes and maximum size of subproblems solved by Algorithm 2.2 ($|S|$), as well as the lower (LB) and upper bounds (UB).

The best bounds are highlighted in the table, with ⊛ indicating that the solution was proven to be either optimum or infeasible.

Table 2.3: Branch-and-bound with decomposition-based lower bounds results for the TUP

Instance	Best known results				Branch-and-bound				
	Time and LB		Time and UB		Time	Nodes	$\|S\|$	LB	UB
12 - 7,2	-	-	-	-	14.4s	$2.8E+06$	17	⊛	86889
12 - 6,3	-	⊛	-	infeas.	1.2s	$9.0E+05$	15	⊛	infeas.
12 - 5,3	-	-	-	-	1.2s	$6.3E+05$	22	⊛	93679
12 - 4,3	-	-	-	-	5.4s	$5.5E+06$	13	⊛	89826
14 - 8,3	-	-	-	-	34.8m	$4.9E+09$	24	⊛	172177
14 - 8,2	-	-	-	-	2.9m	$3.5E+08$	18	⊛	147824
14 - 7,3	3.0h	159797	3.0h	164440	3.8m	$5.1E+08$	26	⊛	164440
14 - 7,2	-	-	-	-	0.5m	$4.9E+07$	25	⊛	146656
14 - 6,3	48.0h	157084	3.0h	159505	0.9m	$8.9E+07$	26	⊛	158875
14 - 6,2	-	-	-	-	0.3m	$3.1E+07$	26	⊛	145124
14 - 5,3	34.8h	⊛	34.8h	154962	2.2m	$2.0E+08$	26	⊛	154962
14 - 5,2	-	-	-	-	0.2m	$1.5E+07$	25	⊛	143357
14A - 8,3	-	-	-	-	20.3m	$2.8E+09$	26	⊛	166184
14A - 8,2	-	-	-	-	2.5m	$2.8E+08$	25	⊛	143043
14A - 7,3	3.0h	153199	3.0h	158760	2.1m	$2.6E+08$	26	⊛	158760
14A - 7,2	-	-	-	-	0.5m	$5.4E+07$	25	⊛	140562
14A - 6,3	48.0h	151044	3.0h	153216	0.5m	$5.5E+07$	26	⊛	152981
14A - 6,2	-	-	-	-	0.1m	$8.1E+06$	26	⊛	138927
14A - 5,3	11.4h	⊛	11.4h	149331	1.1m	$1.2E+08$	26	⊛	149331
14A - 5,2	-	-	-	-	0.6m	$6.0E+07$	24	⊛	137853

(continued on next page)

Table 2.3 continued: Branch-and-bound with decomposition-based lower bounds results for the TUP

Instance	Best known results				Branch-and-bound				
	Time and LB		Time and UB		Time	Nodes	$\|S\|$	LB	UB
14B - 8,3	-	-	-	-	22.1m	$3.0E+09$	22	⊛	165026
14B - 8,2	-	-	-	-	12.8m	$1.5E+09$	26	⊛	141312
14B - 7,3	48.0h	152518	3.0h	157884	4.0m	$5.2E+08$	24	⊛	157884
14B - 7,2	-	-	-	-	1.0m	$1.2E+08$	26	⊛	138998
14B - 6,3	48.0h	150942	3.0h	152740	1.7m	$2.2E+08$	26	⊛	152740
14B - 6,2	-	-	-	-	0.9m	$1.0E+08$	26	⊛	138241
14B - 5,3	-	⊛	-	149455	1.1m	$1.2E+08$	26	⊛	149455
14B - 5,2	-	-	-	-	0.2m	$1.9E+07$	23	⊛	136069
14C - 8,3	-	-	-	-	14.5m	$2.0E+09$	19	⊛	161262
14C - 8,2	-	-	-	-	16.4m	$2.0E+09$	21	⊛	141015
14C - 7,3	3.0h	151581	3.0h	154913	0.8m	$9.5E+07$	22	⊛	154913
14C - 7,2	-	-	-	-	5.5m	$6.5E+08$	26	⊛	138832
14C - 6,3	48.0h	148987	3.0h	150858	1.7m	$2.1E+08$	26	⊛	150858
14C - 6,2	-	-	-	-	0.7m	$7.6E+07$	26	⊛	136394
14C - 5,3	48.0h	147903	3.0h	149482	12.7m	$1.7E+09$	26	⊛	148349
14C - 5,2	-	-	-	-	0.6m	$5.7E+07$	26	⊛	134916
16 - 8,4	3.0h	193458	-	-	3.9h	3.6E+10	10	⊛	infeas.
16 - 8,3	-	-	-	-	48.0h	4.0E+11	11	162902	189415
16 - 8,2	48.0h	156089	3.0h	160705	48.0h	3.9E+11	11	145531	184977
16 - 7,4	-	-	-	-	4.6h	3.8E+10	15	⊛	197028
16 - 7,3	48.0h	160162	3.0h	168860	6.7h	5.1E+10	27	⊛	165765
16 - 7,2	48.0h	149488	3.0h	153978	18.4h	1.3E+11	30	⊛	150433
16A - 8,4	48.0h	206142	-	-	3.8h	3.6E+10	10	⊛	infeas.
16A - 8,3	-	-	-	-	48.0h	4.0E+11	11	175590	214512
16A - 8,2	48.0h	168275	3.0h	171882	48.0h	4.5E+11	9	160739	-
16A - 7,4	-	-	-	-	4.5h	3.9E+10	14	⊛	213416
16A - 7,3	3.0h	172964	3.0h	179960	4.2h	3.1E+10	26	⊛	178511
16A - 7,2	48.0h	162622	3.0h	164620	16.0h	1.2E+11	30	⊛	163709
16B - 8,4	48.0h	215521	-	-	3.8h	3.6E+10	9	⊛	infeas.
16B - 8,3	-	-	-	-	48.0h	4.1E+11	11	178821	217764
16B - 8,2	48.0h	170385	3.0h	180728	48.0h	3.6E+11	10	165737	202897
16B - 7,4	-	-	-	-	5.0h	4.3E+10	13	⊛	223868
16B - 7,3	3.0h	173023	3.0h	181565	37.8h	2.9E+11	30	⊛	180204
16B - 7,2	48.0h	164816	3.0h	170194	38.4h	2.6E+11	26	⊛	167190
16C - 8,4	48.0h	206369	-	-	3.9h	3.6E+10	10	⊛	infeas.
16C - 8,3	-	-	-	-	48.0h	4.0E+11	11	175435	214993
16C - 8,2	48.0h	169698	3.0h	179939	48.0h	3.7E+11	10	164541	204887
16C - 7,4	-	-	-	-	5.6h	4.7E+10	12	⊛	209088
16C - 7,3	48.0h	172755	3.0h	184181	48.0h	3.4E+11	18	176161	180483
16C - 7,2	48.0h	164626	3.0h	169184	37.6h	2.6E+11	27	⊛	166479

(continued on next page)

Table 2.3 continued: Branch-and-bound with decomposition-based lower bounds results for the TUP

Instance	Best known results				Branch-and-bound				
	Time and LB		Time and UB		Time	Nodes	$\|S\|$	LB	UB
18 - 9,4	48.0h	213806	-	-	48.0h	3.7E+11	9	193632	-
18 - 9,3	-	-	-	-	48.0h	3.8E+11	9	186173	262987
18 - 8,4	-	-	-	-	48.0h	3.3E+11	10	197511	254155
18 - 8,3	-	-	-	-	48.0h	3.3E+11	11	187335	248302
18 - 7,4	-	-	-	-	48.0h	3.1E+11	15	200551	217502
20 - 10,5	3.0h	216333	-	-	48.0h	3.3E+11	8	220907	-
22 - 11,5	3.0h	245518	-	-	48.0h	3.1E+11	6	243052	-
24 - 12,6	3.0h	273057	-	-	48.0h	3.5E+10	4	250590	-
26 - 13,6	3.0h	312786	-	-	48.0h	2.8E+10	4	289651	-
28 - 14,7	3.0h	350263	-	-	48.0h	9.2E+09	3	322208	-
30 - 15,7	3.0h	413103	-	-	48.0h	4.1E+09	3	339331	-
32 - 16,8	3.0h	430890	-	-	48.0h	5.2E+09	3	369695	-

Note that we also report results for non-standard instances in Table 2.3, with $q_1 > n$ and $q_2 = 2$. Table 2.3 confirms how the branch-and-bound results clearly outperform the best known results from the literature for the 14-team instances. Before this work, only three 14-team instances had their optimality proven. Xue et al. (2015) required approximately 46h to prove optimality for two of these instances (the runtime to obtain the optimum solution for instance '14B-5,3', collected from Trick and Yildiz's website[7], is unknown). The proposed branch-and-bound with decomposition-based lower bounds is capable of producing (and proving) optimal solutions for all these three instances in approximately 4 minutes. Optimality was also quickly proven for all the other 14-team instances. The procedure required, on average, 5 minutes to solve each instance. It is noteworthy, however, to highlight how instances with higher values for q_1 and q_2 demand more computational effort from the branch-and-bound.

The time limit for 16-team and larger instances was set to 48 hours, thereby enabling a fair comparison with the approaches proposed by Xue et al. (2015).

Table 2.3 reveals how the branch-and-bound obtained 11 optimal solutions for the 16-team instances, improving 8 upper bound values reported in the literature. Nevertheless, some of the results obtained are poor when compared against the best results from the literature. For example, no solution was obtained for instance '16A-8,2'. This demonstrates how obtaining feasible solutions for

[7] http://mat.gsia.cmu.edu/TUP/

highly-constrained instances may take considerable additional time. Without an upper bound, the proposed algorithm behaves like a naive enumeration procedure. For the more constrained instances, even solving the subproblems is difficult. This is confirmed by the smaller size $|S|$ of the largest subproblem solved for these instances. Therefore, despite the impressive results for the 14-team instances, the algorithm's exponential time complexity is noticeable when solving instances with more than 14 teams. This behavior is evident in the results for the 18-team instances, where the average gap is approximately 21%.

Impact of the branch-and-bound components

We present experiments to analyze the impact of the main branch-and-bound components. The objective is to define which components are the most important, with the goal of providing a simpler algorithm or at least of indicating the least relevant components. Four algorithm versions were prepared:

- the complete algorithm, with all the described components;
- the algorithm without the local search procedure presented in Section 2.4.1;
- the algorithm without the partial matching presented in Section 2.4.3;
- and the algorithm without the bound propagation presented in Section 2.4.2;

The different versions of the algorithm were executed for the standard 14-team instances which were considered by other studies. The total runtime and the total number of nodes generated before finding (and proving) an optimum solution were analyzed. Figure 2.8 presents a graph showing the results of these executions. Since the total number of nodes is proportional to the total runtime, only the runtime is shown in the figure. The vertical axis presents the percentage of processing time to solve the instance while the horizontal axis lists the different instances considered. This figure shows that removing any one of the components negatively impacts the total runtime. Among the considered components, the partial matching had the highest overall impact, followed by the local search procedure. The bound propagation had the smallest impact given that the subproblems were solved very quickly.

We also ran experiments disabling other features of the algorithm, such as the lower bound strengthening by decomposition. However, when such features were deactivated the total runtime exceeded the imposed limit of 24 hours.

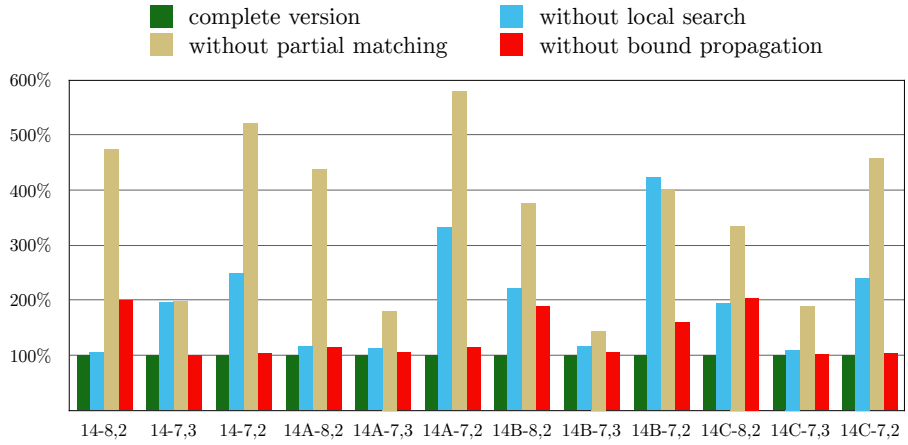


Figure 2.8: Performance of the branch-and-bound with deactivated components on 14-team instances

Parallelization

Table 2.4 presents the runtime (in seconds) and the speed-up obtained with the parallel version discussed in Section 2.4.4 for 14-team instances. An Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz was employed for this particular experiment. The algorithm was executed utilizing both 2 and 16 CPUs. Note how for some instances the speedup is superior to the proportional number of CPUs added (values highlighted in the table). As discussed by Crainic et al. (2006), these speedup anomalies are due to the reduction in the number of nodes explored by the branch-and-bound. By searching the tree in parallel, improved upper bounds may be obtained earlier, enabling additional pruning. More pruning means less nodes to explore, thereby explaining some of the spectacular speedups observed in Table 2.4. In one particular case, the additional overhead negatively impacted the algorithm.

Finally, note how instances '14-5,3' and '14A-5,3', previously solved by Xue et al. (2015) in 46 hours, were solved in only 35 seconds by the proposed parallel branch-and-bound.

Table 2.4: Parallel branch-and-bound gain when utilizing eight times more processors

Instance	2 CPUs	16 CPUs	Speedup
14 - 8,3	1871.0	234.1	7.99x
14 - 8,2	96.8	22.4	4.32x
14 - 7,3	215.2	17.1	12.62x
14 - 7,2	25.5	5.0	5.09x
14 - 6,3	49.4	26.4	1.87x
14 - 6,2	17.5	3.7	4.72x
14 - 5,3	126.1	20.4	6.18x
14 - 5,2	9.7	3.5	2.74x
14A - 8,3	1152.6	114.3	10.08x
14A - 8,2	129.0	28.8	4.47x
14A - 7,3	116.4	20.0	5.82x
14A - 7,2	28.8	7.2	4.00x
14A - 6,3	27.8	3.8	7.24x
14A - 6,2	5.0	4.2	1.18x
14A - 5,3	62.6	15.0	4.19x
14A - 5,2	32.3	4.7	6.85x

Instance	2 cores	16 cores	Speedup
14B - 8,3	1219.9	224.2	5.44x
14B - 8,2	745.0	226.2	3.29x
14B - 7,3	238.8	47.5	5.02x
14B - 7,2	70.2	14.0	5.02x
14B - 6,3	100.3	8.1	12.37x
14B - 6,2	66.9	17.5	3.82x
14B - 5,3	60.5	5.2	11.74x
14B - 5,2	11.7	4.9	2.39x
14C - 8,3	908.2	83.4	10.90x
14C - 8,2	827.7	298.8	2.77x
14C - 7,3	45.6	12.0	3.80x
14C - 7,2	323.0	41.6	7.76x
14C - 6,3	97.2	9.8	9.94x
14C - 6,2	40.2	30.9	1.30x
14C - 5,3	751.7	132.7	5.67x
14C - 5,2	31.1	32.6	0.95x

2.5 Decomposition-based heuristic

The branch-and-bound with decomposition-based lower bounds, despite considerably improving upon all exact methods proposed for the TUP, is incapable of handling large instances. This is an expected result, given the algorithm's exponential time complexity. Seeking to provide high-quality solutions for the largest instances, a decomposition-based heuristic is proposed.

The employed decomposition strategy is very similar to the one proposed in Section 2.4.2. However, rather than being applied to compute lower bounds, it is instead utilized to produce feasible solutions. The time horizon, here represented by rounds, defines subproblems solvable within short computational time. A solution is iteratively constructed by combining these subproblem solutions. Following this, a local search also employing the decomposition is performed to improve solution quality. The constructive procedure is detailed next, in Section 2.5.1, and the local search is explained in Section 2.5.2. Computational experiments evaluating the heuristic are presented in Section 2.5.3.

2.5.1 Constructive procedure

The constructive approach begins by decomposing the problem into subproblems containing η consecutive rounds each. Evidently, η is an important parameter: it must be large enough to produce non-trivial subproblems but also small enough so that subproblems are quickly solved. Figure 2.9 presents an example of such a decomposition, with $\eta = 4$ and one round of intersection among subproblems. The amount of intersection between two subproblems is given by the parameter $step \in \mathbb{Z}^+$, with $step \leq \eta$. Note in the example of Figure 2.9 that $step = \eta - 1$, meaning $step = 3$ in this case.

	Round	Game [Home x Away]			
step	1	AxE	FxB	GxC	DxH
	2	AxF	BxH	ExC	DxG
	3	AxD	BxG	CxH	ExF
	4	FxA	HxB	CxG	ExD
	5	HxA	GxB	CxD	FxE
	6	GxA	BxC	DxF	HxE
	7	AxH	BxF	DxC	GxE
	8	AxG	BxD	CxF	ExH
	9	DxA	CxB	ExG	FxH
	10	BxA	FxC	DxE	HxG
	11	CxA	BxE	HxD	GxF
	12	AxB	CxE	GxD	HxF
	13	AxC	ExB	FxD	HxG
	14	ExA	DxB	HxC	FxG

η

Figure 2.9: Decomposition example with $\eta = 4$ and $step = 3$

Once the problem is decomposed, the solution process begins. Initially, the first subproblem – the one containing the first round – is solved to optimality. Next, the second subproblem is solved taking into account the solution of the first subproblem. Allocations made in previous rounds are fixed when solving each subsequent subproblem. The procedure repeats until all subproblems are solved. If a feasible solution is obtained for all subproblems, then a feasible initial solution is produced for the entire problem. However, fixations enforced by subproblems previously solved often results in an infeasible subproblem. If such a situation occurs, the algorithm backtracks and an alternative solution is produced for the previous subproblem. To reduce

infeasibility and consequently lessen backtracking, a heuristic objective function is employed for the subproblems.

Note that the proposed constructive procedure resembles the relax-and-fix algorithm presented by de Oliveira et al. (2014). However, some significant differences should be highlighted. Rather than dividing the problem into subproblems, de Oliveira et al. (2014) relax the integrality of variables related to some rounds, while keeping variables concerning a subset of rounds integer. Once the partially relaxed problem is solved, the integer variables have their values fixed and other previously-relaxed variables become integer. This procedure repeats until all variables have been fixed. Such a strategy disallows addressing instances with more than 30 rounds because the runtime to solve the linear relaxation is unreasonably large for these instances.

Algorithm 2.3 presents the constructive procedure as a recursive algorithm. The stopping criterion is a completely formed solution (lines 1-2), which is immediately returned. Otherwise, a subproblem P is defined (line 3) and solved (line 4). All solutions obtained are stored in a list L, together with the optimal solution. Beginning with the best solution, the algorithm iterates over all solutions $s' \in L$ (line 5). The assignments in s' are then included in S (line 6). Next, a recursive call is made to solve the remaining subproblems (line 7). If a feasible solution is obtained, the algorithm returns it (line 8). Otherwise, it removes allocations of s' from S (line 9) and continues with the next solution. If no subproblem solution in L culminates in feasible assignments for S, the algorithm returns an empty, infeasible solution (line 10).

Algorithm 2.3: Decomposition-based constructive algorithm for the TUP

Input: Solution S (initially $\emptyset$), current round r (initially one), subproblem size η and intersection parameter *step*

Constructive(S, r, η, *step*)

```
1   if S is a feasible solution then
2       return S // success: feasible solution is returned
3   P ← subproblem {r, ..., min(r + η − 1, |R|)} considering allocations in S
4   L ← list of feasible solutions (allocations) for P, sorted by increasing cost
5   for s' ∈ L do
6       S ← S ∪ s' // fix allocations of subproblem solution s'
7       if Constructive(S, r + step, η, step) ≠ ∅ then
8           return S
9       S ← S\s' // unfix (remove) allocations of subproblem solution s'
10  return ∅ // backtracks since all subproblem solutions resulted in infeasibility
```

Reducing infeasibility

Constraint (c) – every umpire should visit the home of every team at least once – is the main cause of infeasibility when solving the last subproblems. Taking this into account, a heuristic objective function is proposed to reduce infeasibility.

Rather than exclusively minimizing travel distance, subproblems include the number of different locations visited by the umpires in their objective function. The principle is straightforward: by maximizing the number of locations visited by each umpire in the early rounds, Constraint (c) becomes less relevant for the last rounds, thereby reducing the risk of infeasibility. The objective function for the subproblems is given by Equation (2.16), with $\ell_{i,u}$ indicating whether location i is visited by umpire u ($\ell_{i,u} = 0$) or not ($\ell_{i,u} = 1$), and ψ_u the multiplier applied to each location not visited by umpire u.

$$\sum_{e \in E} \sum_{u \in U} d_e x_{e,u} + \sum_{i \in I} \sum_{u \in U} \psi_u \ell_{i,u} \tag{2.16}$$

Note that the vector ψ represents a critical parameter given how it strongly influences solution quality. The focus of the constructive method is, however, on feasibility and thus ψ is generally composed of large values. Equation 2.17 presents the calculation of ψ_u. In this equation, ω corresponds to a regular weight (multiplier), r to the first round of the subproblem and ℓ_u^0 to the initial number of unattended locations of umpire u. The principle behind the square root of r is to give higher (but not too high) penalties in later rounds. As for summing ℓ_u^0 in ψ_u, the goal is to assign (slightly) higher priority to umpires with more unattended locations.

$$\psi_u = \omega\sqrt{r+1} + \ell_u^0 \tag{2.17}$$

Note from Equation 2.17 that ψ_u is a heuristic multiplier applying problem-specific information. The impact of such a heuristic objective function is evaluated throughout Section 2.5.3.

2.5.2 Local search

Once a feasible solution is obtained by the constructive approach, the local search phase begins. Here the decomposition proposed in the previous section serves as a neighborhood within a *Hill Climbing* (HC) procedure. The principle is to solve

subproblems while fixing allocations of games not included in the subproblem. A subproblem is, again, defined by a set of η consecutive rounds. Once no improvement is achievable from a certain decomposition (set of subproblems), the value of η is incremented. The procedure continues until the time limit is reached or $\eta = |R|$, in which case a subproblem represents the entire TUP instance.

Algorithm 2.4 describes the local search decomposition method. The algorithm iterates until either the entire TUP instance is solved or the time limit is reached (line 1). For each iteration, the round r and the number of subproblems without improvement c are initialized (lines 2-3). Then, until a local optimum is reached (line 4), a subproblem P is defined (line 5) and solved (line 6). If the solution improves upon the previous one, meaning that the cost of solution S is reduced, it is accepted and the counter is reset (lines 7-9). Otherwise, the counter is incremented (lines 10-11). Finally, r is updated to consider the next subproblem (line 12). When a local optimum is reached, η is incremented (line 13). Once one of the stopping criteria is reached, the best solution obtained is returned (line 14).

Algorithm 2.4: Decomposition-based local search for the TUP

Input: Solution S, subproblem size η and intersection parameter *step*
LocalSearch(S**,** η**,** *step*)

```
1  while η ≤ |R| and time limit not reached do
2      r ← 1 // first subproblem begins with the first round
3      c ← 0 // counter of non-improving iterations
4      while c < ⌊|R|/step⌋ do
5          P ← subproblem {r, ..., min(r + η − 1, |R|)} considering S
6          S_P ← optimum solution of P
7          if S_P improves solution S then
8              update S with allocations from S_P
9              c ← 0
10         else
11             c ← c + 1
12         r ← (r + step) mod |R|
13     η ← η + 1
14 return S
```

2.5.3 Computational experiments

The decomposition-based heuristic was coded in Java 8 and the experiments were executed on an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz computer with 16GB of RAM memory running Ubuntu Linux 12.04 LTS. Subproblems were solved by Gurobi 7.5 employing Formulation (2.1)-(2.7) with the altered objective function given by Equation (2.16). The experiments were executed with $\eta = 5$ and $step = 2$, since larger values for η resulted in long runtimes. The penalty ω was set initially to 50 during the constructive procedure. Whenever infeasible solutions are produced, the algorithm automatically increases ω's value by 25 units. The procedure continues until a feasible solution is obtained. As with the developed branch-and-bound, the source code and all solution files have been made publicly available online at our automated TUP benchmark website[8].

Table 2.5 compares the results obtained with the proposed heuristic method against the best known ones, including those produced by the branch-and-bound with decomposition-based lower bounds. Both medium- and large-size instances were considered for the experiments. The table reports the best known solution (column BKS) followed by (i) results obtained by the constructive approach and (ii) the final solution obtained after local search. Runtimes are presented in seconds and column gap reports the gap between the obtained solution and the previous best known, with gap calculated as $\frac{UB-BKS}{UB}$. The constructive method was executed until a feasible solution was returned – possibly updating multiple times the value for ω –, and the local search ran for exactly one hour. The total runtime was always below three hours.

Table 2.5 demonstrates the efficiency of the proposed decomposition-based heuristic. Many instances had their best known solution improved. For the largest instances, with at least 20 teams, solutions produced by the constructive procedure alone improved upon previous best solutions. For smaller 14- and 16-team instances, however, results were less competitive, with average gap of 3.17% after local search.

The impact of employing a heuristic objective function was also evaluated. Table 2.6 shows the initial solution quality for two scenarios, one employing the TUP's original objective function (Orig. obj.) and another employing the proposed heuristic objective function (Heur. obj.). The table also displays the value for ω that resulted in a feasible solution. Solution distance is omitted

[8]https://benchmark.gent.cs.kuleuven.be/tup

Table 2.5: Decomposition-based heuristic results for the TUP

Instance	BKS	Constructive			Local search		
		Time	UB	gap$^+$	Time	UB	gap$^+$
umps14 - 7,3	164440	11.9	176300	6.7%	3600	172510	4.7%
umps14 - 6,3	158875	32.1	168691	5.8%	3600	163933	3.1%
umps14 - 5,3	154962	26.3	157419	1.6%	3600	157402	1.6%
umps14A - 7,3	158760	17.7	171366	7.4%	3600	166329	4.6%
umps14A - 6,3	152981	34.3	160563	4.7%	3600	157917	3.1%
umps14A - 5,3	149331	23.3	156821	4.8%	3600	151590	1.5%
umps14B - 7,3	157884	4.1	169668	6.9%	3600	165421	4.6%
umps14B - 6,3	152740	27.1	157216	2.8%	3600	156659	2.5%
umps14B - 5,3	149455	28.4	153953	2.9%	3600	152357	1.9%
umps14C - 7,3	154913	8.2	164545	5.9%	3600	164360	5.7%
umps14C - 6,3	150858	38.4	158212	4.6%	3600	153542	1.7%
umps14C - 5,3	148349	27.6	153577	3.4%	3600	151312	2.0%
umps16 - 8,2	160705	35	177915	9.7%	3600	164665	2.4%
umps16 - 7,3	165765	11.9	173304	4.4%	3600	171425	3.3%
umps16 - 7,2	150433	12.1	161644	6.9%	3600	155656	3.4%
umps16A - 8,2	171882	105.5	191631	10.3%	3600	171882	0.0%
umps16A - 7,3	178511	23.8	183946	3.0%	3600	181879	1.9%
umps16A - 7,2	163709	16.6	170700	4.1%	3600	168447	2.8%
umps16B - 8,2	180728	82.1	181232	0.3%	3600	180540	-0.1%
umps16B - 7,3	180204	20.5	193558	6.9%	3600	191974	6.1%
umps16B - 7,2	167190	28.7	176386	5.2%	3600	172579	3.1%
umps16C - 8,2	179939	15.7	203485	11.6%	3600	186253	3.4%
umps16C - 7,3	172755	26.9	195462	11.6%	3600	188818	8.5%
umps16C - 7,2	164626	30.2	173871	5.3%	3600	172319	4.5%
umps18 - 5,5	213806	73.9	223190	4.2%	3600	221480	3.5%
umps20 - 5,5	-	366.8	251782	-	3600	250569	-
umps22 - 5,5	-	758.7	279342	-	3600	278842	-
umps24 - 5,5	-	1897.8	310468	-	3600	305514	-
umps26 - 5,5	354134	2498.5	352016	-0.6%	3600	351932	-0.6%
umps28 - 5,5	398101	2095.7	390832	-1.9%	3600	390635	-1.9%
umps30 - 5,5	450919	5983.4	444870	-1.4%	3600	443739	-1.6%
umps32 - 5,5	502890	4279.8	493007	-2.0%	3600	491075	-2.4%

whenever the resulting solution is infeasible in respect to Constraint (c). Note how the decomposition-based constructive algorithm consistently fails to provide a feasible solution when the original objective function is employed. It was capable of providing feasible solutions only for four out of 40 considered instances. It is therefore clear that the heuristic objective function is essential to lead the constructive algorithm towards feasible solutions.

Table 2.6: Impact of heuristic objective function within the decomposition-based constructive heuristic for the TUP

Instance	Orig. obj.	Heur. obj.		Instance	Orig. obj.	Heur. obj.	
		ω	Obj.			ω	Obj.
umps14 - 7,3	-	150	176300	umps14B - 7,3	-	75	169668
umps14 - 6,3	-	225	168691	umps14B - 6,3	-	200	157216
umps14 - 5,3	-	150	157419	umps14B - 5,3	-	175	153953
umps14A - 7,3	-	150	171366	umps14C - 7,3	-	100	164545
umps14A - 6,3	-	225	160563	umps14C - 6,3	-	275	158212
umps14A - 5,3	-	125	156821	umps14C - 5,3	-	125	153577
umps16 - 8,2	179837	50	177915	umps16B - 8,2	-	125	181232
umps16 - 7,3	-	50	173304	umps16B - 7,3	-	75	193558
umps16 - 7,2	-	50	161644	umps16B - 7,2	-	75	176386
umps16A - 8,2	-	50	191631	umps16C - 8,2	-	50	203485
umps16A - 7,3	185202	50	183946	umps16C - 7,3	196821	50	195462
umps16A - 7,2	170353	50	170700	umps16C - 7,2	-	100	173871
umps18 - 5,5	-	100	223190	umps26 - 5,5	-	75	352016
umps20 - 5,5	-	100	251782	umps28 - 5,5	-	50	390832
umps22 - 5,5	-	75	279342	umps30 - 5,5	-	75	444870
umps24 - 5,5	-	100	310468	umps32 - 5,5	-	50	493007

2.6 Conclusions and future work

This chapter introduced three decomposition approaches for the TUP, devoting special attention to both computation of tight lower bounds and production of high-quality feasible solutions.

The algorithms improved a large number of lower and upper bounds. Among these improving results, optimality was proven for all the 14-team instances and for 11 of the 16-team instances. While previous approaches required over 24 hours of runtime to solve the less constrained 14-team instances, the research presented in this chapter enabled solving such instances within a few minutes of runtime. Parallelization of the algorithm resulted in even lower runtimes. When optimality was not proven, the branch-and-bound with decomposition-based lower bounds was capable of generating competitive feasible solutions for instances with 16 teams, improving the best known result in one case. It was also proven that no feasible solutions exist for instances '16-8,4'.

A decomposition-based heuristic was proposed to address larger instances. Several new best known solutions were produced within limited runtimes.

In summary, the time-structure decomposition proposed and evaluated within branch-and-bound and heuristic algorithms obtained very strong results, outperforming all other approaches proposed for the problem. These results motivate questions concerning the general applicability of such a straightforward decomposition principle. Can it outperform classical heuristic and exact algorithms for other problems? This and other questions are addressed throughout the following chapters.

Concerning the TUP more particularly, there seems to be room for improvement within the proposed algorithms. The branch-and-bound currently employs a DFS to explore the branching tree, greedily selecting the next node. Future research directions include evaluating other strategies in addition to alternative criteria for selecting the node to process. As for the decomposition-based heuristic, future work include investigating different objective functions for the subproblems generated during the constructive algorithm.

Chapter 3

Nurse Rostering Problem

This chapter addresses the *Nurse Rostering Problem* (NRP), a challenging timetabling problem in which nurses must be assigned to shifts while respecting hard constraints and minimizing the violation of soft constraints. The NRP is also very relevant in practice, having been the subject of two prestigious international algorithm optimization challenges. This chapter focuses on the problem defined during the *First International Nurse Rostering Competition* (INRC-1), for which a broadly studied set of benchmark instances is available.

This chapter builds upon the content of my presentation at PATAT'2012 (Santos et al., 2012)[1] and my contributions to the extended journal version (Santos et al., 2016)[2], which are partially reproduced here together with the novel content, algorithm and results. Initially, Section 3.1 discusses the NRP while introducing the problem defined during the INRC-1. Next, Section 3.2 presents an IP formulation for the problem with a polynomial number of constraints and variables. The formulation is evaluated within two state-of-the-art solvers, with many instances solved quickly. For other instances, however, large optimality gaps motivated the investigation of different approaches. Dantzig-Wolfe decomposition is applied, resulting in a formulation with an exponential number of variables whose linear relaxation provides strong bounds. Section 3.3

[1]Santos, H. G., **Toffolo, T. A. M.**, Ribas, S., and Gomes, R. A. M., (2012). Integer programming techniques for the nurse rostering problem. Proceedings of the 9th International Conference on Practice and Theory of Automated Timetabling (PATAT) 2012, pages 257-282.

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

describes the decomposition and the column generation algorithm employed to solve the linear relaxation of the reformulated model. While capable of providing strong bounds, the column generation is unable to provide integer solutions for most instances. Branching would be the expected approach, however runtime was already long for solving the column generation. Therefore, with a view to obtain competitive solutions within short computational times, a heuristic employing three different decompositions is proposed in Section 3.4. Many principles explored throughout Chapter 2 are applied here within an algorithm consisting of constructive and local search phases. The different decompositions are analyzed and computational experiments validate the resulting methodology. Every single best known solution was generated by the proposed decomposition-based heuristic, which was even capable of obtaining improved solutions for the long-studied dataset addressed. By combining the heuristic's results and those obtained by column generation, most solutions were proven optimal. This chapter finishes with Section 3.5, where conclusions are discussed.

3.1 Introduction

A significant amount of research has been devoted towards computationally solving the NRP. Much of the literature, however, concentrates on specific case studies and consequently focuses on the particularities of certain institutions. In such situations, comparing different solution strategies proves a very difficult task. The INRC-1 (Haspeslagh et al., 2012) was organized to stimulate research in the NRP and offered an opportunity for evaluating different solution strategies for the problem by proposing a set of benchmark instances.

The NRP may be generally described as the problem of assigning working shifts and days-off to nurses throughout a given time horizon, typically one month. A solution may be represented by a matrix M where each cell $M_{n,d}$ contains the set of shifts to be performed by nurse n on day d. While this set may have any number of shifts, in most practical cases and within the INRC-1 problem a nurse performs at most one shift per day. Generally morning (M), evening (E), night (N), and late (L) are possible shift allocations. Days off (-) are indicated by the absence of working shifts in a day. Table 3.1 presents a one-week roster example which indicates the shift allocated to each nurse on each day.

The scope of this chapter is limited to the problem defined on the occasion of the INRC-1. Therefore, a brief description of the approaches that won the

Table 3.1: Example of a one-week NRP solution

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Nurse 1	E	N	N	N	-	-	M
Nurse 2	M	M	E	E	-	-	E
Nurse 3	-	E	M	-	M	M	M

competition is provided.

Valouxis et al. (2012) won the INRC-1 with a two phase algorithm. In the first phase the workload for each nurse and for each day of the week was decided, while the second phase assigned specific daily shifts. Since the INRC-1 imposed quality and runtime constraint requirements, Valouxis et al. (2012) partitioned the problem instances into subproblems of manageable computational size which were subsequently solved using IP. They also applied local optimization techniques for searching across combinations of partial nurse schedules. This sequence was repeated several times depending on the computational time available.

Burke and Curtois (2014) applied an ejection chain based method for small (*sprint*) instances and a branch-and-price algorithm for *medium* and *long* instances defined during the competition. Problem instances were converted into the general staff rostering model proposed and documented by the same team. Their software *Roster Booster*, which includes the aforementioned algorithmic approaches, was then employed.

Bilgin et al. (2012) applied a hyper-heuristic approach combined with a greedy shuffle heuristic. The hyper-heuristic consisted of a heuristic selection method and a move acceptance criterion. The best solution found was further improved by exploring swaps of partial rosters between nurses.

Further details concerning these approaches are available on the competition's website[3]. Various authors proposed approaches for the INRC-1 problem after the challenge, producing new best known solutions in some cases. A wide range of methods were evaluated. Readers interested in these methods are referred to Nonobe (2010), Lü and Hao (2012), Tassopoulos et al. (2015) and Awadallah et al. (2017).

[3]https://www.kuleuven-kulak.be/nrpcompetition/competitor-ranking

The INRC-1 problem

Combinatorial optimization problems are generally associated with hard and soft constraints. Broadly speaking, the difference is that hard constraints must be met while soft constraint, though permissible, should be avoided. The INRC-1 problem considers two hard constraints (I-II):

I) a nurse cannot work more than one shift per day;

II) all shift type demands during the planning period must be met.

Additionally, 13 soft constraints are considered, subdivided into two groups: *ranged* and *logical*. *Ranged* soft constraints are those which state a range (minimum and maximum) of valid values. Each unit outside the defined range incurs a penalty in the objective function. In total, six ranged soft constraints are considered:

1) minimum/maximum number of shifts assigned to a nurse;

2) minimum/maximum number of consecutive free days;

3) minimum/maximum number of consecutive working days;

4) maximum number of working weekends in four weeks;

5) minimum/maximum number of consecutive working weekends;

6) number of days off after a series of night shifts.

Logical soft constraints are those which can be either satisfied or not. Whenever one of these constraints is violated, a penalty is applied in the objective function. The INRC-1 problem considers seven logical soft constraints:

7) complete weekends: if a nurse is assigned to work only part of a weekend (and not the entire weekend) then a penalty occurs;

8) no night shift before free weekend: if a nurse does not work on the weekend, then Friday night should also be free, otherwise a penalty occurs;

9) identical shift types during the weekend: assignments of different shift types to the same nurse during a weekend are penalized;

10) alternative skill: if a shift type requires a certain skill which the assigned nurse does not have, then the solution is penalized accordingly;

11) unwanted patterns: an unwanted pattern is a sequence of assignments which do not correspond to the preferences outlined in the nurse's contract;

12) days on/off request: requests by nurses to work or not on specific days of the week should be respected, otherwise solution quality is compromised;

13) shift on/off request: similar to the previous constraint but for specific shifts on certain days instead.

A solution for the INRC-1 problem must satisfy the hard constraints while minimizing violations of soft constraints. The objective is, therefore, to minimize the total penalty incurred by both ranged and logical soft constraint violations.

A compact IP formulation further detailing the INRC-1 problem is discussed throughout the following section.

3.2 Integer programming formulation

This section presents the IP formulation produced in cooperation with Haroldo Santos[4] and first introduced by Santos et al. (2012), which is employed to derive the methodologies proposed throughout the remainder of this chapter. The formulation successfully models all constraints, explicitly or implicitly, considered within instances of the INRC-1, wherein nurses are hired under different contracts. Despite the large quantity of contractual data within this formulation, most sets and parameters are presented in a generic manner to simplify the notation.

The classification of soft constraints into ranged and logical is also utilized to simplify the notation. The constraint numbers (or indices) presented in the previous section are employed to express parameters and auxiliary variables. Since all soft constraints are nurse-oriented, these parameters and auxiliary variables are generally defined for each soft constraint i and nurse n.

Ranged soft constraints state a range of valid values for a variable in the format $\underline{\gamma} \leq \text{value} \leq \overline{\gamma}$. Therefore, each ranged soft constraint i and nurse n is associated with minimum and maximum valid values $\underline{\gamma}_n^i$ and $\overline{\gamma}_n^i$, respectively. The penalty for violating the minimum limit is denoted by $\underline{\omega}_n^i$ while $\overline{\omega}_n^i$ represents the

[4]Prof. Dr. Haroldo Gambini Santos (haroldo@iceb.ufop.br), Department of Computing, Federal University of Ouro Preto, Brazil

penalty for violating the maximum limit. Moreover, the violation of such a constraint is measured by integer slack variables $\underline{\alpha}_n^i$ and $\overline{\alpha}_n^i$.

Logical soft constraints have a boolean behavior, and therefore a binary slack variable ω_n^i is employed to indicate whether constraint i is violated for nurse n. In case of a violation, penalty ω_n^i is applied to the objective function. Note that some logical soft constraints penalize unwanted patterns, such as Soft Constraints 11 and 12. To simplify expressing these constraints, the set of unwanted allocation patterns for nurse n is defined as $\acute{P}_n$ with each pattern $\acute{p} \in \acute{P}_n$ having size $|\acute{p}|$ and content $\acute{p}[1], \ldots, \acute{p}[|\acute{p}|] \in S$. Analogously, the set of day-related patterns is defined as $\hat{P}_n$, with elements $\hat{p} \in \hat{P}_n$ having size $|\hat{p}|$.

In summary, the following notation is considered:

N : set of nurses;

S : set of shifts, with $\tilde{S} \subset S$ being the set of night shifts and $\hat{S} \subset S$ the set of day shifts;

D : set of days with elements sequentially numbered from 1;

$\tilde{r}_{s,d}$: number of required nurses on day d for shift s;

Π : set of all ordered pairs $(d_1, d_2) \in D \times D : d_1 \leq d_2$ representing windows in the planning horizon;

$\tilde{W}_n$: set of weekends in the planning horizon according to the weekend definition for nurse n, with elements numbered from 1 to $\tilde{w}_n$;

$\tilde{D}_{i,n}$: set of days in the i-th weekend of nurse n;

$l_{i,n}$: last day before the i-th weekend of nurse n;

$\acute{P}_n$: set of unwanted working shift patterns for nurse n;

$\hat{P}_n$: set of unwanted working day patterns for nurse n;

$\underline{\gamma}_n^i, \overline{\gamma}_n^i$: lower and upper limits for ranged soft constraint i for nurse n, respectively;

$\underline{\omega}_n^i, \overline{\omega}_n^i$: penalties associated with violating the lower and upper limits of ranged soft constraint i for nurse n, respectively;

ω_n^i : penalty associated with violating logical soft constraint i for nurse n;

σ_{n,d_1,d_2} : precalculated penalty for the continuous work of nurse n on days $\{d_1, \ldots, d_2\}$ considering Soft Constraints 3 and 7;

τ_{n,d_1,d_2} : precalculated penalty for the continuous rest of nurse n on days $\{d_1, \ldots, d_2\}$ considering Soft Constraint 2;

ψ_{n,i_1,i_2} : precalculated penalty for the continuous work of nurse n on weekends $\{i_1, \ldots, i_2\}$ considering Soft Constraint 5;

$\nu_{n,s,d}$: precalculated penalty for the work of nurse n for shift s on day d considering Soft Constraints 10 and 13.

The formulation considers binary decision variables $x_{n,s,d}$ in addition to four auxiliary variable sets:

$$x_{n,s,d} = \begin{cases} 1 & \text{if nurse } n \text{ is allocated to shift } s \text{ on day } d \\ 0 & \text{otherwise} \end{cases}$$

$$y_{n,i} = \begin{cases} 1 & \text{if nurse } n \text{ works on weekend } i \\ 0 & \text{otherwise} \end{cases}$$

$$w_{n,d_1,d_2} = \begin{cases} 1 & \text{if nurse } n \text{ works from day } d_1 \text{ until day } d_2 \\ 0 & \text{otherwise} \end{cases}$$

$$r_{n,d_1,d_2} = \begin{cases} 1 & \text{if nurse } n \text{ rests from day } d_1 \text{ until day } d_2 \\ 0 & \text{otherwise} \end{cases}$$

$$z_{n,i_1,i_2} = \begin{cases} 1 & \text{if nurse } n \text{ works from weekend } i_1 \text{ until weekend } i_2 \\ 0 & \text{otherwise} \end{cases}$$

Additionally, the formulation considers three sets of slack variables:

$\underline{\alpha}_n^i \in \mathbb{Z}^+$, $\overline{\alpha}_n^i \in \mathbb{Z}^+$: slack variables measuring violations of ranged soft constraint i's lower and upper limits for nurse n, respectively; note that additional indices may be employed for some specific soft constraints;

$\alpha_n^i \in \{0, 1\}$: binary variables indicating whether logical soft constraint i is violated for nurse n ($\alpha_n^i = 1$) or not ($\alpha_n^i = 0$); again, additional indices may be employed for specific soft constraints.

Some slack variables (and their respective constraints) need not be explicitly included. This is the case for constraints which are directly linked to the selection of a specific working or resting window from Π. Take for instance Soft Constraints 2 and 3. The consecutive free and working days assigned to a nurse is indicated directly by variables r_{n,d_1,d_2} and w_{n,d_1,d_2} respectively. There is, therefore, no need to explicitly define variables or constraints for them. This is also true for Soft Constraints 5, 7, 10 and 13. Soft Constraint 5 violations are directly measured by variables $z_{n,i1,i2}$ while violations of Soft Constraints 7 are indicated by the activation of w_{n,d_1,d_2} variables which begin or finish during the course of a weekend. Finally, Soft Constraints 10 and 13 can be implicitly evaluated by activating certain $x_{n,s,d}$ variables (recall that within INRC-1 instances skills and shifts are directly related).

Table 3.2 presents the penalties, range and variables utilized by each soft constraint. Note that some variables require additional indices: it is the case of those measuring the violation of soft constraints involving weekends, patterns or requests.

Table 3.2: *Ranged* and *logical* soft NRP constraints

Ranged soft constraint		Penalties	Range	Variables
1	min/max shifts assigned to a nurse	$\underline{\omega}_n^1$, $\overline{\omega}_n^1$	$\underline{\gamma}_n^1$, $\overline{\gamma}_n^1$	$\underline{\alpha}_n^1$, $\overline{\alpha}_n^1$
2	min/max consecutive free days	τ_{n,d_1,d_2}	-	r_{n,d_1,d_2}
3	min/max consecutive working days	σ_{n,d_1,d_2}	-	w_{n,d_1,d_2}
4	max working weekends in four weeks	$\overline{\omega}_n^4$	$\overline{\gamma}_n^4$	$\overline{\alpha}_{n,i}^4$
5	min/max consecutive working weekends	ψ_{n,i_1,i_2}	-	z_{n,i_1,i_2}
6	days off after series of night shifts	$\underline{\omega}_n^6$	$\underline{\gamma}_n^6$	$\underline{\alpha}_n^6$

Logical soft constraint		Penalties	Variables
7	complete weekends	σ_{n,d_1,d_2}	w_{n,d_1,d_2}
8	no night shift before free weekend	ω_n^8	$\alpha_{n,i}^8$
9	identical shift types during weekend	ω_n^9	α_{n,d_1,d_2}^9
10	alternative skill	$\nu_{n,s,d}$	$x_{n,s,d}$
11	unwanted patterns	ω_n^{11}	$\alpha_{n,\acute{p},d}^{11}$
12	day on/off request	ω_n^{12}	$\alpha_{n,\hat{p}}^{12}$
13	shift on/off request	$\nu_{n,s,d}$	$x_{n,s,d}$

There are $\mathcal{O}(|N| \times |D|^2)$ variables w and r, and therefore the formulation's total number of variables is affected more by the extension of the planing horizon than by the number of nurses. In practice, the number of variables is not expected

to increase considerably given that the planning horizon is generally one month in most hospitals. The same holds for variables z but instead of the number of days, the number of weekends is considered.

Objective Function

The objective function of the problem defined during the INRC-1 is presented by Equation 3.1, which minimizes the total penalty incurred by soft constraint violations. To facilitate reading the equation, components of the objective function are ordered according to the soft constraint indices (see Table 3.2). Note that the penalties for Soft Constraints 3 and 7 are aggregated (σ_{n,d_1,d_2}). Soft Constraints 10 and 13's penalties are also aggregated ($\nu_{n,s,d}$) and, therefore, the objective function consists of summing 11 components.

Minimize:

$$\sum_{n\in N}\left(\begin{array}{l}\left(\underline{\omega}^1_n\underline{\alpha}^1_n+\overline{\omega}^1_n\overline{\alpha}^1_n\right)+\sum_{(d_1,d_2)\in\Pi}\tau_{n,d_1,d_2}r_{n,d_1,d_2}+\sum_{(d_1,d_2)\in\Pi}\sigma_{n,d_1,d_2}w_{n,d_1,d_2}+\\ \sum_{i\in\{1,\ldots,\tilde{w}_c\}}\overline{\omega}^4_n\overline{\alpha}^4_{n,i}+\sum_{i_1,i_2\in\bar{W}_n:i_2\geq i_1}\psi_{n,i_1,i_2}z_{n,i_1,i_2}+\\ \sum_{d\in D}\underline{\omega}^6_n\underline{\alpha}^6_n+\sum_{i\in\{1,\ldots,\tilde{w}_c\}}\omega^8_n\alpha^8_{n,i}+\\ \sum_{i\in\{1,\ldots,\tilde{w}_c\}}\sum_{d_1\in\tilde{D}_{i,n}}\sum_{d_2\in\tilde{D}_{i,n}:d_2>d_1}\omega^9_n\alpha^9_{n,d_1,d_2}+\sum_{s\in S}\sum_{d\in D}\nu_{n,s,d}x_{n,s,d}+\\ \sum_{\acute{p}\in\acute{P}_n}\sum_{d\in\{1,\ldots,|D|-|\acute{p}|+1\}}\omega^{11}_n\alpha^{11}_{n,\acute{p},d}+\sum_{\hat{p}\in\hat{P}_n}\omega^{12}_n\alpha^{12}_{n,\hat{p}}\end{array}\right) \tag{3.1}$$

Constraints

The two hard constraints associated with the INRC-1 problem are formulated by Constraints (3.2) and (3.3): (i) to satisfy the nurse demand for every day and shift and (ii) to limit working shifts for nurses to a maximum of one per day.

$$\sum_{n \in N} x_{n,s,d} = \tilde{r}_{s,d} \qquad \forall d \in D, s \in S \tag{3.2}$$

$$\sum_{s \in S} x_{n,s,d} \leq 1 \qquad \forall n \in N, d \in D \tag{3.3}$$

Constraints (3.4)-(3.13) concern the activation of variables y, w, r and z. Constraints (3.4) and (3.5) ensure that $y_{n,i}$ equals one whenever nurse n works on weekend i. Constraints (3.6) and (3.7) link x, w and r variables. Constraints (3.8)-(3.10) ensure there is no overlap between working and resting windows, while guaranteeing that every active working window is immediately followed by a resting window. Constraints (3.11) ensure that at most one pattern expressed in z variables covers one weekend. Constraints (3.12) forbid the occurrence of two consecutive patterns expressed in z variables, since these should be combined into a single larger one, while Constraints (3.13) link variables y and z.

$$\sum_{s \in S} x_{n,s,d} \leq y_{n,i} \qquad \forall n \in N, i \in \tilde{W}_n, d \in \tilde{D}_{i,n} \tag{3.4}$$

$$\sum_{s \in S} \sum_{d \in \tilde{D}_{i,n}} x_{n,s,d} \geq y_{n,i} \qquad \forall n \in N, i \in \tilde{W}_n \tag{3.5}$$

$$\sum_{s \in S} x_{n,s,d} = \sum_{\substack{(d_1,d_2) \in \Pi: \\ d \in \{d_1,\ldots,d_2\}}} w_{n,d_1,d_2} \qquad \forall n \in N, d \in D \tag{3.6}$$

$$\sum_{s \in S} x_{n,s,d} = 1 - \left(\sum_{\substack{(d_1,d_2) \in \Pi: \\ d \in \{d_1,\ldots,d_2\}}} r_{n,d_1,d_2} \right) \qquad \forall n \in N, d \in D \tag{3.7}$$

$$\sum_{d' \in \{1,\ldots,d\}} w_{n,d',d} + \sum_{\substack{d'' \in D: \\ d'' \geq d+1}} w_{n,d+1,d''} \leq 1 \qquad \forall n \in N, d \in D \tag{3.8}$$

$$\sum_{d' \in \{1,\ldots,d\}} r_{n,d',d} + \sum_{\substack{d'' \in D: \\ d'' \geq d+1}} r_{n,d+1,d''} \leq 1 \qquad \forall n \in N, d \in D \tag{3.9}$$

$$\sum_{\substack{(d_1,d_2)\in\Pi:\\ d\in\{d_1,\ldots,d_2\}}} \Big(w_{n,d_1,d_2} + r_{n,d_1,d_2}\Big) = 1 \qquad \forall n \in N, d \in D \tag{3.10}$$

$$\sum_{i'\in\{1,\ldots,i\}} \sum_{i''\in\{i,\ldots,\tilde{W}\}} z_{n,i',i''} \leq 1 \qquad \forall n \in N, i \in \tilde{W} \tag{3.11}$$

$$\sum_{i'\in\{1,\ldots,i\}} z_{n,i',i} + \sum_{i''\in\{i+1,\ldots,\tilde{W}\}} z_{n,i+1,i''} \leq 1 \quad \forall n \in N, i \in \{1,\ldots,|\tilde{W}|-1\} \tag{3.12}$$

$$\sum_{i'\in\{1,\ldots,i\}} \sum_{i''\in\{i,\ldots,\tilde{W}\}} z_{n,i',i''} = y_{n,i} \qquad \forall n \in N, i \in \tilde{W} \tag{3.13}$$

Constraints (3.14)-(3.21) concern the computation of slack variable values responsible for measuring violations of soft constraints. As before, soft constraints are presented in accordance with the order defined in Table 3.2. Constraints (3.14) formulate the minimum and maximum working days in the planning horizon. Constraints (3.15) limit the number of working weekends within four weeks. Constraints (3.16) impose a minimum number of resting days after a sequence of night shifts. Constraints (3.17) ensure that a nurse is not allocated to a night shift on the day preceding a free weekend. Constraints (3.18) and (3.19) state that allocated shifts should be equal for every working day during a weekend. Finally, undesired patterns for shifts and days are formulated by Constraints (3.20) and (3.21), respectively.

$$\underline{\gamma}^1_n - \underline{\alpha}^1_n \leq \sum_{s\in S}\sum_{d\in D} x_{n,s,d} \leq \overline{\gamma}^1_n + \overline{\alpha}^1_n \qquad \forall n \in N \tag{3.14}$$

$$\sum_{i'\in\{i,\ldots,i+3\}} y_{n,i'} \leq \overline{\gamma}^4_n + \overline{\alpha}^4_{n,i} \qquad \forall n \in N, i \in \{1,\ldots,\tilde{w}_n - 3\} \tag{3.15}$$

$$\sum_{s\in\hat{S}} \sum_{d'\in\{d+1,\ldots,d+\underline{\gamma}^6_n\}} x_{n,s,d'} + \sum_{s'\in\tilde{S}} \underline{\gamma}^6_n x_{n,s',d} \leq \underline{\gamma}^6_n + \underline{\alpha}^6_{n,d}$$
$$\forall n \in N, d \in D : d \leq |D| - \underline{\gamma}^6_n \tag{3.16}$$

$$\sum_{s\in\tilde{S}} x_{n,s,l_{i,n}} - y_{n,i} \leq \alpha^8_{n,i} \qquad \forall n \in N, i \in \tilde{W}_n \tag{3.17}$$

$$\alpha^9_{n,d_1,d_2} \geq x_{n,s,d_1} - x_{n,s,d_2} \qquad \begin{array}{l}\forall n \in N, s \in S, i \in \tilde{W}_n,\\ (d_1,d_2) \in \tilde{D}_{i,n} : d_1 < d_2\end{array} \tag{3.18}$$

$$\alpha^{9}_{n,d_1,d_2} \geq x_{n,s,d_2} - x_{n,s,d_1} \qquad \begin{array}{l}\forall n \in N, s \in S, i \in \tilde{W}_n, \\ (d_1, d_2) \in \tilde{D}_{i,n} : d_1 < d_2\end{array} \tag{3.19}$$

$$\sum_{j \in \{1,\ldots,|\acute{p}|\}} x_{n,\acute{p}[j],d+j-1} \leq |\acute{p}| - 1 + \alpha^{11}_{n,\acute{p},d} \qquad \begin{array}{l}\forall n \in N, \acute{p} \in \acute{P}_n, \\ d \in \{1,\ldots,|D| - |\acute{p}| + 1\}\end{array} \tag{3.20}$$

$$\sum_{s \in S} \sum_{\substack{j \in \{1,\ldots,|\hat{p}|: \\ \hat{p}[j] \geq 1\}}} x_{n,s,\hat{p}[j]} + \sum_{\substack{j \in \{1,\ldots,|\hat{p}|: \\ \hat{p}[j] \leq -1\}}} \left(1 - \sum_{s \in S} x_{n,s,-\hat{p}[j]}\right) \leq |\hat{p}| - 1 + \alpha^{12}_{n,\hat{p}}$$
$$\forall n \in N, \hat{p} \in \hat{P}_n \tag{3.21}$$

3.2.1 Computational experiments

Formulation (3.1)-(3.21) was evaluated using the solvers CPLEX 12.7 and Gurobi 7.5 on an Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz computer with 128Gb of RAM memory running Linux Ubuntu 16.04.2 LTS. Table 3.3 details the results obtained by these state-of-the-art solvers. Computational time was restricted to one hour. The final lower (LB) and upper (UB) bounds are presented together with the computed optimality gap $\frac{UB-LB}{UB} \times 100$ and the required runtime. All instances proposed during the INRC-1 were considered. However, since both CPLEX and Gurobi quickly obtained the optimal solution for all *sprint* instances, results for these instances are omitted.

Results in Table 3.3 indicate that although there are large instances which are easy for CPLEX and Gurobi – optimality was proven in under 10 minutes – there exist instances for which CPLEX could not reach any feasible solution within one hour of runtime. It is noteworthy how Gurobi performed better than CPLEX when employing Formulation (3.1)-(3.21), being capable of obtaining feasible solutions for all instances. Nevertheless, optimality gaps of over 30% are still observed for *long hidden*, *long hint* and *medium hidden* instances. These results serve to show that, despite the recent progresses in generic IP solvers, in many real-world applications the hybridization of these solvers with methods which consider problem-specific information remains critical.

Table 3.3: Experiments with Formulation (3.1)-(3.21)

Instance		Model Dimensions			CPLEX 12.7				Gurobi 7.5			
		Vars	Constrs	Non-zeros	LB	UB	Gap	Time	LB	UB	Gap	Time
long early	01	52,729	17,241	1,012,492	197.0	197	0.0%	32.1s	197.0	197	0.0%	37.3s
	02	52,803	17,241	1,012,566	219.0	219	0.0%	3.0m	219.0	219	0.0%	9.9m
	03	52,838	17,241	1,012,601	240.0	240	0.0%	28.6s	240.0	240	0.0%	41.1s
	04	52,831	17,241	1,012,594	303.0	303	0.0%	22.6s	303.0	303	0.0%	31.3s
	05	52,789	17,241	1,012,552	284.0	284	0.0%	25.0s	284.0	284	0.0%	25.8s
long hidden	01	63,205	28,370	1,065,275	319.0	373	14.5%	1.0h	341.0	347	1.7%	1.0h
	02	63,205	28,370	1,065,275	81.0	117	30.8%	1.0h	81.0	89	9.0%	1.0h
	03	63,620	29,210	1,068,150	25.4	-	-	1.0h	19.0	42	54.8%	1.0h
	04	63,205	28,370	1,065,275	13.8	-	-	1.0h	21.0	22	4.5%	1.0h
	05	62,880	27,530	1,062,510	41.0	41	0.0%	56.6m	41.0	41	0.0%	30.0m
long late	01	63,005	27,875	1,063,670	175.4	284	38.3%	1.0h	232.0	241	3.7%	1.0h
	02	63,005	27,875	1,063,670	88.0	270	67.4%	1.0h	229.0	229	0.0%	50.3m
	03	63,005	27,875	1,063,670	72.0	695	89.6%	1.0h	218.0	220	0.9%	1.0h
	04	63,005	27,875	1,063,670	72.0	259	72.2%	1.0h	216.0	222	2.7%	1.0h
	05	62,631	27,243	1,061,472	79.5	139	42.8%	1.0h	83.0	84	1.2%	1.0h
long hint	01	62,820	27,480	1,062,280	18.0	38	52.6%	1.0h	29.0	39	25.6%	1.0h
	02	59,551	23,990	1,051,461	9.0	35	74.3%	1.0h	17.0	17	0.0%	46.3m
	03	59,620	23,990	1,051,450	36.0	141	74.5%	1.0h	43.0	65	33.8%	1.0h
medium early	01	30,279	8,668	622,441	240.0	240	0.0%	1.5m	240.0	240	0.0%	17.8s
	02	30,309	8,668	622,471	240.0	240	0.0%	31.6s	240.0	240	0.0%	22.2s
	03	30,309	8,668	622,471	236.0	236	0.0%	30.9s	236.0	236	0.0%	29.2s
	04	30,273	8,668	622,435	237.0	237	0.0%	1.3m	237.0	237	0.0%	1.6m
	05	30,348	8,668	622,510	303.0	303	0.0%	33.9s	303.0	303	0.0%	22.6s
medium hidden	01	37,415	16,070	635,725	72.7	161	54.8%	1.0h	86.0	138	37.7%	1.0h
	02	37,415	16,070	635,725	191.4	328	41.6%	1.0h	202.0	237	14.8%	1.0h
	03	37,415	16,070	635,725	25.1	53	52.7%	1.0h	29.0	39	25.6%	1.0h
	04	37,415	16,070	635,725	62.1	111	44.1%	1.0h	67.0	86	22.1%	1.0h
	05	37,415	16,070	635,725	85.3	215	60.3%	1.0h	99.0	138	28.3%	1.0h
medium late	01	34,850	14,062	623,360	154.6	157	1.5%	1.0h	155.0	157	1.3%	1.0h
	02	34,814	14,062	623,352	18.0	18	0.0%	2.3m	18.0	18	0.0%	9.7m
	03	29,486	8,872	603,434	29.0	29	0.0%	7.2m	29.0	29	0.0%	5.4m
	04	34,770	13,902	622,880	35.0	35	0.0%	12.9m	35.0	35	0.0%	14.0m
	05	35,810	14,450	630,920	107.0	107	0.0%	20.5m	107.0	107	0.0%	28.4m
med. hint	01	34,886	14,062	623,384	34.0	34	0.0%	59.0m	34.0	34	0.0%	17.8m
	02	34,814	14,062	623,352	66.9	74	9.6%	1.0h	72.0	72	0.0%	41.0m
	03	34,886	14,062	623,384	102.9	119	13.5%	1.0h	113.0	115	1.7%	1.0h

3.3 Dantzig-Wolfe decomposition

Formulation (3.1)-(3.21), proposed throughout the previous section, has the ideal structure for applying Dantzig-Wolfe decomposition. The original problem is decomposed into a master problem and $|N|$ pricing problems, one for each nurse.

Figure 3.1 presents the coefficient matrix of the original LP (left image) of Formulation (3.1)-(3.21) on a NRP instance with six nurses and the same LP

after sorting the rows and columns by nurse (right image). The dots indicate non-zero coefficients in the constraint matrix. Each block corresponds to the variables and constraints associated with a nurse. Note how similar this figure is to Figure 2.2, presented in the previous chapter on page 20. Indeed, the formulations considered for the NRP and TUP have very similar structures, both exhibiting the required block structure for Dantzig-Wolfe decomposition.

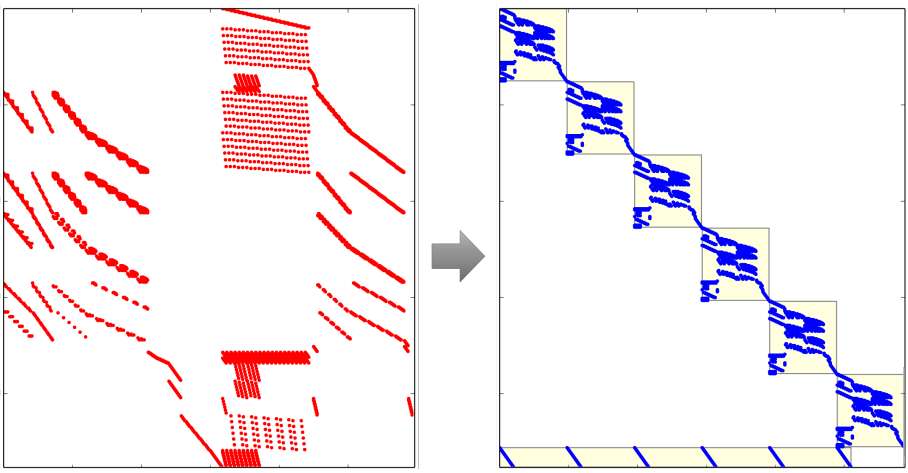


Figure 3.1: Representation of Dantzig-Wolfe's decomposition

In Formulation (3.1)-(3.21), all constraints except the cover ones (Constraints (3.2)) are nurse-oriented. The decomposition is thus straightforward. Each pricing problem concerns finding the optimal allocations (schedule) for one nurse subject to dual costs on each shift and day. Note that the pricing problems are not identical, since nurses have different preferences and contracts. The master problem is given by Formulation (3.22)-(3.25), where Ω_n is the set of all possible schedules for nurse n, c_k the cost (violation of soft constraints) of schedule k, and $a_{s,d,k}$ is a parameter indicating whether the nurse of schedule k is assigned to shift s of day d ($a_{s,d,k} = 1$) or not ($a_{s,d,k} = 0$). Binary variable λ_k defines whether schedule k is selected ($\lambda_k = 1$) or not ($\lambda_k = 0$).

Minimize:

$$\sum_{n \in N} \sum_{k \in \Omega_n} c_k \lambda_k \tag{3.22}$$

Subject to:

$$\sum_{k \in \Omega_n} \lambda_k = 1 \qquad \forall n \in N \tag{3.23}$$

$$\sum_{n \in N} \sum_{k \in \Omega_n} a_{s,d,k} \lambda_k = \tilde{r}_{s,d} \qquad \forall d \in D, s \in S \tag{3.24}$$

$$\lambda_k \in \{0, 1\} \qquad \forall n \in N, k \in \Omega_n \tag{3.25}$$

3.3.1 Column generation

Formulation (3.22)-(3.25) has an exponential number of variables (schedules), which prohibits its complete generation. Rather than generating all possible schedules, column generation is applied to solve the formulation's linear relaxation. As previously explained in Section 2.3.1, pricing problems are solved at each iteration to obtain columns (here schedules) with negative reduced cost, which are subsequently added to the master problem. A negative reduced cost column for nurse n is a column $k \in \Omega_n$ for which $v_n + \sum_{d \in D} \sum_{s \in S} a_{s,d,k} w_{s,d} > c_k$, where v_n and $w_{s,d}$ represent the dual values corresponding to Constraints (3.23) and (3.24), respectively, and where c_k corresponds to the cost of schedule k, given by penalties of violated soft constraints. If such columns are found, they are added to the master problem, which is subsequently re-solved. The algorithm continues until no column with negative reduced cost exists, in which case the linear relaxation of the reduced master problem is solved.

The pricing problems are solved employing Formulation (3.1)-(3.21) with Constraints (3.2) relaxed. Note that each pricing problem corresponds to a single nurse, and therefore only one nurse is considered within the formulation. Additionally, the objective function is modified so as to produce the most negative reduced cost column.

3.3.2 Computational experiments

Experiments were conducted employing column generation to solve the linear relaxation of Formulation (3.22)-(3.25). The goal is to obtain stronger lower bounds, thereby enabling more precise heuristic solution quality evaluation. The column generation procedure was implemented in Java and, again, experiments were executed on Core i7 3.4GHz computers with 16Gb of RAM memory running Linux Ubuntu 12.04. CPLEX was employed to solve both the reduced master LP and pricing IP problems.

Note that the approach evaluated here has many similarities with that proposed by Burke and Curtois (2014). However, Burke and Curtois (2014) did not report results for the challenging *hidden* instances due to compatibility issues between their code and these instances. Since these are the instances for which CPLEX and Gurobi obtained poor results employing Formulation (3.1)-(3.21) (see Section 3.2.1), we implemented and evaluated the column generation in order to improve lower bounds.

Table 3.4: Column generation results for NRP instances

Instance		LB*	Col. Gen.	
			LB	Time
long early	01	197	197	498s
	02	219	219	542s
	03	240	240	497s
	04	303	303	484s
	05	284	284	486s
long hidden	01	341	345	508s
	02	86	89	584s
	03	36	38	606s
	04	19	22	585s
	05	41	41	653s
long late	01	235	235	498s
	02	229	229	547s
	03	219	219	500s
	04	221	221	602s
	05	83	83	513s
long hint	01	29	30	685s
	02	17	17	647s
	03	43	51	581s
medium early	01	240	240	285s
	02	240	240	278s
	03	236	236	279s
	04	237	237	294s
	05	303	303	342s
medium hidden	01	88	96	737s
	02	197	213	640s
	03	28	34	782s
	04	73	76	865s
	05	91	118	485s
medium late	01	156	156	970s
	02	18	18	568s
	03	29	29	330s
	04	35	35	535s
	05	107	107	1559s
med. hint	01	34	34	652s
	02	72	72	1446s
	03	113	114	1314s

Table 3.4 presents the best known lower bound (LB*) and those obtained by the column generation (LB) together with the required runtime, in seconds. Note that lower bounds were rounded up, remaining valid since every feasible solution for the addressed NRP has integer objective values. Multiple instances had their best known lower bound improved. However, the average runtime to

solve the relaxation is long for proceeding with a branch-and-price algorithm for large NRP instances. The current algorithm's bottleneck is the long runtime required by CPLEX to solve pricing problems. Indeed, employing a general IP solver to handle the pricing is not the best strategy for the NRP, given the better performance of the pricing solver proposed by Burke and Curtois (2014). Nevertheless, solving the column generation for all instances was useful: tight, improved lower bounds were generated.

3.4 Decomposition-based heuristic

A heuristic subproblem optimization scheme is proposed in which Formulation (3.1)-(3.21) is employed to solve NRP subproblems. Heuristic rules are applied to decompose the problem creating subproblems defined by the fixation of certain variables of the original problem. The algorithm essentially consists of two subsequent phases: construction and local search phase.

The construction phase builds a feasible initial solution using a straightforward greedy algorithm. An allocation matrix M with $|N|$ rows and $|D|$ columns is created (as in Table 3.1), with all $M_{n,d}$ cells initially set to days off. Then, for each day d and shift s, the demand $\tilde{r}_{s,d}$ is satisfied by selecting the nurse n for which the allocation incurs the smallest increase (or largest decrease) in the objective function of the current solution. The solution is updated and this process is repeated until all demands are satisfied. Since $|N|$ nurses are analyzed for each demand, the resulting algorithm has time complexity $\mathcal{O}(\tilde{R} \times |N|)$, where $\tilde{R}$ represents the total demand for nurses, meaning $\tilde{R} = \sum_{s \in S} \sum_{d \in D} \tilde{r}_{s,d}$.

Following the generation of the initial solution, a local search algorithm employing different decompositions is applied to improve this solution. The decomposition scheme and resulting local search algorithm are discussed in the following section.

3.4.1 Decomposition scheme

The local search phase explores the search space by solving subproblems defined by different decomposition approaches, inspired by the three dimensions of variables x: *time* (or days), *nurses* and *shifts*. Given a feasible solution S, a neighbor is obtained in three steps: (i) selecting the subproblem to be solved,

(*ii*) fixing allocations in S which are not part of the selected subproblem and, finally, (*iii*) solving to optimality the subproblem generated by these fixations. The differences between the subproblems lie in the rules employed to generate them. Note that in contrast to Santos et al. (2016), additional decompositions are considered and subproblems are solved in any order. However, precautions must be taken to prevent solving twice subproblems which cannot improve the solution. The three decompositions considered are explained in the following paragraphs.

Time-based decomposition

The *Time*-based decomposition exploits the NRP's time structure to decompose the problem. NRP subproblems are generated by fixing all nurse allocations of $|D| - \eta^t$ days, where η^t is a parameter defining the number of unfixed days. Generally, consecutive days are considered, so as to include constraints concerning consecutive working (or resting) days. Figure 3.2 depicts an example of the *Time*-based decomposition for the NRP.

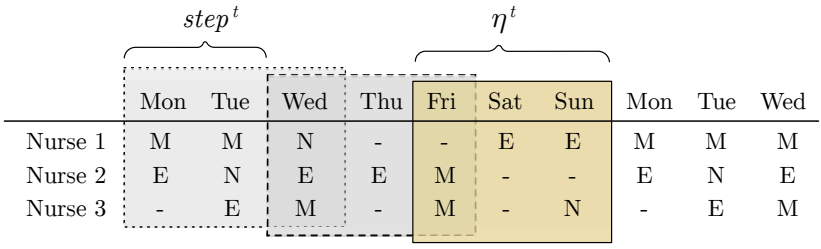


Figure 3.2: *Time*-based decomposition with $\eta^t = 3$ and $step^t = 2$

The decomposition has two parameters: $step^t$ and η^t. The smaller the value of $step^t$, the greater the number of different subproblems. Parameter $step^t$ also indicates the degree of overlap between subproblems. The other parameter, η^t, defines the size of each subproblem and is therefore critical. Small values may result in subproblems which do not contain any better solution while large values may create unmanageably large subproblems. Note how similar this decomposition is to that defined for the TUP in Section 2.5, sharing similar principles and parameters.

Nurse-based decomposition

The *Nurse*-based decomposition is very similar to the *Time*-based decomposition. NRP subproblems are generated by fixing $|N| - \eta^n$ nurse allocations for all days, where η^n is a parameter which defines the number of unfixed nurses. Ideally, all nurse combinations of size η^n would be considered. This would result, however, in a very large number of subproblems. To circumvent this issue and therefore limit the number of subproblems, a strategy similar to that employed for the *Time*-based decomposition is utilized. Figure 3.3 presents this strategy and depicts an example of the *Nurse*-based decomposition to the NRP.

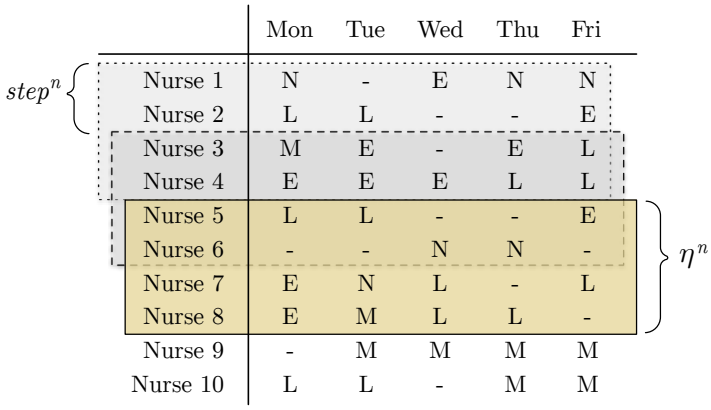


Figure 3.3: *Nurse*-based decomposition with $\eta^n = 4$ and $step^n = 2$

Similarly to the *Time*-based decomposition, parameters η^n and $step^n$ are employed. However, in contrast to the *Time*-based decomposition, there is generally little or no relation between consecutive nurses. The ordering of nurses is, therefore, important to define which subproblems the decomposition will generate. Generally, a random ordering of nurses is considered by the algorithm.

Shift-based decomposition

In the *Shift*-based decomposition, subproblems are created by fixing all the allocations of $|S| - 1$ shifts. The decomposition does not rely on any specific parameter and always results in $|S|$ different subproblems.

Given that an average instance has three to five shifts, it may appear that the subproblems generated by this decomposition are hard to solve. However, counter-intuitively, such subproblems may actually be solved in short runtimes.

3.4.2 Heuristic algorithm

The local search heuristic consists of solving the subproblems defined by the decomposition strategies presented in the previous section. Algorithm 3.1 presents the heuristic approach. Seven arguments are required: the initial solution S_0; parameters for the *Time*-based decomposition η^t, $\bar{\eta}^t$ and $step^t$; and parameters for the *Nurse*-based decomposition η^n, $\bar{\eta}^n$ and $step^n$. Note that $\bar{\eta}^t$ and $\bar{\eta}^n$ specify upper limits for η^t and η^n values, respectively.

Algorithm 3.1: Decomposition-based local search algorithm

Input: initial solution S_0 and parameters for the decompositions
LocalSearch(S_0, η^t, $\bar{\eta}^t$, $step^t$, η^n, $\bar{\eta}^n$, $step^n$)

1 $S \leftarrow S_0$
2 $k \leftarrow 0$ // counter of solution improvements
3 $\mathcal{L} \leftarrow$ *Time*-based, *Nurse*-based and *Shift*-based decomposition subproblems considering parameters η^t, $step^t$, η^n and $step^n$
4 **while** $\mathcal{L} \neq \emptyset$ **and** time limit not reached **do**
5 $P \leftarrow$ random subproblem from $\mathcal{L}$
6 $\mathcal{L} \leftarrow \mathcal{L} \backslash \{P\}$
7 solve subproblem P and update solution S
8 **if** solution S was improved **then**
9 $k \leftarrow k + 1$ // updating solution improvements counter
10 **if** $\mathcal{L} = \emptyset$ **and** ($k > 0$ **or** $\eta^t < \bar{\eta}^t$ **or** $\eta^n < \bar{\eta}^n$) **then**
11 $k \leftarrow 0$ // resetting solution improvements counter for new subproblems $\mathcal{L}$
12 update values for η^t, $step^t$, η^n and $step^n$
13 shuffle list of nurses // for *Nurse*-based decomposition
14 $\mathcal{L} \leftarrow$ *Time*-based, *Nurse*-based and *Shift*-based decomposition subproblems considering parameters η^t, $step^t$, η^n and $step^n$
15 **return** S

The algorithm begins by setting solution S to the constructive one (line 1) and counter k to zero (line 2). Then list $\mathcal{L}$ is created incorporating all subproblems from the considered decompositions (line 3). Next the main loop begins, and is executed until there are no subproblems to solve or a specified time limit is reached (line 4). A random subproblem $P \in \mathcal{L}$ is selected (line 5), being subsequently removed from $\mathcal{L}$ (line 6). Subproblem P is then solved, updating

solution S (line 7). If S is improved, counter k is incremented (lines 8-9). Once $\mathcal{L}$ is empty, meaning all its subproblems were solved, updated subproblems may be created. $\mathcal{L}$ is populated with subproblems if solution S was improved ($k > 0$) or if some η or *step* value may be updated, meaning that novel subproblems will be generated (line 10). In such cases, k is reset (line 11) and values for η and *step* are updated for all decompositions (line 12). Generally η is incremented and *step* is set to half the value of η. As part of the *Nurse*-based decomposition, the order of nurses are shuffled (line 13). Afterwards, $\mathcal{L}$ is recreated including all subproblems from the considered decompositions (line 14). Finally, when the main loop finishes, the updated solution S is returned (line 15).

It is noteworthy that if no time limit is set and $\bar{\eta}^t$ equals $|D|$ (or $\bar{\eta}^n$ equals $|N|$), the original problem will be solved during the heuristic's final iteration.

3.4.3 Computational experiments

The decomposition-based heuristic was coded in C++ and compiled on GCC/G++ version 4.8.4. The source code is available online[5], in addition to all models and solutions[6]. All experiments were executed on an Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz computer with 128Gb of RAM memory running Linux Ubuntu 16.04.2 LTS. In contrast to experiments reported in Section 3.2.1, CPLEX 12.7 demonstrated slightly better performance than Gurobi 7.5 when solving subproblems, and was therefore employed throughout the experiments. Furthermore, parallel mode was disabled so that exactly one CPU is utilized by the solver.

The instance set proposed during the INRC-1 was employed within the experiments. The algorithm parameters were tuned using the iRace package (López-Ibáñez et al., 2011) with a budget of 500 runs and considering a subset of instances containing one random instance per category (8 instances in total). The following values were suggested by iRace and employed during experiments: $\eta^t = 4$, $\bar{\eta}^t = 30$, $step^t = 3$, $\eta^n = 2$, $\bar{\eta}^n = 5$ and $step^n = 4$.

Tables 3.5 and 3.6 report, for each instance, the best known solutions (BKS) and those reported by multiple papers from the literature together with the best and average results obtained by the decomposition-based heuristic. The proposed method was executed 10 times for each instance with different random seeds,

[5] https://www.github.com/tuliotoffolo/nrp
[6] https://benchmark.gent.cs.kuleuven.be/nrp/

Table 3.5: Results of decomposition-based heuristic on instances of set *long*

Instance		BKS	Nonobe (2010)	Bilgin et al. (2012)	Lü and Hao (2012)	Valouxis et al. (2012)	Burke and Curtois (2014)	Tassopoulos et al. (2015)	Santos et al. (2016)	Rahimian et al. (2017)	Decomposition-based heuristic	
											Best	Average
long early	01	197	197	197	197	197	197	197	197	197	197	197.0
	02	219	219	220	222	219	219	219	219	219	219	219.0
	03	240	240	240	240	240	240	240	240	240	240	240.0
	04	303	303	303	303	303	303	303	303	303	303	303.0
	05	284	284	284	284	284	284	284	284	284	284	284.0
long hidden	01	346	-	-	346	363	-	349	346	488	346	346.6
	02	89	-	-	89	90	-	89	89	101	89	89.1
	03	38	-	-	38	38	-	38	38	59	38	39.4
	04	22	-	-	22	22	-	22	22	38	22	22.0
	05	41	-	-	45	41	-	41	41	114	41	43.0
long late	01	235	235	241	237	235	235	239	235	332	235	237.6
	02	229	229	245	229	229	229	234	229	289	229	230.8
	03	220	220	233	222	220	220	227	220	320	220	220.3
	04	221	221	246	227	221	221	232	222	320	221	223.3
	05	83	83	87	83	83	83	83	83	156	83	83.0
long hint	01	31	-	33	-	-	-	-	-	164	**30**	31.4
	02	17	-	17	-	-	-	-	-	96	17	17.0
	03	53	-	55	-	-	-	-	-	272	**51**	51.9

Table 3.6: Results of decomposition-based heuristic on instances of set *medium*

Instance		BKS	Nonobe (2010)	Bilgin et al. (2012)	Lü and Hao (2012)	Valouxis et al. (2012)	Burke and Curtois (2014)	Tassopoulos et al. (2015)	Santos et al. (2016)	Rahimian et al. (2017)	Decomposition-based heuristic	
											Best	Average
medium early	01	240	241	242	240	240	240	240	240	240	240	240.0
	02	240	240	241	240	240	240	240	240	240	240	240.0
	03	236	236	238	236	236	236	236	236	236	236	236.0
	04	237	238	238	237	237	237	237	237	237	237	237.0
	05	303	304	304	303	303	303	303	303	303	303	303.0
medium hidden	01	111	-	-	117	130	-	122	111	192	111	116.0
	02	220	-	-	220	221	-	221	221	312	**219**	220.3
	03	34	-	-	35	36	-	34	34	73	34	34.4
	04	78	-	-	79	81	-	79	78	124	78	80.8
	05	119	-	-	119	122	-	124	119	174	**118**	120.4
medium late	01	157	176	163	164	158	157	161	157	159	157	159.6
	02	18	19	21	20	18	18	19	18	18	18	18.1
	03	29	30	32	30	29	29	30	29	29	29	29.0
	04	35	37	38	36	35	35	35	35	35	35	35.0
	05	107	125	122	117	107	107	112	107	107	107	111.2
med. hint	01	34	-	40	-	-	-	-	-	39	34	34.8
	02	72	-	91	-	-	-	-	-	80	72	73.8
	03	117	-	134	-	-	-	-	-	121	**115**	117.7

always respecting the runtime limit of 10 minutes. Note that the comparison against the best results from literature ignores the difference concerning order of magnitude in terms of runtime for *long* instances: Valouxis et al. (2012) report, for example, runtimes of up to 600 minutes for these instances. The number of executions is also ignored by the comparison. Santos et al. (2016), for instance, present the best results obtained from a large set of experiments considering different parameters. The results for *sprint* instances are not presented since these are no longer challenging instances: the heuristic was capable of very quickly finding optimal solutions for all instances on all executions.

The tables demonstrate how the proposed decomposition-based heuristic obtained all best known solutions for the problem, improving the results for five instances. Since the heuristic was able to robustly find good solutions within 10 minutes of sequential processing time, results for longer runs are not reported. This is justified since running the heuristic for longer times could result in solving the original problem (see Section 3.4.2), which would take a prohibitively long runtime for some instances.

3.4.4 Best results

A summary of the results is presented in Table 3.7. The table displays the previous best known solution (column BKS), the best lower bound (LB) obtained

Table 3.7: Previous upper bounds and updated lower and upper bounds

Instance		BKS	LB	UB	Gap
long early	01	197	197	197	0.0
	02	219	219	219	0.0
	03	240	240	240	0.0
	04	303	303	303	0.0
	05	284	284	284	0.0
long hidden	01	346	345	346	0.3
	02	89	89	89	0.0
	03	38	38	38	0.0
	04	22	22	22	0.0
	05	41	41	41	0.0
long late	01	235	235	235	0.0
	02	229	229	229	0.0
	03	220	219	220	0.5
	04	221	221	221	0.0
	05	83	83	83	0.0
long hint	01	31	30	30	0.0
	02	17	17	17	0.0
	03	53	51	51	0.0

Instance		BKS	LB	UB	Gap
medium early	01	240	240	240	0.0
	02	240	240	240	0.0
	03	236	236	236	0.0
	04	237	237	237	0.0
	05	303	303	303	0.0
medium hidden	01	111	96	111	13.5
	02	220	213	219	2.7
	03	34	34	34	0.0
	04	78	76	78	2.6
	05	119	118	118	0.0
medium late	01	157	156	157	0.6
	02	18	18	18	0.0
	03	29	29	29	0.0
	04	35	35	35	0.0
	05	107	107	107	0.0
med. hint	01	34	34	34	0.0
	02	72	72	72	0.0
	03	117	114	115	0.9

either by the column generation approach or one of the state-of-the-art IP solvers, the best solution generated by the decomposition-based heuristic and, finally, the current optimality gap $\frac{UB-LB}{UB} \times 100$. Most instances were solved to optimality and the proposed heuristic was capable of improving some best known solutions. Currently, the most difficult among the unsolved instances is '*medium hidden 01*', for which the optimality gap is 13.5%.

3.5 Conclusions and future work

This chapter proposed a decomposition-based heuristic for the NRP introduced on the occasion of the INRC-1 based on a compact IP formulation. Additionally, Dantzig-Wolfe decomposition was applied to the IP formulation, with the resulting formulation being solved by column generation. As with Chapter 2, particular attention was assigned to both the computation of strong lower bounds and the fast production of high-quality solutions.

The compact IP formulation solved all *sprint* instances in addition to some instances of the *medium* and *large* benchmark sets, but did not provide good bounds and solutions for the challenging instances. Solving the linear relaxation of the Dantzig-Wolfe reformulation resulted in better lower bounds, but generally no integer solution was produced. A decomposition-based heuristic algorithm was then proposed, built upon the compact IP formulation. The algorithm was evaluated with the state-of-the-art CPLEX solver, obtaining the best known solutions for *all* INRC-1 instances, requiring very short computational times and outperforming the best heuristics for the problem in terms of both performance and solution quality. Among these solutions, five improve upon the best known solutions from the literature. These results, when coupled with the bounds produced by the Dantzig-Wolfe reformulation, enabled proving optimality for solutions of previously unsolved instances.

The decomposition strategy utilized within the decomposition-based heuristic resembles the one applied to the TUP in Chapter 2. In fact, exploiting the time-structure of the problem has proven a simple yet successful decomposition approach for deriving competitive subproblem optimization heuristics for both problems.

Chapter 4

Project Scheduling Problem

This chapter addresses a generalization of the *Project Scheduling Problem* (PSP), a problem which is the subject of several studies throughout computer science, mathematics and operations research, given its difficulty and practical importance. This generalization encompasses multiple projects, multiple job execution modes, precedence constraints, renewable and non-renewable resource constraints and resource sharing among different projects. A solution for this problem consists of a feasible job schedule which does not exceed the stipulated limits of renewable and non-renewable resources while respecting precedence constraints between jobs. A set of execution modes must be selected given that job duration and the quantity of required resources varies depending upon the selected mode. The primary objective is to reduce the sum of the projects' duration. A secondary objective is also defined: to reduce the last project's completion time.

This chapter builds upon Toffolo et al. (2016b)[1], which is partially reproduced here together with the novel content, algorithm and results. First, Section 4.1 introduces the addressed problem and presents a literature review concerning it. Next, an IP formulation is presented in Section 4.2. The formulation is then evaluated employing state-of-the-art solvers and instances from a benchmark set. Despite successfully modeling all problem constraints, the formulation's performance within solvers is very poor, being incapable of

[1] **Toffolo, T. A. M.**, Santos, H. G., Carvalho, M. A. M., and Soares, J. A. (2016). An integer programming approach to the multimode resource-constrained multiproject scheduling problem. *Journal of Scheduling*, 19(3):295–307.

producing any solution in most cases. To produce feasible, high-quality solutions, a decomposition-based heuristic algorithm is proposed in Section 4.3. Differently from the decomposition-based heuristics proposed for the TUP and the NRP (Chapters 2 and 3), here the subproblem optimization is embedded within a metaheuristic framework. The resulting algorithm and its components, which include multiple IP formulations, are described in detail. Computational experiments evaluating the heuristic are presented in Section 4.4. Given that a generalization of the PSP is addressed in this chapter, a special case of the problem is also considered for evaluation. Several instances had their best known result improved, despite the short runtime. Finally, Section 4.5 presents conclusions and future work directions.

4.1 Introduction

A PSP, in its general form, consists of scheduling the processing times of *jobs* (tasks or activities) contained in a project. These jobs are interrelated by precedence constraints, that is, a job may require another to have concluded before its start. This class of problem models many general and practical situations occurring throughout engineering and management sciences, being tackled by experts of various fields ranging from civil engineering to software development. One natural generalization of the PSP is the *Resource-Constrained Project Scheduling Problem* (RCPSP), which includes constraints concerning resource consumption during each job's processing. The RCPSP is $\mathcal{NP}$-hard in the strong sense (Błażewicz et al., 1983) and was claimed to be "one of the most intractable problems in Operations Research" by Möhring et al. (2003).

Various Integer Programming (IP) formulations for the RCPSP are found in the literature. Pritsker et al. (1969) proposed a binary programming formulation where variables $x_{j,t}$ indicate whether job j is scheduled at time t ($x_{j,t} = 1$). In this formulation, known as *discrete-time*, the number of binary decision variables depends upon an upper limit $\bar{t}$ for the number of timeslots required to complete the project, which can be heuristically defined. Therefore, the number of variables in this formulation is $\mathcal{O}(n \times \bar{t})$, where n represents the number of jobs. Kolisch and Hartmann (2006) extended this formulation to handle different execution modes, introducing an additional index to the binary variables. Koné et al. (2011) proposed an event-based formulation called OOE (On/Off Event-based), where an event corresponds to the starting time of one or more jobs. General integer variables t_e are employed to indicate the starting

time of each event e. These variables are linked to binary variables $x_{j,e}$ which indicate whether task j starts or is still being processed during event e. As the maximum number of events is equal to the number of jobs n, this formulation has $\mathcal{O}(n^2)$ variables. However, the number of constraints in this formulation is $\mathcal{O}(n^3)$, since it is necessary to link activities to their start and end events. This renders the formulation less practical in terms of being solved by IP solvers.

There are many other generalizations of the PSP and RCPSP. For comprehensive research concerning project scheduling problems' historical origins, classification, complexity analysis and solution methods, the reader is referred to Weglarz (1999), Klein (2000), Demeulemeester and Herroelen (2002), Hartmann and Rieger (2002), Józefowska and Weglarz (2006) and Artigues et al. (2013). Additionally, a comprehensive classification and computational analysis of heuristics and metaheuristics applied to the RCPSP is provided by Kolisch and Hartmann (1999, 2006).

The PSP considered throughout this chapter is a more recent RCPSP generalization: the *Multi-Mode Resource-Constrained Multi-Project Scheduling Problem*, which is henceforth referred to as the *Generalized Project Scheduling Problem* (GPSP). This problem was selected as the subject of the *Multidisciplinary International Scheduling Conference: Theory and Applications* (MISTA) Challenge 2013 (Wauters et al., 2016).

The GPSP consists of a set P of projects, where each project $p \in P$ is composed of a set J_p of non-preemptive jobs, which must be scheduled. Each project p is also associated with a *release time*, that is, a time when its jobs' processing may be started. The start and end of a project are delimited by *dummy* jobs j_p^- and j_p^+, the first and last jobs of each project, respectively.

Scheduling a project means determining the starting time of its jobs while respecting precedence constraints and resource availability. Jobs may consume *local resources* (resources exclusive to a project) and *global resources* (resources shared among all projects). These resources can be either (*i*) *renewable*, with capacity renewed every timeslot (working hours, for example) or (*ii*) *non-renewable*, with a fixed initial capacity (money, for example). Each job may be executed on one or more *execution modes*, each consuming a specific quantity of resources and resulting in different job completion durations. Note that dummy jobs do not consume any resources and their duration is always zero.

One practical example of a problem that can be modeled as a GPSP is software project management. Assume a software company is responsible

for the development and maintenance of multiple softwares (projects) thereby requiring the execution of tasks which consume resources that may be non-renewable (money, for example) and renewable (employee hours). The tasks within these projects may be executed in different modes depending on the resources assigned. Therefore, task durations depend on their execution modes. Solving this practical example was considered by Kerzner (2013) to constitute one of the most important responsibilities of a project manager, i.e. to plan the integration and execution of the tasks. In fact, the application of optimization techniques to software engineering has received increasing attention throughout the academic community, creating a new field called Search-Based Software Engineering. In the work of Alba and Chicano (2007), for instance, resources represent employees, each associated with a set of skills and wages, plus a maximum degree of dedication to a project. They used a genetic algorithm to obtain solutions minimizing the cost and time of software projects.

Recently some methodologies were proposed for the GPSP. Asta et al. (2016), the MISTA Challenge 2013 winners, proposed an approach combining Monte-Carlo and Hyper-Heuristics along with several neighborhoods which are explored by stochastic local search. Their approach employs an indirect solution representation where solutions are always decoded by a constructive algorithm. To speed up this constructive algorithm a prefix matching method is employed. In the construction phase, modes are randomly selected and feasibility for non-renewable resources is achieved by stochastic local search. Several neighborhoods which perform large modifications in the solution were developed, including one in which allocation priorities are changed for all tasks of a given project.

Geiger (2013), who ranked second in the challenge, proposed an Iterative Variable Neighborhood Search for the GPSP which explores the solution space through systematic exchanges of neighborhood structures (Hansen and Jaumard, 1997). This approaches considers a set of feasible schedules X which is associated with two vectors, $M = \{m_1, ..., m_n\}$ and $S = \{S_1, ..., S_n\}$. M represents the execution mode chosen for each job and S the permutation of job indices. Initial modes are randomly assigned to M. If M is not feasible considering non-renewable resources, a procedure which randomly changes modes is applied until feasibility is reached. If feasibility is not achieved, M is rebuilt from scratch. Later, sequence S is built, assigning higher priority to activities with earlier starting times. The local search is performed in parallel. Once a local optimum is reached, the best solution obtained is updated and the search continues with the best known alternative.

We also competed in the MISTA Challenge 2013 (Toffolo et al., 2013), ranking third with an algorithm that exhibits certain similarities with the one discussed in this chapter. Despite such similarities, the algorithm which competed is far less general and encountered difficulties in handling large problem instances.

A lower bound on a project's earliest finish time is the *critical path* duration. The *Critical Path Method* (Kelley and Walker, 1959) is a tool for general project management that represents the precedence constraints as a network, where each job is a node and arcs connect jobs to their predecessors and successors. This method computes the earliest and latest start and finish times for each job such that the project is not delayed, while respecting precedence constraints. The critical path itself is the sequence of related jobs that cannot be delayed without delaying the entire project, denoted by a path between the two dummy jobs in the network. The critical path duration is the sum of these job durations. In order to calculate the critical path duration for a GPSP instance, some constraints are excluded: the duration of a job is fixed as the duration of the fastest execution mode and all resource constraints are ignored. Once a project is scheduled it is assigned both a *makespan*, defined as the difference between the project's finish and release times, and a *project delay*, defined as the difference between the project's critical path duration and the actual project duration.

The GPSP minimizes two objectives: *Total Project Delay* (TPD) and *Total MakeSpan* (TMS). The TPD is defined as the sum of all project delays while the TMS is defined as the time required to finish all projects, given by the difference between the maximum finish time and the minimum release time among all projects. The TPD is the primary objective, while the TMS is a tie-breaker.

Next, an IP formulation for the GPSP is presented and evaluated.

4.2 Integer programming formulation

This section presents an IP formulation for the GPSP which is a straightforward adaption of the formulations presented by Kolisch and Sprecher (1997) and Pritsker et al. (1969). These authors proposed time-indexed formulations for the RCPSP in which the number of variables increases with the duration of jobs and projects. Koné et al. (2011) proposed an event-based formulation that does not depend on the duration of jobs and projects. The number of events is given by the total number of jobs. Their formulation is thus particularly interesting for instances with long-duration jobs, in which the number of jobs is usually

smaller than the number of time slots. Koné et al. (2011) presented situations in which an event-based formulation outperforms time-indexed formulations.

Both event-based and time-indexed formulations for the GPSP were considered. However, the event-based formulation performed very poorly when compared against the time-indexed formulation, with state-of-the-art solvers unable to solve some of the easier instances when employing it. Koné et al. (2013) conducted a detailed comparison of formulations for the RCPSP which corroborates our finding. They concluded that event-based formulations result in poor linear relaxation bounds, being only adequate for problems with high level of parallelism involving large and very different job durations. Therefore, we focus upon time-indexed formulations.

Before presenting an IP for the GPSP, the necessary input data is introduced:

P : set of projects;

J : set of all jobs;

J_p : set of jobs associated with project p, such that $J_p \subseteq J$;

M_j : set of execution modes for job j;

K : set of non-renewable resources;

R : set of renewable resources;

T : set of available time-slots $\{1, ..., |T|\}$;

$T_{j,m}$: set of possible time-slots for beginning job j in mode m, $\{te_j, ..., tl_{j,m}\}$, where te_j is the earliest start time of job j given by the critical path and $tl_{j,m}$ is the latest start time of job j in mode m such that the job ends within $|T|$ time-slots;

B : set of direct precedence relations expressed by ordered pairs (j, l), where j should be scheduled before l;

B_j : set of jobs with which job j has a direct precedence relation, i.e. jobs l such that either $(l, j) \in B$ or $(j, l) \in B$;

$d_{j,m}$: duration of job j in mode m;

$u_{k,j,m}$: required units of non-renewable resource k for job j to be processed in mode m;

$v_{r,j,m}$: required units of renewable resource r for job j to be processed in mode m;

o_k : available units of non-renewable resource k;

q_r : available units of renewable resource r;

ω_1 : weight associated with objective TPD;

ω_2 : weight associated with objective TMS.

The following variables are employed by the formulation:

$x_{j,m,t}$: binary variable that is equal to 1 when the job j is processed in mode m and starts at time t and 0 otherwise.

z : completion time for the final project.

The objective function minimizes the sum of completion times for projects, given by the sum of start times of final *dummy* jobs j_p^+ $(p \in P)$, together with the completion time of the last project[2].

Minimize:

$$\omega_1 \sum_{p \in P} \sum_{t \in T_{j_p^+,0}} t \, x_{j_p^+,0,t} + \omega_2 z \tag{4.1}$$

Subject to:

$$\sum_{m \in M_j} \sum_{t \in T_{j,m}} x_{j,m,t} = 1 \qquad \forall j \in J \tag{4.2}$$

$$\sum_{m \in M_j} \sum_{t \in T_{j,m}} (t + d_{j,m}) \, x_{j,m,t} \leq \sum_{m \in M_l} \sum_{t \in T_{l,m}} t \, x_{l,m,t} \quad \forall (j,l) \in B \tag{4.3}$$

$$\sum_{j \in J} \sum_{m \in M_j} \sum_{t'=t-d_{j,m}+1}^{t} v_{j,m,r} x_{j,m,t'} \leq p_r \qquad \forall r \in R, t \in T \tag{4.4}$$

$$\sum_{j \in J} \sum_{m \in M_j} \sum_{t \in T_{j,m}} u_{j,m,k} x_{j,m,t} \leq o_k \qquad \forall k \in K \tag{4.5}$$

[2]Constants included in objective function of the MISTA Challenge 2013 Problem were omitted for the sake of clarity.

$$\sum_{t \in T_{j_p^+,0}} t\, x_{j_p^+,0,t} \leq z \qquad \forall p \in P \tag{4.6}$$

$$x_{j,m,t} \in \{0,1\}, \;\; t_j \in \mathbb{Z}^+ \qquad \forall j \in J, m \in M_j, \; t \in T_{j,m} \tag{4.7}$$

Constraints (4.2) ensure every job is allocated to exactly one starting time and mode. Meanwhile, Constraints (4.3) force precedence relations to be satisfied. Constraints (4.4) and (4.5) control the usage of renewable and non-renewable resources, respectively. Constraints (4.6) compute the final completion time of all projects. Note that the final dummy jobs (j_p^+) are utilized within these constraints. Finally, Constraints (4.7) define the bounds of the employed variables.

4.2.1 Computational experiments

Formulation (4.1)-(4.7) was implemented in two state-of-the-art IP solvers: CPLEX 12.7 and Gurobi 7.5. Experiments were conducted considering the dataset from the MISTA Challenge 2013 and executed on an Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz computer with 128Gb of RAM memory running Linux Ubuntu 16.04.2 LTS. Both CPLEX and Gurobi were executed in sequential mode (single threaded). The formulation was preprocessed before generation to reduce the number of variables by tightening the size of T_{jm} for every job $j \in J$ and its execution modes $m \in M_j$. Moreover, while the number of time-slots $|T|$ should also be as tight as possible, it must be large enough to enable the production of feasible schedules. With this in mind, $|T|$ was set in accordance with the best known solutions reported by Wauters et al. (2016). The principle is to ensure there exists no improving solution using more than $|T|$ time-slots. Equation 4.8 describes how $|T|$ is calculated, where CP_p represents the critical path completion time for project p, TPD the solution's total project delay and LB_p the optimal project delay (or a lower bound) for project p if it is scheduled alone with all resources available. Note that the MISTA Challenge 2013 instances employ projects from the PSPLib[3], and therefore some lower bounds are available.

$$|T| = \max_{p \in P} \left(\mathrm{CP}_p + \mathrm{TPD} - \sum_{p' \in P : p' \neq p} \mathrm{LB}_{p'} \right) \tag{4.8}$$

[3]http://www.om-db.wi.tum.de/psplib/library.html

Table 4.1 details the number of projects $|P|$ and total number of jobs $|J|$ for each instance in addition to the generated model's dimensions. The number of variables, constraints and non-zero coefficients are also displayed. The table presents the results obtained by both CPLEX 12.7 and Gurobi 7.5. Column 'Gap' shows the best gap obtained considering the runtime limit of eight hours, while columns 'T_{relax}' and 'T_{total}' present the runtime for solving the root linear relaxation and the total runtime, respectively. For compactness, runtimes are presented in different time units: seconds (s), minutes (m) and hours (h).

Note from Table 4.1 that both IP solvers were capable of quickly finding optimal solutions for the first three instances. The fourth instance was solved by Gurobi after 7.5h. For the remaining instances, the solvers either obtained poor results (high gap values) or did not find a feasible solution at all. For instances B-9 and B-10 Gurobi was incapable of solving the linear relaxation within eight hours. This confirms the difficulty of the problem.

Table 4.1: Experiments with IP formulation

Inst.	Problem		Model Dimensions			CPLEX 12.7			Gurobi 7.5		
	$\|P\|$	$\|J\|$	Variables	Constrs	Non-zeros	Gap	T_{relax}	T_{total}	Gap	T_{relax}	T_{total}
A-1	2	24	687	183	8,716	0.0%	0.0s	0.1s	0.0%	0.0s	0.1s
A-2	2	44	7,752	378	121,675	0.0%	0.1s	14.7s	0.0%	0.0s	2.8s
A-3	2	64	12,110	477	207,167	0.0%	0.1s	3.0s	0.0%	0.0s	1.0s
A-4	5	60	9,507	950	136,350	10.4%	0.2s	8.0h	0.0%	0.1s	7.5h
A-5	5	110	43,569	1,285	655,140	-	8.2s	8.0h	-	4.5s	8.0h
A-6	5	160	149,491	2,545	2,019,375	-	44.0s	8.0h	-	32.4s	8.0h
A-7	10	120	165,291	1,416	2,287,198	-	3.4m	8.0h	76.3%	2.0m	8.0h
A-8	10	220	201,465	1,396	2,759,682	-	2.7m	8.0h	-	1.5m	8.0h
A-9	10	320	306,071	4,886	4,239,473	-	2.3m	8.0h	-	1.7m	8.0h
A-10	10	320	750,061	10,217	11,626,885	-	24.9m	8.0h	-	57.3m	8.0h
B-1	10	120	114,773	4,412	1,581,784	27.2%	17.9s	8.0h	49.9%	16.2s	8.0h
B-2	10	220	223,630	1,436	2,984,698	-	5.6m	8.0h	-	3.6m	8.0h
B-3	10	320	419,744	6,255	6,867,559	-	10.5m	8.0h	-	11.6m	8.0h
B-4	15	180	397,623	13,982	5,790,186	-	4.5m	8.0h	71.6%	4.7m	8.0h
B-5	15	330	587,780	11,414	8,473,331	-	9.9m	8.0h	-	38.1m	8.0h
B-6	15	480	737,034	10,387	11,663,620	-	29.6m	8.0h	-	18.5m	8.0h
B-7	20	240	398,163	14,034	4,984,221	-	3.0m	8.0h	70.6%	1.9m	8.0h
B-8	20	440	1,486,984	3,858	19,166,272	-	8.0h	8.0h	-	2.4h	8.0h
B-9	20	640	4,405,295	52,707	65,528,163	-	7.9h	8.0h	-	-	8.0h
B-10	20	460	1,345,014	3,450	19,623,133	-	2.4h	8.0h	-	-	8.0h
X-1	10	120	126,233	1,168	1,584,486	52.3%	1.7m	8.0h	66.6%	22.4s	8.0h
X-2	10	220	310,399	6,392	4,459,465	-	3.3m	8.0h	-	1.6m	8.0h
X-3	10	320	334,415	5,261	5,429,664	-	6.1m	8.0h	-	5.8m	8.0h
X-4	15	180	254,992	1,629	3,495,502	-	3.1m	8.0h	73.0%	2.0m	8.0h
X-5	15	330	982,377	18,216	14,598,595	-	30.5m	8.0h	-	2.9h	8.0h
X-6	15	480	938,782	12,715	12,942,495	-	14.2m	8.0h	-	28.0m	8.0h
X-7	20	240	435,077	15,243	5,497,018	-	2.5m	8.0h	71.3%	1.7m	8.0h
X-8	20	440	1,150,713	21,249	16,352,555	-	18.2m	8.0h	-	58.8m	8.0h
X-9	20	640	3,539,176	42,814	49,364,174	-	3.8h	8.0h	-	4.6h	8.0h
X-10	20	450	1,819,489	31,863	24,207,867	-	2.2h	8.0h	-	43.3m	8.0h

4.3 Decomposition-based heuristic

The poor performance of Formulation (4.1)-(4.7) coupled with the well-known difficulty of the GPSP justifies addressing the problem with heuristic methods. In this section a decomposition-based heuristic algorithm is proposed, consisting of constructive and local search phases.

The major obstacles for building feasible GPSP solutions are the non-renewable resources. While the use of renewable resources only impacts a project's duration, the use of non-renewable resources may produce infeasible solutions. In the proposed algorithm, job execution modes are defined beforehand so that they respect the limits of non-renewable resources. Once a set of feasible modes is obtained, a decomposition-based heuristic utilizes it to iteratively build an initial feasible solution. This initial solution is subsequently refined by a local search, which employs the same decomposition. This local search consists of a Forward-Backward Improvement (FBI) (Li and Willis, 1992) method hybridized with an IP model to modify the schedule. These methods are integrated within an Iterated Local Search (ILS) (Lourenço et al., 2010) metaheuristic framework employing a biased rebuild procedure to perturb solutions. Figure 4.1 presents an outline of the developed algorithm. More details are presented in the following sections.

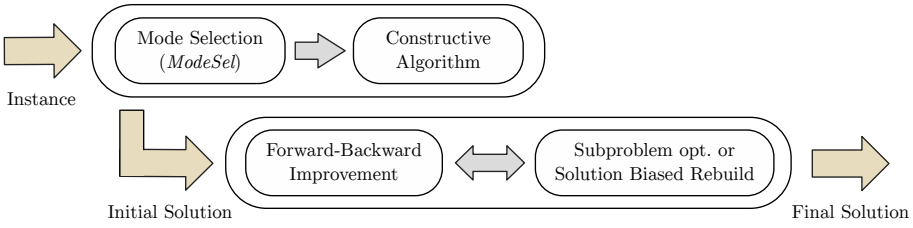


Figure 4.1: Outline of the developed algorithm

4.3.1 Constructive algorithm

The generation of an initial solution is conducted in two steps, as outlined in Figure 4.1. First, a feasible set of execution modes is obtained. Then a subproblem optimization procedure iteratively creates a feasible solution using these execution modes as guidance.

Obtaining a feasible set of execution modes

In contrast to Asta et al. (2016) and Geiger (2013), an exact algorithm is employed to obtain a feasible set of execution modes considering non-renewable resources. This approach resembles that of Coelho et al. (2011), with the difference being they model the problem as a Boolean Satisfiability Problem (SAT) and solve it using the DPLL algorithm from Davis and Loveland (1962). One drawback of the SAT approach is that an exponential number of clauses is required to model non-renewable resources constraints and the authors encountered difficulties in explicitly handling them.

The problem of selecting the initial modes is denoted henceforth as *ModeSel*. The *ModeSel* is solved by an IP model which defines, for each job, a mode such that total non-renewable resource consumption limits are always respected. The model itself is as hard as the m-dimensional knapsack problem and essentially consists of the original model without the renewable resources constraints.

To present the *ModeSel* formulation, the same nomenclature (input data) from Section 4.2 is considered. The following decision variables are employed:

$x_{j,m}$: binary variable that is equal to 1 when job j is executed in mode m and 0 otherwise;

t_j : integer variable indicating the start time of job j;

The objective function of the *ModeSel* formulation, presented by Equation (4.9), minimizes two parcels. For each parcel a weight is assigned, indicating its priority. These weights are defined hierarchically such that $\omega_1 \gg \omega_2$. The first parcel minimizes the sum of project completion times by considering the start time of each project's last (*dummy*) job. The second parcel is responsible for minimizing job durations. Note that a heuristic strategy is employed, whose principle is to prioritize jobs with a higher number of direct precedence constraints, $|B_j|$, since these jobs may potentially become bottlenecks within the schedule.

Minimize:

$$\omega_1 \sum_{p \in P} t_{j_p^+} \; + \; \omega_2 \sum_{j \in J} \sum_{m \in M_j} |B_j| d_{j,m} x_{j,m} \tag{4.9}$$

Subject to:

$$\sum_{m \in M_j} x_{j,m} = 1 \qquad\qquad \forall j \in J \tag{4.10}$$

$$\sum_{j \in J} \sum_{m \in M_j} u_{k,j,m} x_{j,m} \leq o_k \quad \forall k \in K \tag{4.11}$$

$$t_j + \sum_{m \in M_j} d_{j,m} x_{j,m} \leq t_l \quad \forall (j,l) \in B \tag{4.12}$$

$$t_j \geq te_j \quad \forall j \in J \tag{4.13}$$

$$x_{j,m} \in \{0,1\},\ t_j \in \mathbb{Z}^+ \quad \forall j \in J, m \in M_j \tag{4.14}$$

Constraints (4.10) guarantee that only one mode is selected for each job while Constraints (4.11) ensure that non-renewable resource capacities are respected. The start time of a job is limited by Constraints (4.12) and (4.13). Constraints (4.12) states that a job can only begin after all its predecessors finishes while Constraints (4.13) prevents jobs from beginning before their release time by considering the earliest start times from the critical path. Finally, Constraints (4.14) define variables $x_{j,m}$ as binary and variables t_j as integer.

Building an initial solution

As stated before, the use of non-renewable resources may lead to infeasible solutions. The feasible execution mode set provided by the *ModeSel* solution is employed within the constructive schedule algorithm to circumvent this issue. Since the use of renewable resources only impacts project durations, and these durations are not bounded, the constructive procedure always results in a feasible solution.

The constructive algorithm employs an IP decomposition-based heuristic to construct an initial feasible solution. First the projects are grouped according to their estimated completion times. These estimations are obtained by a simple greedy constructive algorithm, which considers a topological ordering of the jobs and allocates them according to their renewable resource consumption. Following the obtained estimations, projects are divided into sets, such that each project set is scheduled sequentially by the constructive algorithm. The principle is to first schedule projects whose completion times tend to be earlier, since finishing these projects as soon as possible potentially contributes to minimizing the TPD. Moreover, subproblems which consider only a subset of the projects are generally easier to solve, given the smaller number of jobs to

schedule.

The problem is decomposed into sequential and non-overlapping time windows for each project set. At each iteration, a time window is solved by an IP model to allocate jobs within this window. Once a project set is allocated, the procedure restarts with the next set. The algorithm finishes when all jobs are allocated. Original constraints such as precedence and resource consumption are considered by this model. Beyond the input data presented in Section 4.2, five additional parameters are required:

J^* : subset of jobs $j \in J$ that must be scheduled within the time window; this is the case for jobs belonging to projects scheduled in a previous algorithm iteration;

$\tilde{m}_j$: execution mode established by *ModeSel* for job j;

$\delta_{j,m,t}$: estimated gain in completion time for job j achievable by scheduling it using mode m at timeslot t. Jobs $j \in J^*$ have $\delta_{j,m,t}$ set to zero, while other jobs j have $\delta_{j,m,t} = max(1, (t_0 + \eta + d_{j,\tilde{m}_j}) - (t + d_{j,m}))$, signifying the difference of (i) the sum of the first timeslot after the current time window and the job's duration in mode $\tilde{m}_j$ and (ii) the job's completion time in mode m if it starts at time t. The principle here is that the sooner job j finishes, the higher the value for $\delta_{j,m,t}$.

φ_j : estimated finish time for job j's project if job j is not scheduled. This estimation is given by the end of the current time window summed with the time between the start of j and the dummy job representing the end of its project. The critical path with execution modes established by *ModeSel* is utilized. If some $t + d_{j,m}$ (t within the subproblem's time window) is larger than φ_j, then φ_j is updated such that it surpasses all possible finishing times for jobs allocated within the time window.

Note that estimations δ and φ are necessary as there is no precise information concerning the objective function value when constructing the solution. The completion time of a project is only known after all its jobs are scheduled.

The following decision variables are defined:

$x_{j,m,t}$: binary variable that is equal to 1 if job j is processed in mode m and starts at time t and 0 otherwise;

y_p : integer variable indicating the estimated finish time for project p – in general, the greater the number of allocated jobs of project p, the lower the value of y_p;

z : integer variable indicating the latest estimated finished time of projects.

The objective function, given by Equation (4.15), consists of three parcels. As before, hierarchical weights are assigned to the parcels, such that $\omega_1 \gg \omega_2 \gg \omega_3$. The first parcel minimizes a heuristic estimation of each project's completion time. This estimation considers jobs which cannot be allocated in the current time window. The second parcel is responsible for maximizing the number of allocated jobs, taking into consideration the priorities established by δ. Finally, the last parcel aims at minimizing the overall completion time, also based on the estimations previously defined.

Formulation (4.15)-(4.23) permits scheduling multiple projects at once. Experiments revealed, however, that optimizing one project at a time during the constructive phase resulted in higher-quality initial solutions on average. Therefore, each project set contains one project. Note, however, that a general version of the formulation is presented.

Minimize:

$$\omega_1 \sum_{p \in P} y_p \; - \; \omega_2 \sum_{j \in J} \sum_{m \in M_j} \sum_{t \in T_{j,m}} \delta_{j,m,t} x_{j,m,t} \; + \; \omega_3 z \tag{4.15}$$

Subject to:

$$\sum_{m \in M_j} \sum_{t \in T_{j,m}} x_{j,m,t} \leq 1 \qquad \forall j \in J \backslash J^* \tag{4.16}$$

$$\sum_{m \in M_j} \sum_{t \in T_{j,m}} x_{j,m,t} = 1 \qquad \forall j \in J^* \tag{4.17}$$

$$\sum_{j \in J} \sum_{m \in M_j} \sum_{t'=t-d_{j,m}+1}^{t} v_{r,j,m} x_{j,m,t'} \leq q_r \qquad \forall r \in R, t \in T \tag{4.18}$$

$$\sum_{j \in J} \sum_{m \in M_j} \sum_{t \in T_{j,m}} u_{k,j,m} x_{j,m,t} + \sum_{j \in J} u_{k,j,\tilde{m}_j} \left(1 - \sum_{m \in M_j} \sum_{t \in T_{j,m}} x_{j,m,t} \right) \leq o_k \qquad \forall k \in K \tag{4.19}$$

$$\sum_{m \in M_j} \sum_{t \in T_{j,m}} (t + d_{j,m})\, x_{j,m,t} \leq \sum_{m \in M_l} \sum_{t \in T_{l,m}} t x_{l,m,t}$$

$$+ \left(1 - \sum_{m \in M_l} \sum_{t \in T_{l,m}} x_{l,m,t}\right) M \qquad \forall (j,l) \in B \tag{4.20}$$

$$\sum_{m \in M_j} \sum_{t \in T_{j,m}} x_{j,m,t} \;\geq\; \sum_{m \in M_l} \sum_{t \in T_{l,m}} x_{l,m,t} \qquad \forall (j,l) \in B \tag{4.21}$$

$$\varphi_j + \sum_{m \in M_j} \sum_{t \in T_{j,m}} (t + d_{j,m} - \varphi_j)\; x_{j,m,t} \leq y_p \qquad \forall p \in P, j \in J_p \tag{4.22}$$

$$z \geq y_p \qquad \forall p \in P \tag{4.23}$$

$$x_{j,m,t} \in \{0,1\} \qquad \forall j \in J, m \in M_j, t \in T_{j,m} \tag{4.24}$$

$$y_p \in \mathbb{Z}^+,\; z \in \mathbb{Z}^+ \qquad \forall p \in P \tag{4.25}$$

Constraints (4.16) ensure each job is allocated at most once. Constraints (4.17) ensure that jobs which must be scheduled are allocated within the time window. Note that sets $T_{j,m}$ are preprocessed to ensure precedence constraints are not violated for jobs in J^*. Constraints (4.18) and (4.19) ascertain that the available amounts of renewable and non-renewable resources are not exceeded, respectively. Values for q_r o_k are set according to the current solution and *ModeSel* to guarantee a feasible solution is obtained. Constraints (4.20) and (4.21) guarantee the precedence relations between the jobs. Two constraints are required, since it is possible that a job may remain unallocated in a solution. Therefore, precedence constraints should only hold for allocated jobs. Constraints (4.22) set a project's finish time to be larger than or equal to the estimation previously described whenever jobs related to the project remain unallocated. Note that estimation φ is not utilized when all jobs are allocated. Constraints (4.23) compute the final completion time of all projects based on the aforementioned estimations. Finally, Constraints (4.24) and (4.25) define variables x as binary and variables y and z as integer.

The pseudo-code for generating an initial solution is presented in Algorithm 4.1. The algorithm takes as input data: (*i*) the set J of all problem's jobs, (*ii*) an ordered list $\tilde{P}$ of project sets, (*iii*) the set of feasible execution modes $\tilde{M}$ and (*iv*) the window size, given by η.

First the solution is initialized (line 1). The algorithm then proceeds by building a solution for each project set $P' \in \tilde{P}$ (line 2). The beginning of the first

Algorithm 4.1: Decomposition-based constructive algorithm

Input: Set of jobs J, ordered list of project sets $\tilde{P}$, *ModeSel* solution $\tilde{M}$, and subproblem size η

Constructive(J, $\tilde{P}$, $\tilde{M}$, η)

1 $S \leftarrow$ empty solution
2 **for** each set of projects $P' \in \tilde{P}$ **do**
3 $t_0 \leftarrow$ minimum release time for projects in P'
4 **while** S does not contain all jobs from P' **do**
5 $J' \leftarrow$ unallocated jobs in P' feasible for a time in $\{t_0, ..., t_0 + \eta - 1\}$
6 $J^* \leftarrow$ jobs allocated in S with start time within $\{t_0, ..., t_0 + \eta - 1\}$
7 $R' \leftarrow$ renewable resource availability considering S
8 $K' \leftarrow$ non-renewable resource availability considering S and $\tilde{M}$
9 $S' \leftarrow$ solution for subproblem defined by t_0, η, J', J^*, R' and K'
10 $S \leftarrow S \cup S'$ // update solution
11 $t_0 \leftarrow t_0 + \eta$ // move to the next time window
12 **return** S

time window is selected (line 3) and the algorithm iterates until all jobs from projects in P' are allocated (line 4). At each iteration the algorithm selects the unallocated jobs within P' which have the earliest start time in $\{t_0, ..., t_0+\eta-1\}$ (line 5). Jobs already scheduled in S within this time window are also considered (line 6). After selecting these jobs, the amount of renewable resources available at each time-slot is calculated (line 7). A minimum amount of non-renewable resources must be reserved for jobs which are not included within the subproblem. Therefore, non-renewable resource limits are defined so as to guarantee that a feasible solution will be generated (line 8). *ModeSel* solution $\tilde{M}$ is employed to calculate such limits. Next, the subproblem is created and solved (line 9) and its solution is included in solution S (line 10). The time window is subsequently advanced (line 11). Finally, once a feasible solution is generated, it is returned (line 12).

It is important to note that a time limit is imposed upon the solver since it is necessary to guarantee that a feasible solution will be generated. To speed up the solution process, a greedy feasible solution for each subproblem is created and passed to the IP solver before optimization. The greedy algorithm considers a topological ordering of the jobs and allocates them according to their renewable resource consumption. Jobs are allocated to be processed with the modes defined by the *ModeSel* solution.

Another critical aspect of the algorithm concerns the time window size. As also

observed in Chapters 2 and 3, defining the value for η is not straightforward. Time windows must be small enough to ensure that subproblems are easily solved and, at the same time, large enough to indicate relevant allocations.

4.3.2 Local Search algorithm

The local search employs a similar decomposition to that employed by the constructive algorithm. A time window is defined and only jobs currently allocated within this window are considered eligible to have their modes and starting times modified. However, modifications in the beginning or middle of the schedule cannot improve solution quality in most cases. Note that solution quality, given by project completion times, can only be altered if the timeslot of a project's last job is modified. This is clearly not the case for most iterations of the local search. To circumvent this issue, the local search is combined with the *Forward-Backward Improvement* (FBI) procedure, which is described next.

Forward-Backward Improvement Procedure

The FBI is an improvement method proposed by Li and Willis (1992) which is commonly applied to scheduling problems, consisting of two steps: *forward* and *backward*. In the *forward* step, jobs are right-justified in the schedule, meaning that except for the first and last jobs, all job allocations are shifted to the right, beginning from the final job's immediate predecessors until the initial job's immediate successors. This step generates a schedule wherein no job may finish later without advancing some other job or increasing the makespan. Since the final job is not shifted, the current makespan is maintained. If a slack (window with no jobs being executed) in the schedule is generated, the *backward* step attempts to reduce the makespan by eliminating such slack. In this step, jobs are left-justified in the schedule. Therefore, except for the initial job, all job allocations are shifted to the left, beginning from the initial job's immediate successors until the final job. This step generates a schedule wherein no activity can be started earlier without violating constraints.

Figure 4.2 illustrates an example of this process. In this figure, the *dummy* jobs (first and last ones) are marked with a "D". The first graph represents the initial solution and the following two graphs depict the schedules generated after the *forward* and *backward* steps, respectively.

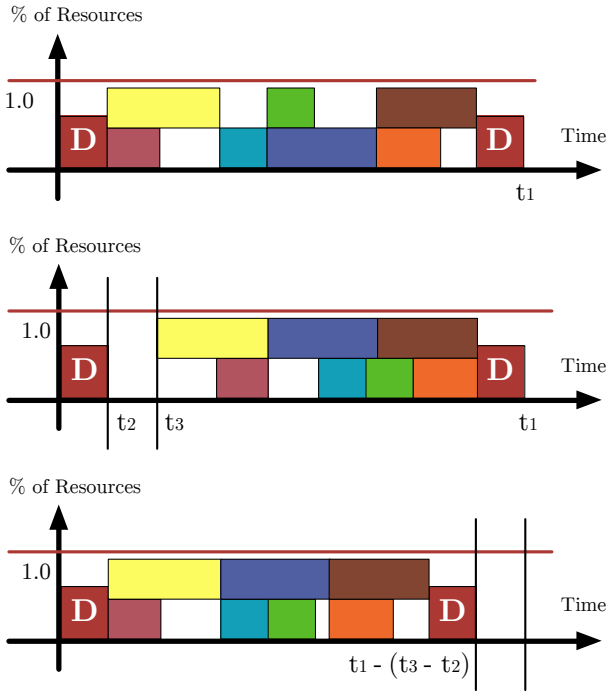


Figure 4.2: Forward-Backward Improvement (FBI) example

Note that the schedule remains feasible throughout the entire process, since precedence relations and resource consumption constraints are satisfied. Each shift is performed employing a *Serial Generation Scheme* (SGS).

After the FBI is applied, some jobs may have not been shifted in any direction and may therefore constitute the problem's bottleneck. A third improvement step is proposed, which marks these stationary jobs and then randomly changes each marked job's mode. Next, a left-shift is applied using the SGS.

These three steps are performed repeatedly while there is an improvement in solution quality and until the time limit has not been reached.

Subproblem optimization scheme

In essence, the subproblem optimization modifies the solution for the FBI procedure to improve it. Given that most time windows do not include any project's last job, the FBI procedure is employed to propagate modifications which potentially do not include the schedule's end.

To modify the solution, two hierarchical objective functions are considered. The first minimizes the sum of completion times for each project's last job within the time window. The second objective aims at altering the solution by employing a heuristic multiplier $w_{j,m,t}$, which defines the priority for allocating job j in mode m at timeslot t. The principle is to assign high priority – low values for $w_{j,m,t}$ – to different allocations. Therefore, current allocations within the solution have high $w_{j,m,t}$ values. Other allocations have their priority multiplied by a random factor, within the range $[0, 1[$. In general, this approach results in different solutions without deteriorating the objective value. In practice, this helps the FBI procedure to avoid many local optima, thereby enabling it to continue improving the solution.

Note that, in contrast to our approach which competed in the challenge (Toffolo et al., 2013), the objective function does not focus solely on modifying job execution modes. In fact, focusing on execution modes would render the subproblem optimization local search inappropriate to address problem instances with single execution modes.

The subproblems are solved by an IP model. A significant amount of preprocessing is applied to make the model as compact as possible. Moreover, sets J, J_p, T and $T_{j,m}$ are restricted to consider only the subproblem's time window. Two decision variable sets are considered:

$x_{j,m,t}$: binary variable that is equal to 1 when job j is processed in mode m and starts at timeslot t, and 0 otherwise;

y_p : auxiliary variable which indicates the completion time of the last job of project p within the considered time window.

The objective function, given by Equation (4.26), is responsible for changing the modes and start times of the jobs whenever possible. Again, hierarchical weights ω_1 and ω_2 are employed, with $\omega_1 \gg \omega_2$.

Minimize:

$$\omega_1 \sum_{p \in P} y_p + \omega_2 \sum_{j \in J} \sum_{m \in M_j} \sum_{t \in T_{j,m}} w_{j,m,t} x_{j,m,t} \tag{4.26}$$

Subject to:

$$\sum_{m \in M_j} \sum_{t \in T_{j,m}} x_{j,m,t} = 1 \qquad \forall j \in J \tag{4.27}$$

$$\sum_{j \in J} \sum_{m \in M_j} \sum_{t \in T_{j,m}} u_{k,j,m} x_{j,m,t} \leq o_k \qquad \forall k \in K \tag{4.28}$$

$$\sum_{j \in J} \sum_{m \in M_j} \sum_{t'=t-d_{j,m}+1}^{t} v_{r,j,m} x_{j,m,t'} \leq q_r \qquad \forall r \in R, t \in T \tag{4.29}$$

$$\sum_{m \in M_j} \sum_{t \in T_{j,m}} (t + d_{j,m}) x_{j,m,t} \leq \sum_{m \in M_l} \sum_{t \in T_{l,m}} t x_{l,m,t} \quad \forall (j,l) \in B \tag{4.30}$$

$$\sum_{m \in M_j} \sum_{t \in T_{j,m}} (t + d_{j,m}) x_{j,m,t} \leq y_p \qquad \forall p \in P, j \in J_p \tag{4.31}$$

$$x_{j,m,t} \in \{0,1\} \qquad \forall j \in J, m \in M_j, t \in T_{j,m} \tag{4.32}$$

$$y_p \in \mathbb{Z}^+ \qquad \forall p \in P \tag{4.33}$$

Constraints (4.27) are responsible for ensuring that each activity is allocated exactly once. Constraints (4.28) and (4.29) guarantee the limits for non-renewable and renewable resources are respected, respectively. Constraints (4.30) ensure precedence relations are enforced. Constraints (4.31) set variables y_p to indicate the last completion time of project p's jobs within the time window. Finally, Constraints (4.32) declare variables x binary while Constraints (4.33) set variables y as integer.

4.3.3 Metaheuristic framework integration

In contrast to the TUP and NRP (Chapters 2 and 3), the GPSP imposes an additional challenge for time-window based subproblem optimization. Note that jobs are spread across all available timeslots. By optimizing time-windows

of size η individually, it is unlikely that jobs scheduled in a late timeslot will be reallocated to much earlier timeslots during the search, unless η is large enough. Given the computational hardness of the GPSP, η tends to be set to a small value, meaning that significant modifications in the projects' schedules are improbable. To handle this situation and enable large alterations in the schedule, the local search algorithm is hybridized within an Iterated Local Search (ILS) framework.

The ILS metaheuristic was introduced by Lourenço et al. (2010) and relies upon perturbations to escape from local optima. Note that the subproblem optimization scheme presented in Section 4.3.2 may also be interpreted as a special local perturbation: one that never worsens the solution. It is not the case of the proposed biased solution rebuild, whose objective is to make large alterations in the schedule. These alterations (or perturbation) may considerably worsen the solution, which may require re-optimization afterwards.

Biased solution rebuild

A biased rebuild solution method that perturbs a guidance (initial) solution method is proposed and works as follows. First, jobs are put in a set sorted according to their start times in the guidance solution and a new solution, S', is created. Then, the jobs are added to S' one at a time into the smallest start time that does not violate precedence or renewable resources constraints. Non-renewable resources constraints are always satisfied because the jobs' execution modes in the guidance solution are re-utilized in the allocations.

Each job's selection probability is given by a Heuristic-Biased Stochastic Sampling (HBSS) (Bresina and Bresina, 1996). Any job may be selected, but the first jobs have far greater probabilities of being selected. The chances of selecting a job is given by $f(r) = e^{-r}$, where r is the position of the job in the sorted set. After being selected, the job is added to S' only if it does not violate any precedence constraints. Once added to the solution, the job leaves the sorted set. The method returns after all jobs are added to S'.

ILS algorithm

The ILS algorithm's pseudo-code is presented in Algorithm 4.2. The algorithm takes as input data: (i) an initial solution; (ii) the maximum number of

iterations; (*iii*) the maximum number of runs of the IP model per iteration; (*iv*) the time window minimum size, η_{min}; and (*v*) the time window maximum size η_{max}.

FBI first is applied to the initial solution, which is copied to S^* and S (line 1), and counters p and *iter* are initialized (line 2). Next, the solution is changed by the IP model described in Section 4.3.2. The algorithm iterates until the time limit is reached (line 3). At each iteration, counter *iter* is incremented (line 4) and the IP model is solved ρ times considering solution S and random time windows, returning a modified solution (lines 5-8). Afterwards, the FBI procedure is executed again (line 9) to improve solution S. If a new best solution is obtained, it is stored and counters ρ and *iter* are reset (lines 10-12). Eventually, *iter* may reach the value $iter_{max}$ causing ρ to be incremented (lines 13-14). After a certain number of iterations without improvement, the solution is perturbed using the biased rebuild method (lines 15-17). In this process, the best solution produced so far is employed to guide the generation of a new solution. The algorithm output is a possibly improved solution (line 18).

Algorithm 4.2: Decomposition-based local search algorithm for the GPSP

Input: Initial solution S_0, maximum number of iterations without improvements $iter_{max}$, maximum number of IP subproblems solved ρ_{max}, and bounds on the time window size, η_{min} and η_{max}

ILS(S_0, $iter_{max}$, ρ_{max}, η_{min}, η_{max})

```
1  S* ← S ← FBI(S_0)
2  ρ ← iter ← 1
3  while time limit is not reached do
4      iter ← iter + 1
5      for i ← 0 to ρ do
6          t_0 ← random time-slot utilized in solution S
7          η ← random time window size in {η_min, ..., η_max}
8          update S by solving subproblem with time-slots {t_0, ..., t_0 + η − 1}
9      S ← FBI(S) // FBI is applied to propagate modifications in solution S
10     if S is better than S* then
11         S* ← S
12         ρ ← iter ← 1
13     if iter > iter_max then
14         ρ ← ρ + 1 // increasing number of subproblems per iteration
15     if ρ > ρ_max then
16         ρ ← 1 // ρ is reset when ρ_max is exceeded
17         S ← new solution employing the biased rebuild method over S*
18 return S*
```

4.4 Computational experiments

All algorithms were coded in C++ and the IP models were solved by Gurobi 7.5, which performed better than CPLEX when solving subproblems. The experiments were executed on Intel(R) Xeon E5-2680v3 CPU @ 2.5GHz computers with 64GB of RAM memory running Red Hat Enterprise Linux ComputeNode 6.5. The decomposition-based heuristic was run in parallel on 4 threads, each with different parameters values. Although the implemented algorithm is clearly sequential, the MISTA Challenge 2013 permitted the use of up to 4 threads. The values for the parameters, presented in Table 4.2, were obtained after several empirical tests. The size of the time windows η is the most critical parameter. After several runs, we observed that subproblems with up to 50 time-slots could be solved by both CPLEX and Gurobi within the runtime limit. Larger time windows, however, generated subproblems which required long execution times to be solved. We also took some precautions and included a thread with a smaller time window size (fourth thread), to ensure the solver returns a solution within the runtime limit. The number of projects in each project set of $\tilde{P}$, given by $\tilde{P}_{\text{size}}$, was set to one. This enabled quickly solving large subproblems during the constructive phase, which ultimately resulted in improved initial solutions. The final algorithm is not very sensitive to the other parameters.

Table 4.2: Parameters employed during experiments

Thread	Constructive		Local Search			
	η	$\tilde{P}_{\text{size}}$	$iter_{max}$	ρ_{max}	η_{min}	η_{max}
1	50	1	10	15	10	50
2	40	1	20	10	10	40
3	30	1	40	20	10	30
4	20	1	40	20	10	20

The algorithms were evaluated for two different project scheduling problems: (*i*) the *Multi-Project Scheduling Problem* (MPSP), considering instances from the MPSPLib[4] and (*ii*) the GPSP, considering instances from the MISTA Challenge 2013[5]. The difference between these two problems lies in the presence of multiple execution modes for jobs. In contrast to the GPSP, there is only one execution mode for jobs in the MPSP. This section first presents and discusses results obtained for the MPSP. Following this, experiments concerning the GPSP instances are presented.

[4]http://www.mpsplib.com
[5]https://gent.cs.kuleuven.be/mista2013challenge/

4.4.1 Multi-project scheduling problem

The MPSPLib consists of 140 different instances varying many characteristics, such as number of local and global resources, average utilization factor of global resources, overload factor, among others. A detailed overview of the instances' characteristics is available at the MPSPLib website[4], which provides a user-friendly interface for uploading new solutions and an updated ranking with best solutions submitted. The website also enables visualizing solutions, a very useful feature for assessing resource usage and solution quality.

Table 4.3 presents the results obtained for the complete MPSLib instance set. The algorithm was executed 10 times for each instance, considering the runtime limit of 10 minutes. The table presents the best known results (BKS) as reported on the website, in addition to the best and average results obtained by the decomposition-based heuristic. Note that the MPSP's primary objective is to minimize the *Average Project Delay* (APD), which is simply the quotient of TPD and the number of projects, $|P|$. APD is therefore an objective equivalent to TPD.

The table demonstrates the efficiency of the proposed decomposition-based heuristic. From the 140 instances considered, 105 had their best known solution improved. For the remaining 35 instances, the solution produced by the proposed algorithm corresponds to the best one in the literature for 13 instances. This result becomes more impressive once one observes that these best known solutions were collected from the best result of over 20 different algorithms, according to the MPSPLib website.

Table 4.3: Results for MPSPLib instances

Instance	BKS		Heuristic best		Heuristic avg.	
	APD	TMS	APD	TMS	APD	TMS
mp_j30_a2_nr1	13.0	80	11.5	77	12.0	74.2
mp_j30_a2_nr2	15.0	60	15.0	60	15.5	60.4
mp_j30_a2_nr3	3.0	65	3.0	65	3.0	65.0
mp_j30_a2_nr4	10.5	54	10.5	54	10.5	54.0
mp_j30_a2_nr5	8.5	58	8.5	58	8.6	58.2
mp_j30_a5_nr1	11.8	92	10.2	87	10.2	88.8
mp_j30_a5_nr2	14.2	83	12.2	79	12.6	83.0
mp_j30_a5_nr3	31.0	112	26.4	105	28.0	111.0
mp_j30_a5_nr4	0.0	76	0.0	76	0.0	76.0
mp_j30_a5_nr5	14.8	101	15.0	94	15.6	96.8

(continued on next page)

Table 4.3 continued: Results for MPSPLib instances

Instance	BKS		Heuristic best		Heuristic avg.	
	APD	TMS	APD	TMS	APD	TMS
mp_j30_a10_nr1	82.2	197	79.4	196	83.0	189.6
mp_j30_a10_nr2	11.3	119	7.7	109	8.1	109.4
mp_j30_a10_nr3	86.3	259	79.7	246	85.3	254.0
mp_j30_a10_nr4	22.1	169	16.8	154	21.3	152.0
mp_j30_a10_nr5	51.6	202	53.3	197	55.5	197.8
mp_j30_a20_nr1	190.2	441	186.9	432	196.7	426.4
mp_j30_a20_nr2	77.0	305	72.5	294	74.5	294.2
mp_j30_a20_nr3	99.5	327	93.5	328	96.3	318.8
mp_j30_a20_nr4	34.0	212	32.8	190	42.7	194.6
mp_j30_a20_nr5	156.4	464	151.6	448	159.8	444.8
mp_j90_a2_nr1	0.0	88	0.0	88	0.0	88.0
mp_j90_a2_nr2	26.5	127	20.0	117	20.2	118.6
mp_j90_a2_nr3	0.0	114	0.0	114	0.0	114.0
mp_j90_a2_nr4	0.0	92	0.0	92	0.0	92.0
mp_j90_a2_nr5	0.0	121	0.0	121	0.0	121.0
mp_j90_a5_nr1	0.0	79	0.0	79	0.0	79.0
mp_j90_a5_nr2	7.4	114	6.0	114	6.0	114.0
mp_j90_a5_nr3	3.0	141	1.6	138	1.8	138.0
mp_j90_a5_nr4	8.4	133	2.0	123	2.6	123.4
mp_j90_a5_nr5	14.0	155	12.2	153	12.9	154.2
mp_j90_a10_nr1	55.1	201	49.2	176	51.0	169.0
mp_j90_a10_nr2	2.7	137	0.0	128	0.0	128.0
mp_j90_a10_nr3	45.2	235	42.2	225	44.0	224.5
mp_j90_a10_nr4	1.0	150	0.7	150	0.7	150.0
mp_j90_a10_nr5	54.4	250	52.4	278	53.3	271.2
mp_j90_a20_nr1	0.0	97	0.0	97	0.0	97.0
mp_j90_a20_nr2	4.6	168	2.5	164	2.7	163.6
mp_j90_a20_nr3	3.0	122	1.2	122	1.2	122.0
mp_j90_a20_nr4	31.6	208	28.2	195	29.1	195.2
mp_j90_a20_nr5	46.3	255	46.9	244	48.6	252.5
mp_j120_a2_nr1	40.0	182	34.5	173	35.6	168.6
mp_j120_a2_nr2	29.0	137	24.0	133	25.3	135.4
mp_j120_a2_nr3	119.0	295	106.5	277	116.0	276.4
mp_j120_a2_nr4	44.5	152	40.0	148	41.4	164.0
mp_j120_a2_nr5	2.0	111	0.0	108	0.1	108.2
mp_j120_a5_nr1	6.2	81	2.0	75	2.5	76.0
mp_j120_a5_nr2	30.2	178	23.8	164	24.8	166.6
mp_j120_a5_nr3	68.2	233	63.6	247	67.2	227.6
mp_j120_a5_nr4	55.6	226	48.2	197	53.1	201.0
mp_j120_a5_nr5	76.4	284	71.0	278	77.6	276.0
mp_j120_a10_nr1	43.5	152	36.9	139	39.1	137.8
mp_j120_a10_nr2	71.0	275	62.9	297	63.8	290.2
mp_j120_a10_nr3	8.5	155	3.4	154	4.0	149.5
mp_j120_a10_nr4	193.0	416	178.1	400	191.4	397.2
mp_j120_a10_nr5	184.2	520	186.4	504	198.7	507.2

(continued on next page)

Table 4.3 continued: Results for MPSPLib instances

Instance	BKS		Heuristic best		Heuristic avg.	
	APD	TMS	APD	TMS	APD	TMS
mp_j120_a20_nr1	6.4	82	2.8	76	3.0	77.7
mp_j120_a20_nr2	33.5	222	26.1	204	30.2	236.4
mp_j120_a20_nr3	40.7	242	32.8	235	33.6	234.6
mp_j120_a20_nr4	25.9	209	17.9	203	18.6	209.2
mp_j120_a20_nr5	24.6	183	15.9	178	16.7	183.2
mp_j90_a2_nr5_AgentCopp1	66.0	200	67.0	190	68.4	188.6
mp_j90_a2_nr5_AgentCopp2	186.5	352	173.5	334	182.1	333.2
mp_j90_a2_nr5_AgentCopp3	45.0	177	36.5	160	39.1	162.0
mp_j90_a2_nr5_AgentCopp4	187.5	346	175.0	329	187.7	332.0
mp_j90_a2_nr5_AgentCopp5	0.0	72	0.0	72	0.0	72.0
mp_j90_a2_nr5_AgentCopp6	69.5	197	66.5	187	69.7	186.2
mp_j90_a2_nr5_AgentCopp7	179.0	338	170.5	330	181.5	330.2
mp_j90_a2_nr5_AgentCopp8	46.5	174	38.0	157	39.2	158.4
mp_j90_a2_nr5_AgentCopp9	185.5	336	179.0	329	192.2	330.6
mp_j90_a2_nr5_AgentCopp10	26.0	117	21.0	109	21.8	108.6
mp_j90_a5_nr5_AgentCopp1	252.0	610	228.4	579	246.4	572.4
mp_j90_a5_nr5_AgentCopp2	291.2	687	263.8	639	284.9	627.0
mp_j90_a5_nr5_AgentCopp3	109.8	357	97.2	330	106.8	326.6
mp_j90_a5_nr5_AgentCopp4	439.8	861	418.6	816	460.3	820.6
mp_j90_a5_nr5_AgentCopp5	98.0	278	88.8	263	99.6	261.4
mp_j90_a5_nr5_AgentCopp6	257.2	613	235.4	574	249.6	566.8
mp_j90_a5_nr5_AgentCopp7	292.0	689	265.4	631	288.5	627.6
mp_j90_a5_nr5_AgentCopp8	115.8	359	101.6	327	107.0	322.8
mp_j90_a5_nr5_AgentCopp9	442.4	838	428.2	808	476.0	818.8
mp_j90_a5_nr5_AgentCopp10	107.4	278	98.4	264	108.4	260.2
mp_j90_a10_nr5_AgentCopp1	266.6	755	246.2	693	266.1	696.6
mp_j90_a10_nr5_AgentCopp2	280.0	786	277.8	744	298.5	744.6
mp_j90_a10_nr5_AgentCopp3	62.3	287	62.4	262	70.2	272.0
mp_j90_a10_nr5_AgentCopp4	303.5	704	290.8	674	321.2	669.0
mp_j90_a10_nr5_AgentCopp5	142.6	393	125.8	365	145.5	365.6
mp_j90_a10_nr5_AgentCopp6	278.3	738	254.5	690	277.4	691.0
mp_j90_a10_nr5_AgentCopp7	135.2	433	134.7	396	142.0	404.0
mp_j90_a10_nr5_AgentCopp8	79.1	295	82.9	282	94.3	287.4
mp_j90_a10_nr5_AgentCopp9	82.3	244	79.1	240	87.5	237.2
mp_j90_a10_nr5_AgentCopp10	63.8	203	51.2	182	58.0	179.6
mp_j90_a20_nr5_AgentCopp1	139.8	474	147.2	456	160.1	465.2
mp_j90_a20_nr5_AgentCopp2	0.0	127	0.0	127	0.0	127.0
mp_j90_a20_nr5_AgentCopp3	15.4	188	10.2	163	11.7	161.5
mp_j90_a20_nr5_AgentCopp4	118.4	366	116.8	363	127.5	359.5
mp_j90_a20_nr5_AgentCopp5	16.4	153	7.5	124	7.8	128.8
mp_j90_a20_nr5_AgentCopp6	62.9	264	65.3	250	68.9	255.4
mp_j90_a20_nr5_AgentCopp7	77.6	300	81.0	298	91.7	284.4
mp_j90_a20_nr5_AgentCopp8	27.1	188	23.5	160	28.1	163.0
mp_j90_a20_nr5_AgentCopp9	171.8	430	171.7	428	186.8	417.2
mp_j90_a20_nr5_AgentCopp10	224.9	525	210.6	485	239.0	489.6

(continued on next page)

Table 4.3 continued: Results for MPSPLib instances

Instance	BKS		Heuristic best		Heuristic avg.	
	APD	TMS	APD	TMS	APD	TMS
mp_j120_a2_nr5_AgentCopp1	81.0	238	69.0	215	71.2	215.6
mp_j120_a2_nr5_AgentCopp2	6.0	111	2.5	104	2.8	104.6
mp_j120_a2_nr5_AgentCopp3	69.0	198	56.5	178	60.1	178.4
mp_j120_a2_nr5_AgentCopp4	8.0	106	3.0	105	4.1	105.4
mp_j120_a2_nr5_AgentCopp5	10.0	105	5.5	96	5.8	96.6
mp_j120_a2_nr5_AgentCopp6	81.0	234	70.5	212	71.8	214.2
mp_j120_a2_nr5_AgentCopp7	7.0	113	2.5	104	2.6	104.2
mp_j120_a2_nr5_AgentCopp8	69.5	192	58.5	179	60.7	176.0
mp_j120_a2_nr5_AgentCopp9	7.0	99	1.0	96	1.4	96.6
mp_j120_a2_nr5_AgentCopp10	9.0	102	4.0	93	4.7	93.4
mp_j120_a5_nr5_AgentCopp1	231.2	644	209.4	596	231.5	597.6
mp_j120_a5_nr5_AgentCopp2	82.4	319	70.0	294	77.7	295.4
mp_j120_a5_nr5_AgentCopp3	230.0	570	209.4	531	243.8	531.6
mp_j120_a5_nr5_AgentCopp4	120.8	421	112.2	386	115.6	382.6
mp_j120_a5_nr5_AgentCopp5	218.0	677	193.8	622	215.0	624.8
mp_j120_a5_nr5_AgentCopp6	234.6	634	215.6	591	233.4	591.4
mp_j120_a5_nr5_AgentCopp7	87.4	314	76.4	293	86.3	292.4
mp_j120_a5_nr5_AgentCopp8	232.6	558	215.0	531	248.0	527.6
mp_j120_a5_nr5_AgentCopp9	124.0	401	112.0	371	119.8	370.6
mp_j120_a5_nr5_AgentCopp10	222.4	668	200.8	618	216.0	620.4
mp_j120_a10_nr5_AgentCopp1	268.9	824	260.3	779	275.5	778.8
mp_j120_a10_nr5_AgentCopp2	110.9	413	107.0	401	117.2	394.7
mp_j120_a10_nr5_AgentCopp3	165.0	508	184.9	488	190.9	489.5
mp_j120_a10_nr5_AgentCopp4	122.9	455	118.7	430	127.9	427.7
mp_j120_a10_nr5_AgentCopp5	95.0	435	86.5	420	94.4	417.0
mp_j120_a10_nr5_AgentCopp6	106.6	413	117.9	397	132.5	393.2
mp_j120_a10_nr5_AgentCopp7	19.4	163	16.5	143	17.1	145.5
mp_j120_a10_nr5_AgentCopp8	31.7	184	28.4	184	31.1	182.8
mp_j120_a10_nr5_AgentCopp9	22.7	185	16.8	165	18.1	160.3
mp_j120_a10_nr5_AgentCopp10	21.0	198	18.2	192	18.7	195.0
mp_j120_a20_nr5_AgentCopp1	103.3	421	112.7	419	128.8	391.7
mp_j120_a20_nr5_AgentCopp2	67.4	327	69.2	314	77.1	317.3
mp_j120_a20_nr5_AgentCopp3	355.2	1013	380.1	1037	394.2	1011.8
mp_j120_a20_nr5_AgentCopp4	65.0	346	70.1	326	74.4	325.6
mp_j120_a20_nr5_AgentCopp5	86.0	390	88.0	373	91.1	379.2
mp_j120_a20_nr5_AgentCopp6	313.6	936	346.8	874	366.3	888.4
mp_j120_a20_nr5_AgentCopp7	303.1	886	343.6	839	351.0	846.0
mp_j120_a20_nr5_AgentCopp8	120.0	393	131.2	408	140.4	404.4
mp_j120_a20_nr5_AgentCopp9	76.5	333	80.8	322	86.8	312.8
mp_j120_a20_nr5_AgentCopp10	95.7	394	105.5	361	112.2	361.5

4.4.2 Generalized project scheduling problem

Experiments for the GPSP were performed on the benchmark instances produced for the MISTA Challenge 2013. Information concerning how these instances were generated is provided by Wauters et al. (2016). Table 4.4 shows the

instances' characteristics, presenting the number of projects $|P|$, number of jobs $|J|$, number of precedence relations between jobs $|E|$ and the amount of global renewable $|R_g|$, local renewable $|R_l|$ and non-renewable resources $|K|$ for each instance. The table also reports the average duration $\langle d_{j,m} \rangle$ of jobs, the average number of execution modes $\langle M_j \rangle$ per job and, finally, the average critical path duration of the projects $\langle \text{CPD} \rangle$.

Table 4.4: Characteristics of the MISTA Challenge 2013 instances

Instance	Dimensions						Average Dimensions		
	$\|P\|$	$\|J\|$	$\|E\|$	$\|R_g\|$	$\|R_l\|$	$\|K\|$	$\langle d_{jm} \rangle$	$\langle \|M_j\| \rangle$	$\langle \text{CPD} \rangle$
A-1	2	24	36	1	2	4	5.19	2.67	14.50
A-2	2	44	80	1	2	4	4.90	2.82	22.50
A-3	2	64	116	1	2	4	5.48	2.88	33.50
A-4	5	60	90	1	5	10	5.01	2.67	14.20
A-5	5	110	200	1	5	10	5.62	2.82	23.00
A-6	5	160	290	1	5	10	5.07	2.88	25.60
A-7	10	120	180	2	0	20	5.33	2.67	16.80
A-8	10	220	400	2	0	20	5.26	2.82	24.60
A-9	10	320	580	1	10	20	5.32	2.88	29.60
A-10	10	320	580	1	10	20	5.48	2.88	30.70
B-1	10	120	180	1	10	20	5.23	2.67	12.90
B-2	10	220	400	2	0	20	5.46	2.82	23.90
B-3	10	320	580	1	10	20	5.38	2.88	29.50
B-4	15	180	270	1	15	30	5.35	2.67	15.80
B-5	15	330	600	1	15	30	5.33	2.82	22.53
B-6	15	480	870	1	15	30	5.39	2.88	31.13
B-7	20	240	360	1	20	40	5.15	2.67	15.35
B-8	20	440	800	2	0	40	5.27	2.82	23.65
B-9	20	640	1160	1	20	40	5.46	2.88	30.10
B-10	20	460	816	2	0	40	5.29	2.83	24.45
X-1	10	120	180	2	0	20	5.07	2.67	14.90
X-2	10	220	400	1	10	20	5.32	2.82	23.00
X-3	10	320	580	1	10	20	5.38	2.88	29.90
X-4	15	180	270	2	0	30	5.24	2.67	14.87
X-5	15	330	600	1	15	30	5.19	2.82	23.60
X-6	15	480	870	1	15	30	5.34	2.88	29.93
X-7	20	240	360	1	20	40	4.94	2.67	14.95
X-8	20	440	800	1	20	40	5.34	2.82	24.45
X-9	20	640	1160	1	20	40	5.24	2.88	28.90
X-10	20	450	798	1	20	40	5.30	2.82	24.10

Table 4.5 provides the results obtained after 10 runs of the proposed approach within the runtime limit established by the MISTA Challenge 2013 organizers (300 seconds). Solutions and additional data are available online[6]. The proposed algorithm was capable of improving seven solutions when compared to the best

[6]https://benchmark.gent.cs.kuleuven.be/psp

results from the challenge. Nevertheless, the stochastic local search proposed by Asta et al. (2016) remains a better overall algorithm for the GPSP.

Table 4.5: Results for MISTA Challenge 2013 instances

Instance	MISTA		Heuristic best		Heuristic avg.	
	TPD	TMS	TPD	TMS	TPD	TMS
A-1	1	23	1	23	1.0	23.0
A-2	2	41	2	41	2.0	41.0
A-3	0	50	0	50	0.0	50.0
A-4	65	42	66	47	67.8	48.5
A-5	153	105	154	104	159.9	106.7
A-6	147	96	144	95	150.8	97.5
A-7	596	196	605	190	611.9	197.2
A-8	302	155	280	144	291.4	146.2
A-9	223	119	206	124	216.8	122.8
A-10	969	314	942	304	957.4	310.1
B-1	349	127	358	131	367.3	127.9
B-2	434	160	437	160	449.1	158.5
B-3	545	210	568	204	600.0	202.2
B-4	1274	289	1465	287	1510.6	297.1
B-5	820	254	875	254	896.6	253.8
B-6	912	227	952	221	991.2	225.8
B-7	792	228	876	234	935.0	235.3
B-8	3176	533	3032	513	3097.1	517.6
B-9	4192	746	4523	736	4695.3	753.3
B-10	3249	456	2974	419	3145.7	426.6
X-1	392	142	390	141	401.2	142.7
X-2	349	163	389	166	407.4	168.0
X-3	324	192	325	177	336.0	176.8
X-4	955	213	967	202	980.0	203.6
X-5	1768	374	1871	370	1988.0	369.8
X-6	719	232	777	231	812.4	235.4
X-7	861	237	909	232	930.0	230.0
X-8	1233	283	1328	279	1404.1	288.8
X-9	3268	643	3468	646	3681.8	655.4
X-10	1600	381	1718	377	1793.8	387.9

Figure 4.3 shows the gap between the results of the 10 runs for all instances in several boxplots. The proposed approach performs very well for some instances, where the generated solutions are almost 9% better than the best ones from the MISTA Challenge 2013. For other instances, however, the algorithm is outperformed by the challenge's winner. Overall, the algorithm is competitive.

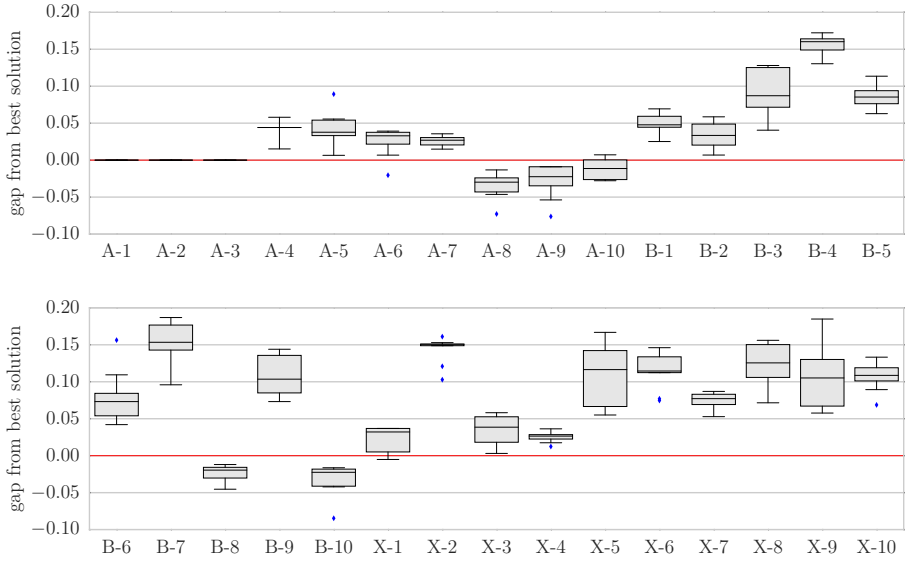


Figure 4.3: Solution values obtained after 10 algorithm runs on the MISTA Challenge 2013 instances

4.5 Conclusions and future work

This chapter presented an IP formulation and a decomposition-based heuristic algorithm with several IP-based components for the *Multi-Mode Resource-Constrained Multi-Project Scheduling Problem*, referred to as *Generalized Project Scheduling Problem* (GPSP). The decomposition applied is very similar to those employed in Chapters 2 and 3. However, some additional components were necessary due to the problem's characteristics.

Much like previous chapters, strong results were obtained. The algorithm was proven competitive for the GPSP and obtained remarkable results for the MPSP, a special case of the GPSP with single execution modes. 75% of the MPSPLib instances had their best known solution improved.

In contrast to the decomposition-based algorithms proposed in Chapters 2 and 3, here the subproblem optimization scheme was embedded within a metaheuristic framework. Despite this difference, one special characteristic was maintained: the employment of IP solvers for solving subproblems. It should be highlighted that solving large problems is too time consuming for the

current generation of IP solvers. However, as we keep researching new ways of addressing these subproblems, the continued evolution of IP solvers significantly contributes towards solving larger subproblems by the same algorithm. Since larger subproblems tend to lead to better solutions, the algorithm's efficiency will be positively impacted by new findings in the field of integer programming. In this context, as highlighted by Fischetti et al. (2010), decomposition-based algorithms such as the one presented in this chapter are very desirable.

Finally, there are several recommendations for future work, as the proposed approach still has room for improvement. One of the key characteristics of the algorithm is the heuristic objective function, which relies on estimations that should be further analyzed. Note how during the constructive phase the proposed objective function employs the critical path as an estimation of a job's schedule impact. The principle is to prioritize jobs that would delay the project's completion the most, however the current estimations may be imprecise. A more precise estimation could consider, for example, other jobs' resource consumption within the project. It is very likely that more suitable objective functions will lead the proposed algorithm towards obtaining better solutions.

Chapter 5

Towards a general solver

Chapters 2, 3 and 4 have proposed different decomposition strategies for optimizing the TUP, NRP and PSP, respectively. For all three problems, decomposition-based heuristics relying on each problem's structure were evaluated, resulting in improvements over the state-of-the-art algorithms. Although differences may be observed when applying the heuristic decompositions to these problems, many similarities are also present. In this chapter, we exploit these similarities to prototype a general decomposition-based heuristic framework, which may be employed as a general solver. Our objective is to investigate to which extent the methods proposed in previous chapters are general.

This chapter provides a starting point for research concerning general decomposition-based heuristic frameworks. The primary components of such a framework are first discussed before evaluating a simplified version considering four different problems. Such a discussion is presented in Section 5.1, which enumerates the principles behind the framework and relates them with the knowledge acquired from the previous chapters. Additionally, requirements and challenges for deriving a general decomposition-based heuristic framework are considered. Next, Section 5.2 describes the framework's initial implementation, presenting its algorithmic components. Preliminary experiments concerning this initial implementation on different problems are presented in Section 5.3. Finally, Section 5.4 concludes this chapter, debating challenges and future research directions for building a competitive general decomposition-based heuristic solver.

5.1 Methodology

Chapters 2, 3 and 4 have successfully applied decomposition-based heuristic procedures to three different problems, resulting in state-of-the-art approaches for each of them. The decomposition applied in these chapters, while employing similar concepts, required circumventing different challenges. This section briefly discusses these challenges, paving the way for a general framework.

The general framework's scheme is presented by Figure 5.1. Note that it is composed by input data and a subproblem solver. The principle is to automatically generate constructive and local search algorithms similar to those presented in the previous chapters by receiving the follow input: problem, decomposition(s) and subproblem characteristics. Additionally, a subproblem solver should be specified (MIP solvers were employed in Chapters 2, 3 and 4). The output is a solution to the problem.

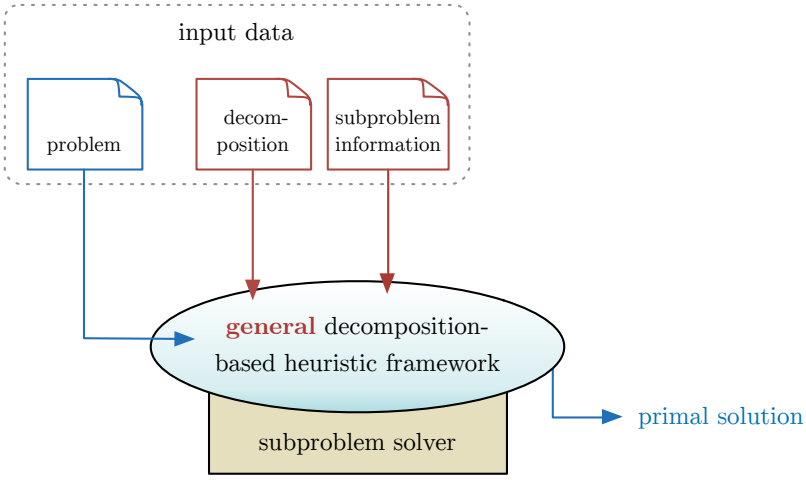


Figure 5.1: Illustration of the general framework components

Defining the input data represents one of the challenges in building such a general framework. These challenges are addressed in the following sections: how to define the *problem*, the *decomposition* and *subproblem characteristics*? Later on, we proceed to the description of the framework's algorithmic components.

5.1.1 Defining the problem

As with any general framework, different and often unexpected problems should be addressable. To that end, it is necessary to define means of formally describing individual problems. In the general framework discussed throughout this chapter, problems are described via MIP formulations. On the one hand, this enables straightforward integration with MIP solvers, meaning that a subproblem solver is always available. It also reduces the learning curve required to use the framework for those who are familiar with MIP. On the other hand, however, requiring a MIP formulation has disadvantages. Different formulations for the same problem may lead to completely different results. The general framework therefore becomes dependent on the MIP formulation. Moreover, requiring the problem to be formulated by linear equations reduces the range of problems the framework is capable of addressing. There are also disadvantages concerning memory usage. Given that MIP formulations rarely correspond to the most compact way of defining a problem, more memory than necessary is often utilized. Despite these disadvantages, such a design decision enabled quickly implementing an initial version for the framework.

5.1.2 Defining the decompositions

The approaches proposed for the TUP, NRP and PSP utilize the time structure of these problems when decomposing them. Despite the novelty of the algorithms proposed, exploiting such a structure is by no means a new idea. In fact, splitting a problem into different subproblems using time windows may be considered a general rule of thumb applied by human planners who manually generate solutions for these problems. Similar situations also occur within other problems. Take those related to vehicle routing, for example. Human planners generally employ the geographic location of customers to heuristically decompose the problem, thereby reducing its size. Given the vast amount of decomposition possibilities, it is essential that the framework allows specifying different (and also unexpected) decompositions.

One of the advantages of employing a MIP formulation to describe the problem is that it simplifies defining the decomposition(s). Essentially, each *minimal subproblem* (smallest possible subproblem) is defined by a set of variables. In the cases of the TUP, NRP and PSP, each minimal subproblem contains the variables corresponding to one *time* (equivalent to a round in the TUP, a day in the NRP and a time-slot in the PSP).

The different minimal subproblems should be connected to each other, so as to enable the creation of larger subproblems which maintain desired problem properties. To that end, a simple graph is employed to indicate related subproblems. In this graph, each minimal subproblem corresponds to a node, with directed arcs indicating related (connected) subproblems. Priorities are assigned to both nodes and edges. This way, nodes (subproblems) with higher priority are explored first. Analogously, arcs with higher priorities are considered first. Larger subproblems are defined by a best first search (BFS). Let P be the large problem, defined in terms of a minimal problem $\mathcal{P}_i$. P is initially set to be equivalent to $\mathcal{P}_i$. Then, the subproblem connected to P by the lowest priority arc is added to P, increasing its size. The procedure continues until a limit η on the number of minimal subproblems is reached or there are no more arcs remaining.

Figure 5.2 illustrates an example considering the TUP, where each node represents a round. For compactness, only the first four rounds are included. Note how the initial round has a higher priority p than those succeeding it. Moreover, all arcs have equal priority, which makes no difference in this particular example since all nodes have exactly one outgoing arc. Finally, note how this graph corresponds to a single path. This is the case for all time-based decompositions proposed throughout the previous chapters.

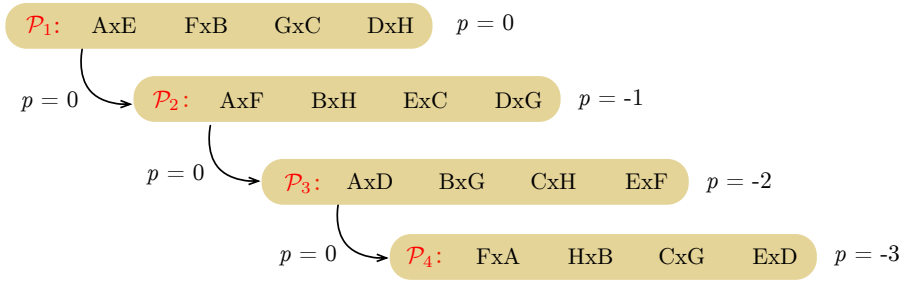


Figure 5.2: Example of decomposition representation for the TUP

In the example of Figure 5.2, obtaining larger problems is straightforward. Let's build a subproblem P with $\eta = 4$ nodes, beginning with the first round (or minimal subproblem $\mathcal{P}_1$). P is first set to be equal to $\mathcal{P}_1$. Then the node connected to P with lowest priority is selected ($\mathcal{P}_2$), being subsequently added to P. Next, the procedure considers the updated $P = \mathcal{P}_1 \cup \mathcal{P}_2$, which results in adding $\mathcal{P}_3$ to P. The procedure repeats until $\eta = 4$, which in this example ultimately results in $P = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_4$.

Note on automatic decomposition detection

The framework here described assumes the input includes the decomposition(s). However, automatically detecting a problem's key characteristics to derive decompositions is definitely a desired feature of any decomposition-based general framework. Indeed, some research groups are investigating the automatic detection of problem structures. One such example is the Operations Research's group from RWTH Aachen University, which investigate automatic problem structure detection to apply Dantzig-Wolfe decomposition within their General Column Generation (GCG) solver[1] (Gamrath and Lübbecke, 2010). In a heuristic context, such as the one here considered, automatic problem structure detection raises many research questions, such as:

- How should the problem be represented so as to facilitate detecting the decomposition(s)?
- What information is mandatory for reasonable detection?
- How to evaluate a decomposition so as to (automatically) differentiate profitable and ineffective ones?
- How to address subproblem specificities?

We believe these and other questions concerning automatic decomposition detection represent very interesting topics for future research projects. In fact, this will be indicated as our main future work direction in Section 5.4.

5.1.3 Defining subproblem characteristics

Chapters 2 and 4 demonstrated the importance of defining heuristic objective functions when applying decomposition-based constructive procedures for the TUP and PSP, respectively. It is therefore desired that the framework implements some mechanism for defining special subproblem characteristics.

Subproblem characteristics are introduced as an additional (possibly incomplete) MIP formulation. This additional formulation enables modifying and extending the formulation for subproblems. Therefore, constraints may be added or rewritten, additional variables may be introduced and the objective function may be completely changed. Each subproblem is thus a combination of the problem's

[1] http://www.or.rwth-aachen.de/gcg

original formulation with the additional MIP. Constraints and variables are identified by their names within the framework, and therefore it is possible to modify portions of the original formulation within subproblems. Note that this abstraction is very flexible: it is in fact possible to overwrite the entire original formulation for subproblems.

Overwriting some subproblem constraints is particularly important when no feasible solution is available, in which case it may be impossible to fix values for some variables. Since a subproblem contains only a subset of the problem's variables, depending on the constraints enforced this subproblem may result in an infeasible MIP. In such situations, the introduction of slack variables is necessary to avoid infeasibility. Additional MIP formulations may be employed to modify such problematic constraints by introducing slack variables, for example. Slack (or auxiliary) variables are identified by the framework as variables not listed within the decomposition, meaning they are not part of any minimal subproblem. These variables are included as part of a subproblem whenever it contains constraints which incorporate them.

Consider again the TUP addressed in Chapter 2 (Section 2.5.1) as an example. An alternative objective function was employed for the subproblems, considering the number of unvisited locations for each umpire. This required adding slack variables to certain constraints for both preventing infeasibility and altering the objective function. This is easily achieved by the inclusion of the incomplete MIP formulation given by Equations (5.1) and (5.2):

Minimize:

$$\sum_{e \in E} \sum_{u \in U} d_e x_{e,u} + \sum_{i \in I} \sum_{u \in U} \psi_u \ell_{i,u} \tag{5.1}$$

Subject to:

$$\sum_{e \in \delta(H_i)} x_{e,u} + \ell_{i,u} \geq 1 \quad \forall i \in I, u \in U \tag{5.2}$$

In these equations, ψ_u is a weight that defines the impact of unvisited locations and $\ell_{i,u}$ indicates whether umpire u visits location i ($\ell_{i,u} = 0$) or not ($\ell_{i,u} = 1$) (see Chapter 2, page 48 for more details). In this example the objective function is rewritten and Equation (5.2) replaces Equation (2.6). Variable $\ell_{i,u}$ is identified as a slack variable, being incorporated into all subproblems including both team i and umpire u.

Finally, the framework also permits using information from the algorithm when solving subproblems. Dedicated keywords may be used within the MIP formulation to include values such as iteration number, minimal subproblem index and subproblem size. This enables, for instance, calculating ψ_u values without modifying the framework source code.

5.2 Algorithmic components

The framework currently implements decomposition-based constructive and local search procedures very similar to those presented in Chapters 2 and 3. Different subproblem MIP formulations may be considered for the constructive procedure and the local search phase. This distinction is important, for example, to prevent infeasibility once a feasible solution is generated by the constructive procedure.

5.2.1 Constructive procedure

Whenever an initial solution is not provided to the framework, a constructive procedure is executed to generate one. Algorithm 5.1 presents this constructive procedure as a recursive method. The algorithm requires four arguments: a solution S (initially empty), the current minimal subproblem (initially one) and parameters η and *step*, which define the size and degree of intersection between subproblems (and therefore number of subproblems), respectively. Here η and *step* have the same meaning as in the previous chapters and should be defined such that $step \leq \eta$. Algorithm 5.1 is very similar to Algorithm 2.3 (presented in Chapter 2), with some minor differences. Similarly, the stopping criterion is also a complete feasible solution (lines 1-2). Until the solution is complete, a subproblem P is defined (line 3) by a BFS on the minimal subproblems graph beginning with $\mathcal{P}_i$. When searching, the arc with lowest priority is selected next (see Section 5.1.2). The BFS stops once either η nodes are selected or all arcs have been considered. The algorithm then proceeds by solving the resulting subproblem P (line 4). All solutions obtained are stored in a list L, together with the optimal solution. Beginning with the best solution, the algorithm iterates over all solutions $S_P \in L$ (line 5). The assignments in S_P are then included in S (line 6). Afterwards, variable j is initialized and updated so as to indicate the next minimal subproblem (lines 7-9). Note how j is incremented until an unsolved subproblem $\mathcal{P}_j$ is detected or the value of $i + step$ is reached.

Next, a recursive call is made to solve the remaining subproblems (line 10). If a feasible solution is obtained, the algorithm returns it (line 11). Otherwise, it removes subproblem solution S_P from S (line 12) and continues with the next solution. If no subproblem solution in L results in a feasible solution S, an empty solution is returned (line 13).

Algorithm 5.1: General decomposition-based constructive algorithm

```
   Let 𝒫 be the ordered list of minimal subproblems given by the decomposition
   Input: Solution S (initially empty), minimal subproblem i (initially one),
          subproblem size η and intersection parameter step
   Constructive(S, i, η, step)
1    if S is a feasible solution then
2        return S // success: feasible solution is returned
3    P ← subproblem defined by BFS on 𝒫_i with up to η nodes (subproblems)
4    L ← list of feasible solutions for P, sorted by increasing cost
5    for S_P ∈ L do
6        S ← S ∪ S_P // set values for variables corresponding with subproblem P
7        j ← i + 1 // pointer j identifies the next minimal subproblem
8        while j < i + step and S contains solution for 𝒫_j do
9            j ← j + 1 // skip the next minimal subproblem (based on parameter step)
10       if Constructive(S, j, η, step) ≠ ∅ then
11           return S
12       S ← S\S_P // unset values for variables corresponding with subproblem P
13   return ∅ // backtracks since all subproblem solutions resulted in infeasibility
```

5.2.2 Local search procedure

The local search phase begins after a feasible solution is obtained by the constructive procedure. The principle behind the local search is to solve subproblems while fixing values for variables not included in the subproblem. It is a concept similar to the hard fixation of variables in *Relaxation Induced Neighborhood Search* (RINS) (Danna et al., 2003). However, in the framework subproblems may have different characteristics (Section 5.1.3) and problem-specific decompositions may lead to higher-quality solutions. Multiple decomposition strategies may be employed, identified by set $\mathcal{D}$. Parameters η^d and $step^d$ control the subproblem size and number of subproblems for decomposition $d \in \mathcal{D}$, respectively, while $\bar{\eta}^d$ establishes an upper limit for η^d's value. Subproblems are, again, generated by a greedy BFS on the graph correlating the different subproblems (Section 5.1.2). Once no improvement

is achievable from subproblems of size η^d, the value of η^d is increased. The algorithm continues until some stopping criterion is met or a local optimum for all subproblems is obtained.

The procedure responsible for generating subproblems is described by Algorithm 5.2. First the list of subproblems $\mathcal{S}$ is initialized (line 1). For each decomposition (line 2), minimal subproblems are sorted with ties on priorities resolved randomly (line 3). The index of each minimal subproblem is stored in K (line 4), which is used to sort out ties during the generation of subproblems by BFS. Variables c_j are defined to control subproblem utilization (line 5) and pointer i is initialized to consider the first minimal subproblem of list $\mathcal{P}^d$ (line 6). Afterwards, subproblems are constructed. The algorithm iterates until all minimal subproblems have been assigned to a subproblem (line 7). Subproblems are created (line 8) and added to list $\mathcal{S}$ (line 9). Variables c_j are then updated (line 10). Next, pointer i is updated to generate the next subproblem (lines 11-13). Ideally, *step* minimal subproblems are skipped, however it may be necessary

Algorithm 5.2: Local search subproblems generation

Let $\mathcal{P}^d$, η^d and $step^d$ be properties of decomposition d, where $\mathcal{P}^d$ represents the ordered list of minimal subproblems, η^d the subproblem size and $step^d$ the intersection parameter
Input: Decompositions $\mathcal{D}$
MakeSubproblems($\mathcal{D}$)

```
1  S ← ∅
2  for each decomposition d ∈ D do
3      sort P^d by priority while shuffling elements with equal priorities
4      K ← map of minimal subproblem indices after sorting/shuffling P^d
5      c_j ← false for j ∈ {1, ..., |P^d|}
6      i ← 1 // pointing to the first minimal subproblem
7      while ∃c_j=false for j ∈ {1, ..., |P^d|} do
8          P ← subproblem given by BFS on P_i^d with up to η^d nodes using K
9          S ← S ∪ {P}
10         c_j ← true for all minimal subproblems indices j included in P
           // updating pointer i by skipping at most step subproblems
11         for k ← 1 to step^d do
12             i ← (i mod |P^d|) + 1
13             if c_i=false then break for-loop
       // sorting subproblems: both decomposition and subproblem priorities are considered
14     sort S by priority while shuffling elements with equal priorities
15     return S
```

to ensure that all minimal subproblems are included and, therefore, i may be increased by a value smaller than *step*. Once all subproblems were generated, they are sorted (line 14). Again, ties on priorities are solved by shuffling. Note that priorities from both the decomposition and minimal subproblems are utilized. Finally, the list of subproblems is returned by the procedure (line 15).

The decomposition-based local search method is detailed by Algorithm 5.3. Three arguments are required: a feasible initial solution S, the set of decompositions $\mathcal{D}$ and the re-optimization parameter *reopt*, which indicates whether subproblems should be solved more than once. Initially, subproblems for all decompositions $d \in \mathcal{D}$ are generated and stored in list P (line 1). The algorithm iterates until either all subproblems are optimized or a stopping criterion (generally time or iteration limit) is met (line 2). Next, variable Δ is set to zero (line 3) and pointers i and ℓ are initialized (line 4). The algorithm then iterates until $i = \ell$ or some stopping criteria is met (line 5). Subproblem P_i is solved (line 6) and Δ is updated to store the gain in solution quality (line 7). When parameter *reopt* is true and the solution improves, ℓ is updated and

Algorithm 5.3: General decomposition-based local search

Let η^d and $step^d$ be properties of decomposition d, where η^d represents the subproblem size and $step^d$ the intersection parameter
Input: Solution S, list of decompositions $\mathcal{D}$, and parameter *reopt*
LocalSearch(S**,** $\mathcal{D}$**,** *reopt***)**

```
1  P ← MakeSubproblems(D)
2  while no stopping criteria met do
3      Δ ← 0 // indicates improvements upon current solution
4      i ← 1, ℓ ← |P| // pointers to first and last subproblems, respectively
5      while i ≠ ℓ and no stopping criteria met do
6          update S with optimum solution of subproblem P_i
7          Δ ← Δ + difference in objective value after solving P_i
8          if reopt = true and Δ < 0 then
9              Δ ← 0
10             ℓ ← i
11         i ← (i mod |P|) + 1
           // updating η and step for all decompositions d (up to the limits defined by η̄)
12     if Δ < 0 or some decomposition d ∈ D may be updated then
13         update all η^d and step^d following rules for their decomposition d
14         P ← MakeSubproblems(D)
15     else break while-loop
16 return S
```

Δ reset (lines 8-10). Next, pointer i is updated to consider the next minimal subproblem (line 11). Once $i = \ell$, all subproblems in P have been resolved. Therefore new subproblems are generated (lines 12-14) or the loop finishes (line 15). New subproblems are generated when either Δ indicates some improvement or there is an η parameter which may be updated. Once one of the stopping criteria is reached, the best solution obtained so far is returned (line 16).

5.3 Framework validation

The initial version of the general framework was implemented in Java 8. The problem and its subproblem characteristics are currently defined by MIP formulations via either *lp* or *mps* files. For more information concerning these files, readers are directed to the documentation of *lpsolve*[2] (Berkelaar et al., 2017). A *json*[3] file is employed to describe the decompositions and their properties: minimal subproblem, the graph connecting them (see Section 5.1.2), minimum and maximum values for η, *step*, and other parameter values.

The framework implements a modeling layer, which enables employing different MIP solvers and even avoiding input files (the formulation may be directly implemented within the framework). Three MIP solvers are currently supported: CPLEX, Gurobi and SCIP. The experiments presented in this section were executed with Gurobi 7.5, given that it had the best performance among the three supported solvers.

To validate the framework, four problems were considered. First, the framework was employed to implement the decomposition-based heuristics introduced by the previous chapters of this thesis for the TUP, NRP and PSP. The results are described in Section 5.3.1. Next, Section 5.3.2 describes how the framework was employed to address an additional problem.

5.3.1 Validation with the addressed problems

The framework was utilized to solve the problems addressed by Chapters 2, 3 and 4. Experiments considering these problems are described in the following paragraphs.

[2] http://lpsolve.sourceforge.net/5.5
[3] http://www.json.org

Traveling Umpire Problem. Input files were created for the TUP, as described throughout this chapter. The framework was then executed as a general solver, considering only the input files, and successfully reproduced the results obtained by the decomposition-based heuristic presented in Chapter 2. The runtime was approximately 8% longer on average for the constructive approach, due to the additional computational effort required for parsing files and creating partial formulations with the general code. Nevertheless, such small runtime increase is irrelevant in practice, and thus the general solver may completely replace the problem-specific code developed for the TUP.

Nurse Rostering Problem. The decomposition-based heuristic for the NRP employs three decomposition strategies, which consider the problem's time structure, subsets of nurses and the different possible shifts. Therefore, three decompositions were added to the *json* file: *Time*, *Nurse* and *Shift*. By employing equal priorities, a random ordering of nurses for the *Nurse*-based decomposition was achieved. Decompositions were also given equal priorities. Additionally, an initial solution produced by the greedy algorithm was passed to the solver, bypassing the constructive phase. No additional MIP formulations were employed and the source code required no modification. The results were successfully reproduced, however longer runtimes were observed to produce similar solutions, as for the TUP.

Project Scheduling Problem. In contrast to the approaches developed for the TUP and NRP, the one developed for the PSP is not generalized by the framework in its current version. The source code had to be modified and extended to enable the implementation of the decomposition-based heuristic presented in Chapter 4. The *ModeSel* formulation and a dedicated subproblem solver including the *Forward-Backward Improvement* procedure (see Section 4.3.2) were implemented and the local search algorithm was adapted to reflect the metaheuristic described in Section 4.3.3. Furthermore, the heuristic estimations employed within the MIP formulations also required additional code. In practice, the modeling layer and parts of the constructive algorithm were the only completely reused components, with the remainder requiring several modifications. Such modifications, although not trivial, enabled the framework to reproduce the results obtained in Chapter 4 with some additional computational overhead of less than one minute on average, which may be considered negligible in practice.

5.3.2 Validation with another problem

In order to successfully validate the initial implementation of the general framework, we employed it to solve a problem not addressed by this research. The *Shift Minimization Personnel Task Scheduling Problem* (SMPTSP) explained in full length by Smet et al. (2014) was selected. The SMPTSP may be described as the problem of assigning $J = \{1, ..., n\}$ tasks to $W = \{1, ..., m\}$ workers while minimizing the number of workers with assigned tasks. Each task $j \in J$ is associated with a duration d_j and fixed start and finishing times. Each worker $w \in W$ can perform a subset $T_w \subseteq J$ of the tasks based on his qualifications and availability. All tasks must be assigned to workers while ensuring that each worker executes at most one task at a time, meaning overlap is not permitted.

Smet et al. (2014) presented a MIP formulation for the SMPSTP considering two variable sets:

$$x_{j,w} = \begin{cases} 1 & \text{if task } j \in J \text{ is assigned to worker } w \in W \\ 0 & \text{otherwise} \end{cases}$$

$$y_w = \begin{cases} 1 & \text{if worker } w \text{ has at least one task assigned} \\ 0 & \text{otherwise} \end{cases}$$

Their formulation is presented by Equations (5.3)-(5.7). The objective function (5.3) minimizes the number of workers with tasks assigned. Constraints (5.4) ensure each task is assigned to exactly one worker. Constraints (5.5) guarantee there is no overlap between tasks, meaning each worker executes at most one task at a time. They also connect variables x and y. These constraints utilize a conflict graph constructed with tasks as nodes and edges connecting tasks which overlap. The set of maximal cliques in this graph is defined as $C = \{K_1, ..., K_t\}$, with every pair of tasks in a clique $K \in C$ overlapping in time. Finally, Constraints (5.6) set the bounds for variables y and Constraints (5.7) declare variables x as binary.

Minimize:

$$\sum_{w \in W} y_w \tag{5.3}$$

Subject to:

$$\sum_{w \in W : j \in T_w} x_{j,w} = 1 \quad \forall j \in J \tag{5.4}$$

$$\sum_{j \in T_w \cap K} x_{j,w} \leq y_w \quad \forall w \in W, K \in C \tag{5.5}$$

$$0 \leq y_w \leq 1 \qquad \forall w \in W \tag{5.6}$$

$$x_{j,w} \in \{0, 1\} \qquad \forall j \in J, w \in W \tag{5.7}$$

Smet et al. (2014) proposed a constructive matheuristic method in which the problem is decomposed by assigning each worker to a different minimal subproblem. Therefore, rather than solving the entire problem, they solve subproblems which consist of a subset of workers. A different objective is employed within subproblems: instead of reducing the number of workers with tasks assigned, subproblems maximize the sum of allocated task durations. After a solution is constructed, the last subproblem is re-optimized using the original objective function to reduce the number of workers with assigned tasks.

The constructive approach introduced by Smet et al. (2014) is generalized by the decomposition-based framework proposed in this chapter, being therefore possible to employ the framework to address the SMPTSP. The framework is utilized as a solver, with no alterations in its original source code.

Three input files were generated: (*i*) problem description, (*ii*) decomposition and (*iii*) subproblem characteristics. The problem description is given by Formulation (5.3)-(5.7). The decomposition is easily described: variables corresponding to one worker constitute a minimal subproblem. The only remaining component concerns the subproblem characteristics. It is necessary to rewrite the objective function and relax Constraints (5.4), as otherwise subproblems would be infeasible. This is easily achieved by including the incomplete MIP formulation specified by Equations (5.8)-(5.10). The objective function is overwritten by Equation (5.8) while Constraints (5.4) are replaced by Constraints (5.9). Note how Equations (5.8)-(5.10) introduce variables α_j to identify whether task j is executed ($\alpha_j = 0$) or not ($\alpha_j = 1$), and a weight ω to guarantee that assigning tasks always has higher priority than minimizing the number of workers. Whenever $d_j > 1$ for all tasks $j \in J$, ω is set to 1. Otherwise, if there is a task j with duration $d_j = 1$, ω is assigned a value of 2. In contrast to Smet et al. (2014), who optimized the last subproblem twice

with different objective functions, employing ω prevented this re-optimization and reduced the number of subproblem optimizations.

Minimize:

$$\sum_{w \in W} y_w + \omega \sum_{j \in J} d_j \alpha_j \tag{5.8}$$

Subject to:

$$\sum_{w \in W : j \in T_w} x_{j,w} + \alpha_j = 1 \quad \forall j \in J \tag{5.9}$$

$$0 \leq \alpha_j \leq 1 \qquad \forall k \in J \tag{5.10}$$

Considering these two MIPs and the decomposition description, the framework acted as a solver and obtained similar results to those reported by Smet et al. (2014). Note that Smet et al. (2014) did not consider any intersection between subproblems, and so we considered $step = \eta$ throughout the experiments. The graph representation of connected subproblems enabled experimenting upon all subproblem configurations evaluated by Smet et al. (2014). Furthermore, the local search implemented within the framework, although different from the one proposed by Smet et al. (2014), also resulted in optimal solutions for all 137 instances from Krishnamoorthy et al. (2012). Detailed results are available together with the source code documentation[4].

5.4 Conclusions and future work

This chapter presented an initial implementation of a general decomposition-based heuristic framework that may be utilized as a general solver. The framework successfully implemented the decomposition-based heuristics presented for the TUP, NRP and PSP in Chapters 2, 3 and 4, respectively. While the PSP required extensive modifications to the source code, the algorithms developed for the TUP and NRP were reproduced by only manipulating input files. The framework was also evaluated for another problem, one which was not studied previously in this manuscript. The SMPTSP was considered, and the constructive matheuristic developed by Smet et al. (2014) was easily implemented by the framework. Creating input files was sufficient for solving

[4]https://github.com/tuliotoffolo/jads

the SMPTSP, meaning no modification of source code was required to reproduce the strong results reported by Smet et al. (2014).

In its current form, the framework already constitutes a very useful tool for quickly investigating different decomposition-based heuristics. It may also be easily extended to support additional algorithms. There is, however, much work to be done in terms of transforming it into a truly general solver.

Future work includes implementing alternative constructive and local search algorithms. Different problem, decomposition and subproblem representations should also be considered. Still, the primary challenge for future research remains automatically detecting a problem's structure. As highlighted in Section 5.1.2, it is currently unclear what the desired problem structure is when considering decomposition-based heuristics. Moreover, the subproblem characteristics should also be automatically detected in such a way that the framework can be re-structured according to Figure 5.3. This introduces many interesting questions concerning future research, as discussed in Section 5.1.2.

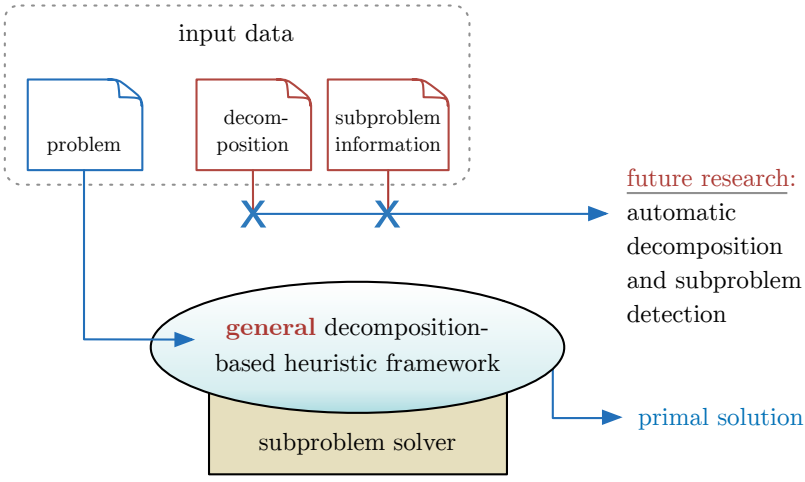


Figure 5.3: Future research directions towards a truly general solver

Finally, in the spirit of reproducible science, the framework's source code is available online[5] together with input files for the TUP, NRP and SMPTPS. A tutorial concerning the framework's utilization is also available. Future developments will also be included within the online repository.

[5]https://github.com/tuliotoffolo/jads

Part II

Heuristic subproblem solutions

Introduction to Part II

Part II alters the tone of the thesis. Rather than approaching different problems with similar decomposition-based strategies, as in Part I, here very specific problem components are explored. The most noticeable difference is, however, related to how subproblems are solved. Whereas Part I followed the trend of optimally solving subproblems, here suboptimal solutions for subproblems are generally accepted, primarily due to computational performance requirements.

Part II addresses logistic problems concerning routing and packing, and represents an adaptation of three papers, two published and one which is currently being prepared for submission. Figures, tables and elements of these texts were adapted to integrate well into the structure and style of this thesis. Nevertheless, each chapter remains self-contained and may be read independently. Figure II illustrates the organizational structure of Part II. In total, three different logistic problems are investigated within Chapters 6, 7 and 8: one classic and two real-world problems.

Chapter 6 studies a decomposition-based local search for the classic *Capacitated Vehicle Routing Problem* (CVRP) in which the different decisions required to solve the problem are exploited. More specifically, two decision sets are in play: *Assignment* and *Sequencing*. The principle behind the decomposition is to guarantee that one of the decision sets (*Sequencing*) is always optimally solved, with a solution therefore represented solely by decisions for the other set (*Assignment*). Nevertheless, a significant computational burden is associated with such an approach. It is in fact shown to be impractical within the proposed local search for the CVRP, resulting in the investigation of an intermediary approach where suboptimal solutions are accepted for *Sequencing*. Different levels of *Sequencing* optimization are studied, resulting in different *search spaces*. Extensive experiments are conducted to evaluate all proposed methods,

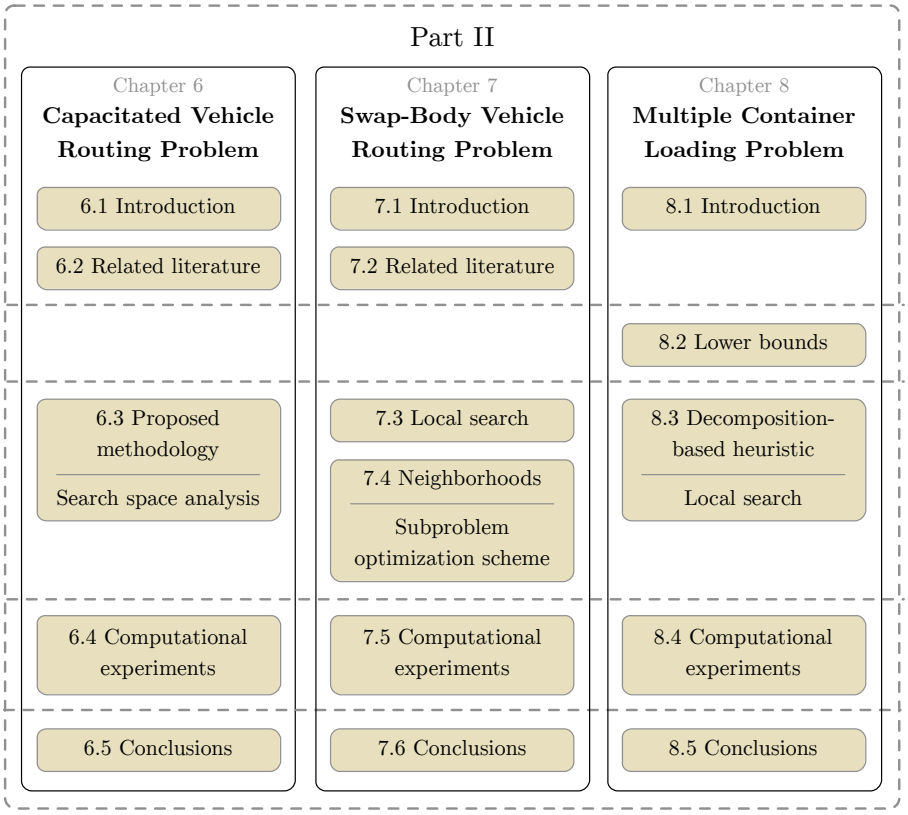


Figure II: Organization of Part II

ultimately resulting in a local search algorithm capable of producing new best known solutions for multiple benchmark instances.

Chapters 7 and 8 employ an alternative methodology. Both chapters investigate real-world problems which were subject of prestigious international optimization algorithm challenges. In fact, the proposed algorithms competed in both challenges, winning one of them and being ranked second in the other one. In both cases, short runtime limits were imposed for the algorithms to produce solutions. Therefore, subproblems were not necessarily solved to optimality. Moreover, with a view to producing high-performing solvers, classical heuristics were hybridized with decomposition-based ones.

Chapter 7 addresses the *Swap-Body Vehicle Routing Problem* (SB-VRP), a generalization of the classical CVRP. Here a subproblem optimization scheme

is employed together with classical and problem-specific neighborhoods within a stochastic metaheuristic local search methodology. The resulting algorithm won the VeRoLog Challenge and remains the state-of-the-art for the problem, generating the best known solutions for the majority of SB-VRP instances.

In Chapter 8, a real-world *Multiple Container Loading Problem* (MCLP) proposed by a car manufacturing company is considered. The MCLP requires the packing of three-dimensional boxes into containers subject to a large set of constraints. The three-dimensional problem is heuristically decomposed into different two-dimensional subproblems. Multiple algorithms are employed to solve the subproblems, including a dynamic programming approach and constructive heuristics. Computational experiments validate the methodology's performance, with it capable of quickly producing many best known solutions.

Chapter 6

Capacitated Vehicle Routing Problem

This chapter addresses the classic *Capacitated Vehicle Routing Problem* (CVRP), an extensively studied optimization problem in which a fleet of vehicles with uniform capacity must be assigned to satisfy customer demands. The objective is to minimize the sum of travel distances over all vehicles, each of which leaves from and must return to a single depot after visiting customers. As with the other problems studied throughout this thesis, benchmark sets are available for the CVRP.

In contrast to the chapters constituting Part I, a different decomposition strategy is investigated here, one based upon the decisions that compose a solution for the problem. Note that a solution for the CVRP is composed of sequential decisions: (*i*) assigning customers to vehicles and (*ii*) sequencing customer visits for each vehicle. These two decision sets, *Assignment* and *Sequencing*, are exploited to decompose the problem. The idea is to ensure that optimum *Sequencing* decisions are taken at all times, essentially reducing the problem to the *Assignment* decisions. In this chapter we investigate this decomposition strategy and its consequences. Local search is applied to explore different *Assignment* solutions, therefore only considering the search space of assignments. However, experimentation revealed that the computational burden associated with optimally solving the *Sequencing* decisions is very costly, thereby justifying the investigation of intermediary approaches in which *Sequencing* is

only partially solved. We employ the neighborhood proposed by Balas and Simonetti (2001) together with smart computational techniques to represent such an intermediary approach. We show how this combination results in a search space that ultimately converges to one in which only *Assignment* decisions are considered.

This chapter is a joint work with Thibaut Vidal[1]. It begins with Section 6.1 introducing the main ideas and motivation behind the present research. Related literature is presented in Section 6.2, further justifying the proposed approach. The methodology is detailed in Section 6.3, which presents the primary principles associated with the proposed approach together with the computational techniques employed to increase the local search efficiency. Experiments presented in Section 6.4 evaluate the methodology. Different computational techniques and parameters are examined. Additionally, the proposed local search is evaluated within a state-of-the-art metaheuristic, resulting in unexpected improvements over the best known solutions for both small- and medium-size instances. Finally, Section 6.5 ends this chapter by presenting conclusions and future research directions.

6.1 Introduction

The Capacitated Vehicle Routing Problem (CVRP) is classically described as a combination of a Traveling Salesman Problem (TSP) with an additional capacity constraint which lends a bin packing substructure to the problem (Toth and Vigo, 2014). It may also be viewed as a set packing problem in which the cost of each set corresponds to the distance of the associated optimum TSP tour (Balinski and Quandt, 1964). These problem representations emphasize the two decision sets at play: customer-to-vehicle *Assignment* and *Sequencing* choices for each route.

Examining the recent progress on metaheuristics for the CVRP, little has changed in recent years concerning intra-route neighborhood search: *Relocate*, *Swap* and *2-opt* neighborhoods and their immediate generalizations are employed, and these neighborhoods alone are sufficient to guarantee that most CVRP solutions resulting from a local search contain TSP-optimal routes. This is generally because classic CVRP instances involve short routes with up to 20 or

[1]Prof. Dr. Thibaut Vidal (vidalt@inf.puc-rio.br), Informatics Department, Pontifical Catholic University of Rio de Janeiro, Brazil

30 visits. For such small problems, even simple neighborhood search methods for the TSP tend to produce optimum tours.

Based on this observation, a larger effort dedicated to TSP tour optimization, as a stand-alone neighborhood, is unlikely to result in further improvements. For this reason, it is very uncommon to observe the use of larger *intra-route* neighborhoods such as *3-Opt* in state-of-the-art (meta)heuristics. Nevertheless, does this mean that *Sequencing* optimization should be abandoned in favor of more extensive search concerning *Assignment* choices? Certainly not. Indeed, even if local minima exhibit TSP-optimal tours, inter-route moves frequently lead to TSP-suboptimal tours which are rejected due to their higher cost, but would otherwise be accepted if such tours were optimized. Such solution improvements would then not arise from separate *Assignment* or *Sequencing* optimizations, but from a careful combination of both.

Figure 6.1 schematically represents the solution set of the CVRP, whose decision variables are split into *Sequencing* decisions (x-axis) and *Assignment* decisions (y-axis). The y-axis also represents solutions in terms of their *Assignment* decisions solely, ignoring *Sequencing* choices. These solutions may be viewed as a projection (Geoffrion, 1970) of the original solutions $\mathcal{S}$ on the space $\mathcal{S}^{\mathrm{A}}$ defined by a single decision subset (*Assignment*). Moreover, from a solution represented in terms of *Assignment* decisions, it is possible to find the best associated complete solution by solving each TSP associated with the routes. This process corresponds to a decoder.

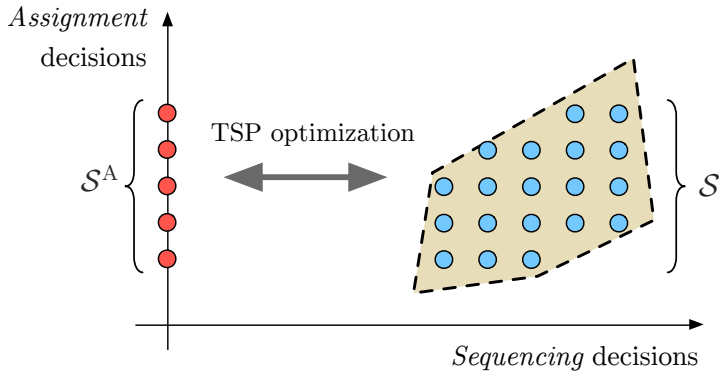


Figure 6.1: Two alternative search spaces for the CVRP

With this picture in mind, it is tempting to conduct a search in space $\mathcal{S}^{\mathrm{A}}$ rather

than $\mathcal{S}$. After all, the size of $\mathcal{S}^{\mathrm{A}}$ is exponentially smaller than that of $\mathcal{S}$ and the solutions of $\mathcal{S}^{\mathrm{A}}$ are, on average, of better quality since they have optimal tours. However, the obvious drawback is that each move evaluation in $\mathcal{S}^{\mathrm{A}}$ requires solving one or several small TSPs to optimality and thus requires significant computational effort. Still, note that considerable progress has been made in the past 30 years with regard to the efficient resolution of TSPs, and small problems with approximately 20 customers are now solvable within a few milliseconds. Based on these observations, this chapter investigates heuristic search methods for the CVRP in terms of answering two questions concerning the decision-set decomposition which results in the search space $\mathcal{S}^{\mathrm{A}}$:

1. Is it practical and worthwhile to search in space $\mathcal{S}^{\mathrm{A}}$ rather than $\mathcal{S}$?
2. If searching in $\mathcal{S}^{\mathrm{A}}$ requires an excessive effort, is it possible to define a search space which maintains most of the key properties of $\mathcal{S}^{\mathrm{A}}$ while enabling more efficient exploration?

As will be demonstrated later in Section 6.4, experiments led us to answer the first question as follows: even with non-trivial memory and speedup techniques (hashtables and move filters) the computational overhead related to the exact resolution of TSPs during each move evaluation does not appear worth the gain in terms of solution quality for a complete search in space $\mathcal{S}^{\mathrm{A}}$.

By contrast, our answer to the second question is positive. Rather than requiring a complete exact resolution for each TSP, the dynamic programming approach of Balas and Simonetti (2001), hereafter referred to as B&S, may be employed to perform a restricted optimization of routes during local search move evaluations. Given a range parameter k and an initial tour, the B&S approach is capable of producing the vertex sequence with minimum cost such that no vertex is displaced by more than k positions. This defines a search space $\mathcal{S}_k^{\mathrm{B}}$ such that $\mathcal{S}_0^{\mathrm{B}} = \mathcal{S}$ and $\mathcal{S}_{n-1}^{\mathrm{B}} = \mathcal{S}^{\mathrm{A}}$, with n representing the total number of customers. Moreover, even for a fixed k, *tunneling* techniques are proposed which enable the exploitation of past solutions to dynamically reshape the search space, in such a way that $\mathcal{S}_k^{\mathrm{B}}$ converges towards $\mathcal{S}^{\mathrm{A}}$ as the search progresses.

6.2 Related literature

This section reviews some key milestones concerning the management of *Sequencing* and *Assignment* decisions in vehicle routing heuristics.

Sequencing and *Assignment* decisions were first optimized separately in early constructive heuristics, giving rise to different families of methods. Route-first cluster-second algorithms (Bodin and Berman, 1979, Beasley, 1983) first produce a giant TSP tour before subsequently assigning sequences of consecutive visits into separate trips to produce a complete solution. In cluster-first route-second methods (Fisher and Jaikumar, 1981), a clustering algorithm is employed to group customer visits into clusters, followed by TSP optimizations. Finally, *petal algorithms* (Foster and Ryan, 1976, Renaud et al., 1996) are based on an a-priori generation of candidate routes (petals), followed by the resolution of a set packing or covering problem.

In the development of local search and metaheuristic algorithms which ensued in the 1990s and thereafter, *Assignment* and *Sequencing* optimizations began to be better integrated via classical neighborhood local search (*Relocate*, *Swap*, *2-opt*, *2-opt**) which optimize both decision subsets. These neighborhood search methods form the basis of the vast majority of state-of-the-art algorithms. Petal algorithms have withstood the test of time and high-quality routes are now extracted from metaheuristic local minima instead of being listed in advance (see Muter et al., 2010, Subramanian et al., 2013b).

The variety of vehicle routing problem variant has also triggered studies concerning problem decompositions. Vidal et al. (2013) established a review of the classical variants and their associated constraints, objectives, and decision sets, called *attributes*. These attributes were classified in relation to their impact on *Sequencing* and *Assignment* decisions and on *Route evaluations* within heuristics, a classification which forms the basis of the state-of-the-art results for dozens of VRP variants. Many known problem attributes come jointly with new decision subsets when optimizing vehicle routing with packing, timing or scheduling constraints (Goel and Vidal, 2014, Pollaris et al., 2015), visit choices (Vidal et al., 2016) or service-mode choices (Vidal et al., 2015). Decision-set decompositions are employed throughout many of the aforementioned papers to perform a search in the space of *Sequencing* and *Assignment* choices and optimally determine the remaining decision variables during each route and move evaluation.

In this chapter, the decision-set decomposition does not result from supplementary problem attributes, but is instead used to define exponential-size polynomially-searchable neighborhoods and transform the search space. Exponential-size neighborhoods have a long history in the combinatorial optimization literature (Deineko and Woeginger, 2000, Ahuja et al., 2002,

Bompadre, 2012). Most of these neighborhoods are based on shortest path or matching subproblems, as well as specific graph and distance matrix structures with which some $\mathcal{NP}$-hard problems become tractable (consider, as examples, Halin graphs or Monge matrices). As a rule of thumb, larger neighborhoods and faster search procedures are generally desirable. There are, however, theoretical limitations to the size of polynomially-searchable neighborhoods. Gutin and Yeo (2003) proved that, for the TSP, no neighborhood of cardinality at least $(n-k)!$ for a given constant k may be searched in polynomial time unless NP $\subset$ P\poly.

The neighborhood of Balas and Simonetti (2001) (B&S) is one such exponential-size neighborhood for the TSP. Given a tour represented as a permutation σ of n customers and a value k, it contains all permutations π of σ such that π fulfills $\pi(1) = 1$ and $\pi(i) \leq \pi(j)$ for all $i, j \in \{1, \ldots, n\}$ such that $i + k \leq j$. In other words, if i precedes j by more than k positions in σ, then $\pi(i)$ precedes $\pi(j)$. This neighborhood contains $2^{\Theta(n)}$ solutions and may be explored in $\mathcal{O}(k^2 2^{k-2} n)$ time using dynamic programming. This is a linear time complexity when k is constant and a polynomial time complexity when $k = \mathcal{O}(\log n)$. Balas and Simonetti (2001) performed extensive experiments and demonstrated that this dynamic programming procedure may be used as a stand-alone neighborhood to improve high-quality local minima of the TSP and its immediate variants. Later, Irnich (2008), Hintsch and Irnich (2017), and Gschwind and Drexl (2016) employed this neighborhood to solve arc-routing problems with possible cluster constraints and dial-a-ride problems. One common characteristic of these studies is that they employed B&S as a stand-alone neighborhood for route improvements. Only one conference presentation (Irnich, 2013) highlighted the possibility of employing the B&S neighborhood together with some classical CVRP moves, but the performance of such an approach remained largely unexplored.

We seek to go one step further. Rather than applying this tour optimization procedure in combination with a single neighborhood, we investigate employing it in a *systematic* manner in combination with every move (*Swap* or *Relocate*, for example) within a CVRP local search. This leads to a redefinition of the search space, which is discussed in the following section.

6.3 Proposed Methodology

The methodology is presented as a local search algorithm, which can later be easily incorporated into any metaheuristic for the CVRP. The proposed techniques are better described when considering indirect solution representations. There is, however, no widely accepted term in the literature for referring to the elements which represent such indirect solutions. To circumvent this issue, we henceforth employ the term *primitive solutions*. Before proceeding to describing the methodology, some basic definitions related to neighborhood local search and indirect solution representations are recalled:

Definition 6.1. *Primitive solutions and search space*: consider a combinatorial optimization problem of the form $\min_{s \in S} f(s)$, where S is the solution space and f the objective function to minimize. Let S^{p} be the set of primitive solutions and let the decoder $z : S^{\mathrm{p}} \to S$ be an injective application that transforms any primitive solution $s' \in S^{\mathrm{p}}$ into a complete solution $s \in S$. A neighborhood is defined as a mapping which associates with each primitive solution s' a set of neighbors $\mathcal{N}(s') \subset S^{\mathrm{p}}$. A *move* is an operation that transform a primitive solution s' into one of its neighbors in $N(s')$. The graph induced by S^{p} and $\mathcal{N}$ is referred to as the search space, in which nodes represent solutions and edges indicate moves.

Definition 6.2. *TSP-optimal tour*: a tour σ is TSP-optimal if there exists no permutation π of σ with a shorter total distance.

Definition 6.3. *B^k-optimal tour*: a tour σ is B^k-optimal if there exists no permutation π of σ with smaller cost such that $\pi(i) \leq \pi(j)$ for all $i, j \in \{1, \ldots, n\}$ with $i + k \leq j$.

Throughout the following sections, we propose a choice of search space and introduce techniques for efficiently exploring it.

6.3.1 Search spaces

This section examines the search spaces associated with the set of all solutions, of those with TSP-optimal tours, and of those with B^k-optimal tours. Afterwards, we present a discussion justifying the search space choice. Finally, the general local search algorithm is presented.

Search space $\mathcal{S}$

Classical local search methods for the CVRP do not distinguish between primitive and complete solutions. In the search space $\mathcal{S}$, solution sets S and S^{p} are equal and the decoder z corresponds to the identity function.

Figure 6.2 presents the search space $\mathcal{S}$ associated with *Relocate* moves for a small asymmetric CVRP with three customers. There are 13 possible solutions for this problem, each represented by a set of ordered customer visits. Each vehicle is responsible for one ordered set. Solution '[1,2,3]', for example, employs one vehicle to visit customers 1, 2 and 3, while solution '[1][2][3]' employs three vehicles, one per customer. Each node in the graph indicates a solution, positioned on the x-axis according to its solution quality (the more to the right, the better a solution is). Outgoing arcs represent moves which transform the solution into one of its neighbors. Moreover, solutions with identical customer-route assignments are grouped within dashed areas. Note that, for this instance size, it is always possible to reach the optimum solution from any starting point in two successive moves. In a local search that explores the neighborhood in random order and applies an improving move as soon as it is found, the worst case corresponds to five moves (when the initial solution is '[2][1,3]' or '[1][2][3]').

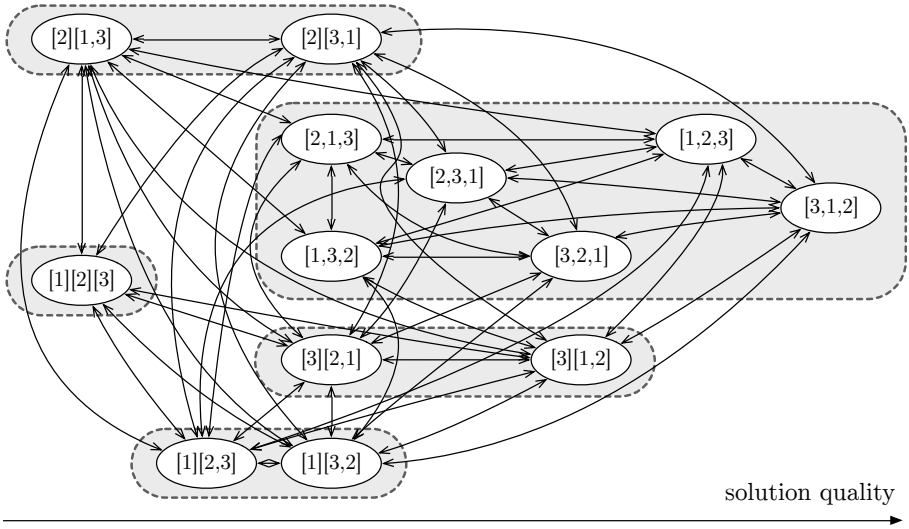


Figure 6.2: Search space $\mathcal{S}$ for a small asymmetric CVRP instance

Search space $\mathcal{S}^{\mathrm{A}}$

As discussed earlier in this chapter, CVRP solutions may be represented in terms of their *Assignment* decisions, excluding the *Sequencing* decisions in the representation and delegating the choices of the best visit sequences to a decoder. With such a paradigm, one may define a local search in the space S^{p} of primitive solutions, where each $s' \in S^{\mathrm{p}}$ represents a partition of the customer set into subsets whose sums of demands do not exceed the vehicle capacity. The decoder z is based on an exact TSP solver, responsible for generating the best visit sequence originating from and finishing at the depot for each subset of customers. In this sense, the image $z[S^{\mathrm{p}}] \subset S$ exclusively contains solutions with TSP-optimal tours.

The neighborhood used to explore search space $\mathcal{S}^{\mathrm{A}}$ can remain similar to classical CVRP neighborhoods, based on relocations or exchanges of customers between subsets, or involve other families of moves specialized for partition problems. Figure 6.3 represents the resulting search space with simple *Relocate* moves. Only TSP-optimal tours are explored and therefore the size of the search space reduces down to five solutions. The other solutions and their connections are represented in light gray. Note, in our small example, that now at most three successive improving moves may be applied to attain the optimum from '[1][2][3]'.

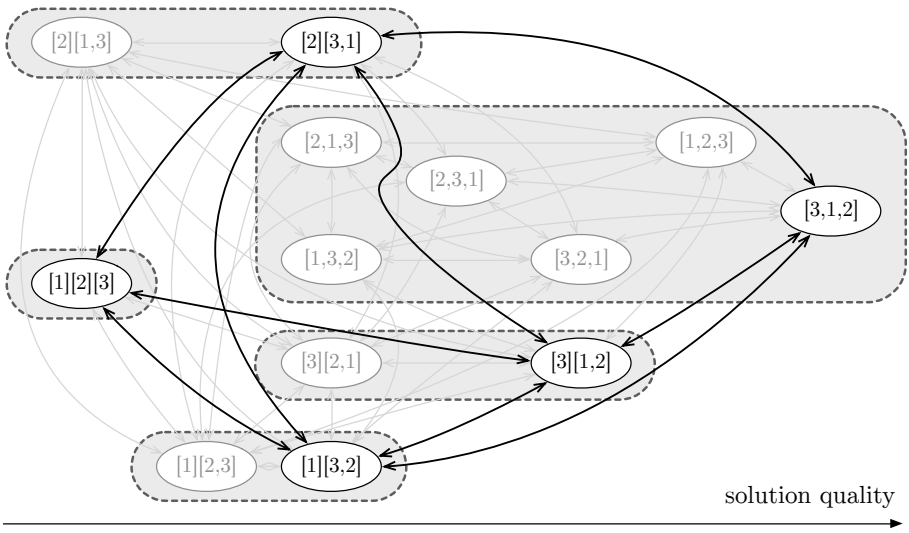


Figure 6.3: Search space $\mathcal{S}^{\mathrm{A}}$ for a small asymmetric CVRP instance

Search space $\mathcal{S}^{\mathrm{A}}$ is much smaller than $\mathcal{S}$ and our computational experiments (Section 6.4) demonstrate that a search in this space indeed leads to solutions with higher quality. However, each move evaluation in this space requires executing an algorithm with exponential worst-case time complexity, a TSP solver, in order to decode each primitive solution in the neighborhood for cost evaluation. Although research on the TSP has culminated in very efficient algorithms over the past thirty years, thousands (or millions) of small TSP instances should be solved during a local search in $\mathcal{S}^{\mathrm{A}}$, and thus the total computational effort dedicated towards decoding can grow prohibitively large. Moreover, bad behavior in a single case is sufficient, without any other safeguard, to stall the entire algorithm.

Search space $\mathcal{S}_k^{\mathrm{B}}$

To circumvent the aforementioned issues, we study an alternative search space in which set S^{p} contains complete solutions (with their *Assignment* and *Sequencing* decisions), but where the decoder z is nontrivial and consists of applying B&S multiple times to each route with a fixed k value until all tours are B^k-optimal. With these assumptions, the image $z[S^{\mathrm{p}}]$ contains exclusively complete solutions with B^k-optimal tours. Since the primitive solutions form a subset of the complete solutions, the application of B&S may also be viewed as a post-optimization step *during* classical CVRP move evaluations, opening the way for additional solution improvements. A careful analysis of the resulting search space gives even more significance to this approach, due to three properties:

Property 6.1. From an initial solution containing a B^k-optimal tour, a local search in space $\mathcal{S}_k^{\mathrm{B}}$ explores only B^k-optimal tours.

Property 6.2. For a fixed k, the computational effort of decoding a route grows polynomially with the number of customers and number of calls to B&S.

Property 6.3. The search space $\mathcal{S}_k^{\mathrm{B}}$ is such that $\mathcal{S}_0^{\mathrm{B}} = \mathcal{S}$ and $\mathcal{S}_{n-1}^{\mathrm{B}} = \mathcal{S}^{\mathrm{A}}$, with n being the total number of customers.

These three simple properties are all fundamental for the methodology that follows. Property 6.1 demonstrates how space $\mathcal{S}_k^{\mathrm{B}}$ contains fewer solutions than $\mathcal{S}$ and that the overall quality of these solutions tends to be higher (since non-B^k-optimal tours are filtered out). Moreover, Property 6.2 offers some computational time guarantees: even if the computational effort grows

quickly with k, the decoder runtime is guaranteed to remain stable when k is constant, eliminating the possibility of a computational effort *peak* for specific TSP instances. Finally, Property 6.3 demonstrates how k balances the effort dedicated towards the optimization of the *Assignment* and *Sequencing* decision sets, and establishes $\mathcal{S}_k^{\mathrm{B}}$ as an intermediate search space generalizing $\mathcal{S}$ and $\mathcal{S}^{\mathrm{A}}$.

Figure 6.4 illustrates the search space $\mathcal{S}_k^{\mathrm{B}}$ for the same example as previous figures when $k = 1$. It is an intermediate space between those depicted by Figures 6.2 and 6.3 which correspond to $\mathcal{S}_0^{\mathrm{B}} = \mathcal{S}$ and $\mathcal{S}_2^{\mathrm{B}} = \mathcal{S}^{\mathrm{A}}$, respectively. Note how solution $[1, 2, 3]$ is now a neighbor of $[2][3, 1]$ and $[1][3, 2]$, as indicated by the two dotted arcs.

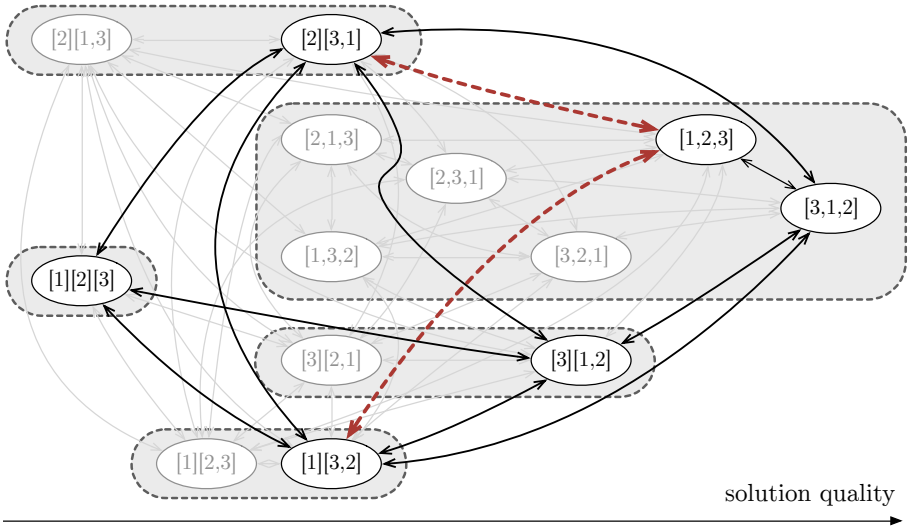


Figure 6.4: Search space $\mathcal{S}_1^{\mathrm{B}}$ for a small asymmetric CVRP instance

Discussions and search space choice

In light of these observations, we have conducted computational experiments on search space $\mathcal{S}_k^{\mathrm{B}}$ employing the dynamic programming algorithm of Balas and Simonetti (2001) to decode each solution, as well as on search space $\mathcal{S}^{\mathrm{A}}$ using the TSP solver *Concorde* (Applegate et al., 2003). Despite several speedup techniques (Section 6.3.2), the search in space $\mathcal{S}^{\mathrm{A}}$ remained inefficient throughout our experiments, especially for instances with large average route cardinality (number of customers in the route). We therefore decided to focus

on search space $\mathcal{S}_k^{\mathrm{B}}$ and devised several speedup techniques to enable its efficient exploration.

6.3.2 Efficient exploration strategies

To efficiently explore space $\mathcal{S}_k^{\mathrm{B}}$, we developed a local search algorithm employing multiple speedup techniques: *neighborhood size reduction*, *dynamic move filters*, *concatenation techniques* and *efficient memory structures*. Most of these techniques seek to limit the search effort in $\mathcal{S}_k^{\mathrm{B}}$.

Neighborhood size reduction

The majority of recent local search based metaheuristics for the CVRP limits the neighborhood $\mathcal{N}(s)$ of an incumbent solution s to solutions generated by moves which involve vertices close to each other. Based on concepts described by Johnson and McGeoch (1997) and Toth and Vigo (2003), we restrict the search to a subset of moves that reconnect at least one vertex i with a vertex j belonging to the Γ closest vertices of i. The procedure requires a straightforward preprocessing step. The neighborhood size becomes $\mathcal{O}(\Gamma n)$, enabling a significant speedup for large-scale problem instances.

Dynamic move filters

To further restrict the search to promising moves, each move ϕ's feasibility is evaluated in $\mathcal{O}(1)$, in terms of capacity constraints, being discarded if it leads to an infeasible solution. The total cost $f(\phi(s))$ of the solution generated by ϕ prior to its optimization by the B&S decoder is evaluated subsequently. This cost represents an upper bound for the final cost of the move in $\mathcal{S}_k^{\mathrm{B}}$ after the application of the decoder. The move evaluation is pursued only if the solution cost has increased by a factor $1 + \psi$ or less due to its application, that is, only if Equation (6.1) is satisfied. Otherwise, the move is discarded.

$$f(\phi(s)) \leq (1 + \psi) \times f(s) \tag{6.1}$$

Parameter ψ plays an important role in defining how many moves are evaluated. The higher the value of ψ, the less pruning is induced by Equation 6.1.

Contrastingly, when $\psi = 0$ only immediately improving neighbors are evaluated. Defining a good value for ψ is not trivial, given it is an instance-dependent parameter. Since a fixed value would not suit instances with different sizes and characteristics, we propose an adaptive parameter. The principle consists of adjusting ψ to ensure a target range $[\xi^-, \xi^+]$ for the fraction of filtered moves. After every 1,000 move evaluations, the fraction ξ of filtered moves is collected and whenever it falls below or surpasses the desired range, ψ is updated. If this fraction is too large, then ψ is increased by a multiplicative factor α. In contrast, if ξ is insufficient, then the parameter ψ is decreased:

$$\psi = \begin{cases} \psi \times \alpha & \text{if } \xi < \xi^-, \\ \psi \ / \ \alpha & \text{if } \xi > \xi^+, \\ \psi & \text{otherwise.} \end{cases} \tag{6.2}$$

Experiments concerning the impact of different target ranges are discussed in Section 6.4.

Memory structures

The B&S algorithm, employed as a decoder, requires a computational effort which grows linearly with route cardinality and exponentially with parameter k. It is therefore essential to restrain the use of this procedure to a strict minimum and avoid decoding twice the same route over the course of the search. To that end, we employ a memoization scheme (Michie, 1968) to cache routes that have been decoded. A hashtable is used to implement the cache since it allows $\mathcal{O}(1)$ queries given the *hash* value(s) associated with a route.

Two important aspects should be discussed. First, the number of possible routes grows exponentially with instance size, whereas the available memory space is finite. Therefore, some strategy is necessary to limit memory consumption. To that end, only decoded routes are cached, with the original routes being identified by their associated hash values. Moreover, an upper bound $\mathcal{M}^{\max}$ on the memory usage is defined and each route in the cache is associated with an utilization counter which is incremented whenever the entry is queried. This enables removing the least-utilized entries if memory usage reaches the upper bound. Whenever the bound is reached, a cleaning procedure is executed in linear time: first the median of entries utilization is obtained and then up to 50% of the elements whose utilization is below or equal to the median are removed. Next, the utilization counter of all non-deleted entries is reset to zero.

The second aspect to be discussed concerns the effort spent querying the memory. Querying memory for a given route supposes the availability of hash values which characterize the associated sequence of visits, but a direct approach that sweeps through the route to compute this index already takes $\mathcal{O}(n)$ time. To avoid this bottleneck, specific hash functions and calculation techniques based on concatenations in $\mathcal{O}(1)$ are employed. The details of these concepts are discussed throughout Section 6.3.3.

General local search algorithm

The local search algorithm including all discussed components is presented in Algorithm 6.1. The algorithm begins by searching in the neighborhood defined by $\mathcal{N}(s)$ involving vertex pairs (i, j) with $j \in \Gamma(i)$ (lines 1-2). The routes modified by the move are identified (lines 3-4) and the dynamic filters are then applied (lines 5-8). If the move is considered promising, it is evaluated. First the cost c is initialized (line 9). Then, each route is decoded (lines 10-16) and the cost is included in c (line 17). Note that the cache is employed during the decoding process (lines 11-12). Moreover, the B&S algorithm may be applied multiple times, until a B^k-optimal tour is obtained. It is in fact applied at least once to verify that the solution is B^k-optimal. If the evaluated move results in an improving solution, it is applied (lines 18-20). The procedure repeats until a local minimum of $\mathcal{S}_k^{\mathrm{B}}$ is reached (line 21), which is immediately returned (line 22).

Algorithm 6.1: Efficient local search in space $\mathcal{S}_k^{\mathrm{B}}$

Input: An initial complete solution s, dynamic move filter parameter ψ and neighborhood size reduction parameter Γ

LocalSearch(s, ψ, Γ)

repeat
  // enumerate $\mathcal{O}(\Gamma n)$ moves - candidate lists based on vertex proximity
  for each move $\phi(s) \in \mathcal{N}(s)$ involving vertex pair (i, j), $j \in \Gamma(i)$ **do**
    ϕ modifies at most two routes of s: let σ^1 and σ^2 be these routes
    let c_0 be the sum of the cost of routes σ^1 and σ^2 in s
    // filter infeasible moves with respect to capacity in $\mathcal{O}(1)$
    if σ^1 or σ^2 are infeasible with respect to capacity **then**
      continue
    // filter non-promissory moves in $\mathcal{O}(1)$
    if $f(s) + f(\sigma^1) + f(\sigma^2) - c_0 > (1 + \psi) \times f(s)$ **then**
      continue
    // finally, *decode* routes σ^1 and σ^2 to evaluate move ϕ in $\mathcal{S}_k^{\mathrm{B}}$
    $c \leftarrow 0$
    for each route σ^i with $i \in \{1, 2\}$ **do**
      // compute hash key and check cache in $\mathcal{O}(1)$:
      $(\bar{\sigma}^i, \bar{c}_i) \leftarrow LookUp(\sigma_1^i)$
      // if route not cached, apply B&S dynamic programming to decode it
      if $(\bar{\sigma}^i, \bar{c}_i)$ not found **then**
        $(\bar{\sigma}^i, \bar{c}_i) \leftarrow$ BalasSimonetti(σ^i)
        while $\bar{\sigma}^i$ is not a B^k-optimal tour **do**
          $(\bar{\sigma}^i, \bar{c}_i) \leftarrow$ BalasSimonetti($\bar{\sigma}^i$)
        include $(\sigma^i \rightarrow (\bar{\sigma}^i, \bar{c}^i))$ in the cache
      $c \leftarrow c + \bar{c}_i$
    // if move is an improving one in $\mathcal{S}_k^{\mathrm{B}}$, it is applied:
    if $c \leq c_0$ **then**
      $s \leftarrow \phi(s)$
      replace routes (σ^1, σ^2) by $(\bar{\sigma}^1, \bar{\sigma}^2)$ in s
until s is a local minimum
return s

6.3.3 Constant-time evaluation

The concatenation strategy of Vidal et al. (2016) is employed to perform efficient cost and load-feasibility evaluation. This strategy exploits the fact that any route obtained from a classical move $\phi(s)$ on a solution s corresponds to a

recombination of a bounded number of visit sequences of s. New routes may be expressed as a concatenation of sequences $\sigma^1 \oplus \cdots \oplus \sigma^b$. We extend this approach to enable constant time computation of hash values.

To efficiently evaluate the cost, load, and other attributes of new routes, we rely on preliminary preprocessing these attributes on the $\mathcal{O}(n^2)$ subsequences of consecutive visits which compose a solution s. Four values are calculated: the total demand $Q(\sigma)$ of a sequence σ, its distance $C(\sigma)$, and *hash values* $H^p(\sigma)$ and $H^s(\sigma)$. For a sequence $\bar{\sigma} = [v]$ containing a single customer v with demand q_v, $Q(\bar{\sigma}) = q_v$, $C(\bar{\sigma}) = 0$, $H^p(\bar{\sigma}) = \rho \times v$ and $H^s(\bar{\sigma}) = \rho^v$, where ρ represents a prime number. Moreover, Equations (6.3)-(6.5) extend these quantities, by induction, for any sequence of customers $\sigma^1 \oplus \sigma^2$ expressed as the concatenation of two sequences σ^1 and σ^2. In these equations $d_{v,w}$ indicates the distance between customers v and w.

$$Q(\sigma^1 \oplus \sigma^2) = Q(\sigma^1) + Q(\sigma^2) \tag{6.3}$$

$$C(\sigma^1 \oplus \sigma^2) = C(\sigma^1) + d_{\sigma^1_{|\sigma^2|},\sigma^2_1} + C(\sigma^2) \tag{6.4}$$

$$H^p(\sigma^1 \oplus \sigma^2) = H^p(\sigma^1) + \rho^{|\sigma^1|} \times H^p(\sigma^2) \tag{6.5}$$

$$H^s(\sigma^1 \oplus \sigma^2) = H^s(\sigma^1) + H^s(\sigma^2) \tag{6.6}$$

Two hash functions are employed, as a means of reducing chances of two distinct sequences having identical hashes. Their inductive definitions are presented by Equations (6.5) and (6.6). The first is a multiplicative hash function H^p which considers the permutation of visits. Note that when implementing such a function, it is important to pre-compute and bound values for ρ^ℓ ($\ell \in \mathbb{Z}^+, \ell \leq n$) in addition to also bounding hash values so as to prevent overflow during multiplication. The second is an additive hash function H^s which ignores the sequence and therefore solely considers the set of visited customers. Functions H^p and H^s are also defined by Equations (6.7) and (6.8), respectively.

$$H^p(\sigma) = \sum_{i=1}^{|\sigma|} \rho^i \times \sigma_i \tag{6.7}$$

$$H^s(\sigma) = \sum_{i=1}^{|\sigma|} \rho^{\sigma_i} \tag{6.8}$$

These hash functions fit our purposes particularly well due to the aforementioned

inductive definition, which offers a means of computing a route's hash values in $\mathcal{O}(1)$ during move evaluations. To further reduce the risk of two routes having similar hashes, the hash functions are applied twice employing different values for ρ, resulting in four hash values. For the first two values, ρ is set to the smallest prime number greater than n (number of customers) while for the third and fourth hash values ρ is assigned a value of 31 (multiplier used by Kernighan and Ritchie, 1988). These hash values serve, along with the distance of the route and its number of visits, to correctly distinguish a route in $\mathcal{O}(1)$. Note that even with this strategy, a tiny chance of false positives remains. To prevent issues and wrong solutions, the sequences are double-checked by an $\mathcal{O}(n)$ procedure whenever an improving move is accepted. However, no false positive was registered within our computational experiments considering multiple runs on 100 different instances.

Equations (6.3)-(6.6) may be employed iteratively, in lexicographic order, to obtain information concerning all sequences during the preprocessing phase. Afterwards, the same equations are used for move evaluations. Since any route obtained from a classical move corresponds to the concatenation of a bounded number of sequences, it is possible to obtain the associated load, distance, and hash values by solving these equations a limited number of times. The information concerning subsequences is updated every time an improving move is applied. Although this takes $\mathcal{O}(n^2)$ time in the worst case, it remains an efficient approach in practice since the proportion of accepted (improving) moves is very low and at most two routes are impacted by a move.

6.3.4 Using memory to reshape the search space

The previous section introduced memory structures to limit the search effort in $\mathcal{S}_k^{\mathrm{B}}$, including a cache structure that implements a memoization scheme. Here we propose an alternative cache strategy which goes one step further: it not only avoids decoding a route twice but also reshapes the search space $\mathcal{S}_k^{\mathrm{B}}$.

In the cache structure previously presented, henceforth referred to as *permutation cache*, the decoded sequence obtained from each route is stored in the cache. Therefore, even if two sequences σ^1 and σ^2 contain the same customers permuted in different orders, two entries will be stored in the cache. This approach has two significant disadvantages. First, every customer permutation stores a sequence in the cache, leading to additional memory requirements. Second, improving

sequences for a customer set may be ignored. To circumvent this issue, an alternative cache strategy is proposed, hereafter called *set cache*.

In contrast to the *permutation cache*, within the *set cache* strategy only the best tour obtained for a customer set is stored independently of the permutation. Note that if search space $\mathcal{S}^{\mathrm{A}}$ was considered, only optimal tours would be stored in the cache and, therefore, it would be enough to store one tour per customer set. When producing B^k-optimal sequences with B&S, however, it is important to ensure each queried permutation is decoded (optimized) at least once, as otherwise possibly-improving sequences would not be considered. This requires storing the best sequence for a given customer set and a reference (hash values) for permutations previously evaluated for this set of customers. Figure 6.5 presents a flowchart detailing both the *permutation* and *set cache* strategies for storing B^k-optimal tours.

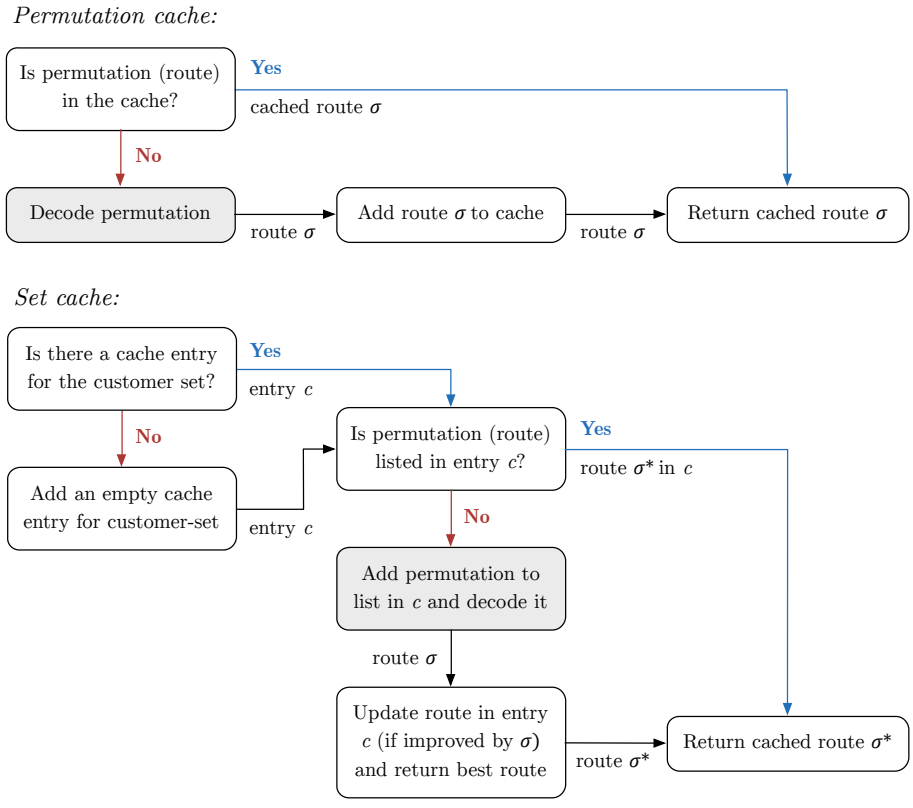


Figure 6.5: *Permutation* and *set cache* strategies

The *set cache* strategy strongly influences the search space. In fact, it modifies the search space, causing a *tunneling* effect. Note that, as the search progresses, certain solutions are cached causing those with similar customers but inferior quality to be removed from the search space. In the hypothetical situation where all permutations have already been decoded (hypothetical due to the exponential memory size and computational time required), the *set cache* systematically returns the TSP-optimal tour for each route and the algorithm behaves as though it were searching in $\mathcal{S}^{\mathrm{A}}$. With this situation in mind, one may informally state that the tunneling strategy contributes towards reshaping the search space $\mathcal{S}_k^{\mathrm{B}}$ into a space more and more similar to $\mathcal{S}^{\mathrm{A}}$ as the search progresses.

Take for instance the small example depicted in Section 6.3.1 (Figures 6.2–6.4). Figure 6.6 illustrates the effect of including solution '$[3,1,2]$' in the cache when the *set cache* strategy is employed. It causes solution '$[1,2,3]$' to be removed, with its input being redirected to '$[3,1,2]$'. Therefore, despite being B^k-optimal, solution '$[1,2,3]$' is replaced by '$[3,1,2]$', which has identical assignments but a lower cost due to better *Sequencing* decisions. Figure 6.6 also demonstrates how the search space converges towards $\mathcal{S}^{\mathrm{A}}$ as the search progresses. In fact, the resulting search space in this example (Figure 6.6) is already equivalent to $\mathcal{S}^{\mathrm{A}}$, presented previously within Figure 6.3.

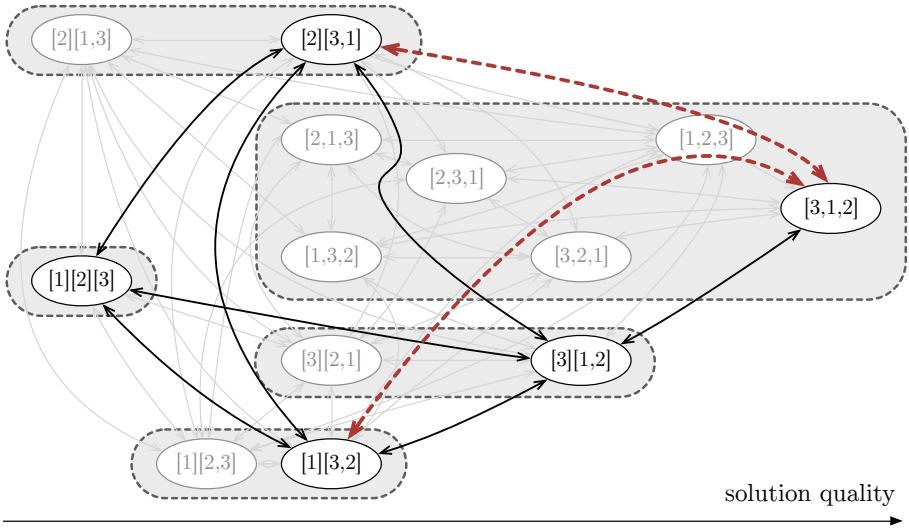


Figure 6.6: Dynamic reshaping (*tunneling*) of the search space

6.4 Computational experiments

In this section we investigate the impact of the primary design choices and parameters, in addition to measuring the benefits of systematic *Sequencing* optimization. The resulting algorithm is also compared against state-of-the-art algorithms from the literature. The benchmark instances proposed by Uchoa et al. (2017) are considered throughout the experiments.

The proposed algorithm was coded in C++ and executed on Intel(R) Xeon(R) E5-2680v3 CPU @ 2.5GHz computers with 64GB of RAM memory running Red Hat Enterprise Linux ComputeNode 6.5. A memory limit of 8Gb was imposed for the cache (see Section 6.3.2). *Concorde* was employed to solve TSP problems when searching in neighborhood $\mathcal{S}^{\mathrm{A}}$, always using memory cache to prevent decoding twice the same route as much as possible.

6.4.1 Search space and computational effort

Initial experiments were conducted with a simple local search on search spaces $\mathcal{S}$, $\mathcal{S}^{\mathrm{A}}$ and $\mathcal{S}^{\mathrm{B}}_k$ to observe the growth of computational time and identify a range of k values for which the approach remains practical. This local search is equivalent to that presented in Algorithm 6.1, beginning from an initial solution generated by the straightforward *savings algorithm* proposed by Clarke and Wright (1964). The only noteworthy observation concerns the value of ψ, which is set to $\psi = \infty$ within these experiments. The goal is to analyze the results of exploring the search spaces without the interference of this particular move filter.

Figure 6.7 presents two boxplot graphs: the first indicates the solution gap obtained by the local search and the second presents the required runtime. These graphs consider the different search spaces $\mathcal{S}$, $\mathcal{S}^{\mathrm{B}}_k$ ($k \in [1, 9]$) and $\mathcal{S}^{\mathrm{A}}$. Each boxplot aggregates 2,000 solutions resulting from 20 executions per instance, each with a different random seed. The dotted lines indicate average values. Although instances may differ considerably from each other, they could be aggregated in Figure 6.7 given the homogeneous behavior of the local search for all instances. Differences are observed, however, in the degree of improvement observed for different instances. The figure enables one to conclude that searching in space $\mathcal{S}^{\mathrm{A}}$ provides, as expected, better solutions on average. It also reveals how solution quality consistently increases as k is incremented

(remember that $\mathcal{S} = \mathcal{S}_0^{\mathrm{B}}$). Nevertheless, there is a visible gap between $\mathcal{S}_k^{\mathrm{B}}$ ($k \leq 9$) and $\mathcal{S}^{\mathrm{A}}$ in terms of solution quality for the considered instances.

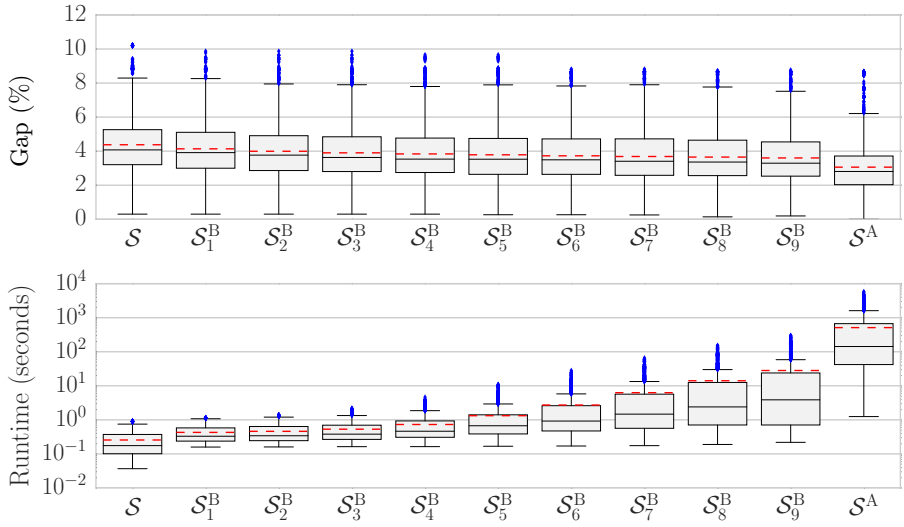
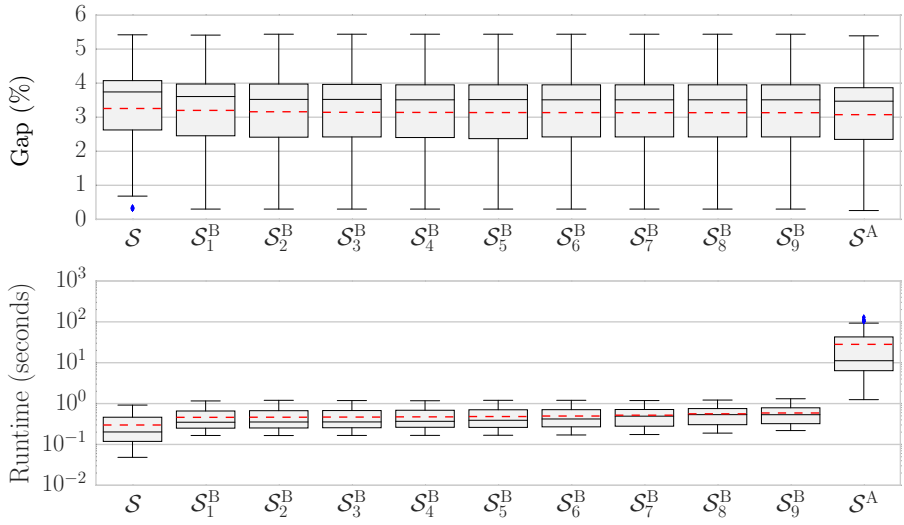


Figure 6.7: Results (solution quality and runtime) of local search solution on different search spaces

The computational times required by the local search to converge considering the different search spaces are also presented in Figure 6.7. The runtime includes the computational time required by B&S and *Concorde*. The boxplots are presented in log-scale given the enormous difference when searching in $\mathcal{S}$ and $\mathcal{S}^{\mathrm{A}}$. When exploring space $\mathcal{S}_k^{\mathrm{B}}$, the runtime was always below two seconds for $k \leq 2$. When applying *Concorde* solver (space $\mathcal{S}^{\mathrm{A}}$), however, the average runtime was approximately 500 seconds, with runtimes for large instances exceeding 1 hour. Clearly, the additional runtime required to run *Concorde* in this case prohibits embedding the proposed local search within metaheuristic algorithms in practice.

The impact of searching in different spaces on both solution quality and runtime was also analyzed for instances with different average route cardinalities. Figure 6.8 presents four boxplot graphs concerning two instance subsets. The first instance subset (Figure 6.8a) is composed of the 20% instances with smallest average route cardinality. This results in instances with average route cardinalities in the range $[3.0, 4.55]$. The second instance subset (Figure 6.8b) represents the other extreme, constituted by the 20% instances with largest

(a) Instances with average route cardinality in range [3.0, 4.55]:



(b) Instances with average route cardinality in range [16.47, 24.43]:

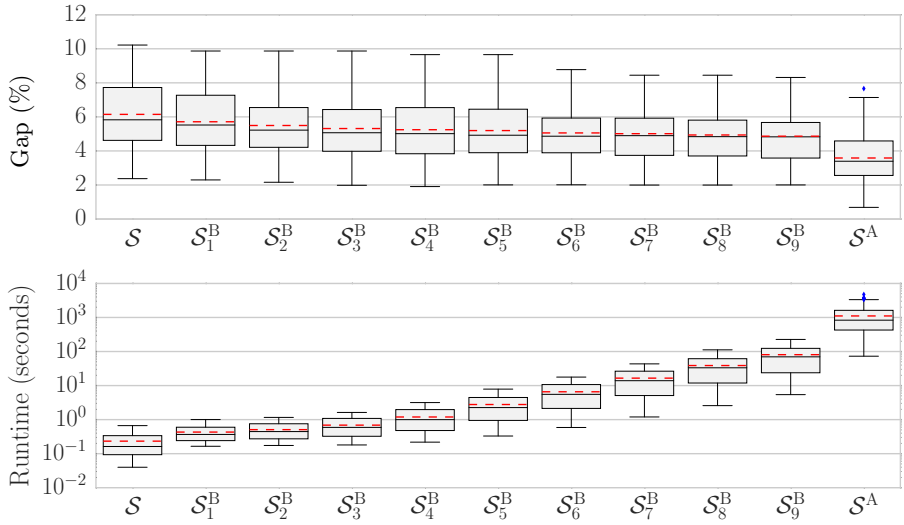


Figure 6.8: Results (solution quality and runtime) of local search on different search spaces for instances with different route cardinalities

average route cardinality. For these instances, the average route cardinality is within the range $[16.47, 24.43]$. In total, each boxplot represents 400 solutions.

Figure 6.8 enables us to draw many conclusions. First, it is clear that the impact of searching in $\mathcal{S}_k^{\mathrm{B}}$ (for $k \geq 1$) and $\mathcal{S}^{\mathrm{A}}$ on solution quality is far less noticeable for instances with small route cardinalities. Although some difference may be noticed even in Figure 6.8a, it is far from being as evident as in Figure 6.8b. As one would expect, the impact on solution quality is much higher on instances with more customers per vehicle. The same is true for the impact on computational time when considering search spaces $\mathcal{S}_k^{\mathrm{B}}$ ($k \leq 9$). For search space $\mathcal{S}^{\mathrm{A}}$, solving the TSP (via *Concorde*) remains an obstacle in terms of runtime for all instances.

6.4.2 Parameters and speedup techniques

Given that exploring $\mathcal{S}^{\mathrm{A}}$ with the proposed techniques resulted in prohibitively long runtimes, we proceed by considering only search spaces $\mathcal{S}$ and $\mathcal{S}_k^{\mathrm{B}}$ within an advanced metaheuristic: the *Unified Hybrid Genetic Search* (UHGS). The UHGS employed is described in full length by Vidal et al. (2014). The local search within the UHGS was replaced by the one proposed in this chapter (see Algorithm 6.1). The split method and moves applied remained the same (*Swap*, *Relocation*, *2-opt*, *2-opt**, and immediate generalizations such as *Swap1-2* and *Relocate2*). Note that the UHGS accepts infeasible solutions due to capacity, penalizing them in the objective function. A penalty of 100 is applied to each capacity unit extrapolated.

The parameters require calibration to establish a good balance between runtime and solution quality. Within preliminary experiments, we obtained a configuration which represents a reasonable balance: $k = 2$, *set cache* (tunneling) activated and ψ computed dynamically to filter 90%-95% of the moves. In this section we analyze the impact of modifying these parameters, examining each parameter in turn while keeping the others fixed. The experimentation is restricted to a set of 30 medium-scale CVRP instances with $n \in \{195, ..., 311\}$, as they require limited computational time while being simultaneously representative of borderline cases and challenging for current state-of-the-art heuristics (which not necessarily obtain the optimal solution on all runs). For each instance, ten runs have been conducted with different random seeds. The maximum number of iterations without improvements in the UHGS was set to 20,000 for all experiments in this section.

Impact of cache strategy and parameter k

Figure 6.9 presents the impact of the cache strategy (*set* and *permutation cache*) and parameter k concerning both solution quality and computational time. It enables one to conclude that searching in $\mathcal{S}_k^{\mathrm{B}}$ $(k > 0)$ results in better solutions than searching in $\mathcal{S}$ for these instances. Increasing the value of k leads to better solutions, particularly when *permutation cache* is employed. When *set cache* is employed, the impact of k's value ($k \in [1, 4]$) on solution quality is less noticeable for these instances. Nevertheless, larger k values come at a high cost: runtime grows considerably.

Figure 6.9 also shows that the *set cache* is an overall better strategy than *permutation cache*. Note how it resulted in both better solutions and smaller runtimes for all k values considered. This is an expected result, given the *set cache* effects: it shapes $\mathcal{S}_k^{\mathrm{B}}$ into a space closer to $\mathcal{S}^{\mathrm{A}}$ during the search and enables better memory management by storing at most one route per customer set.

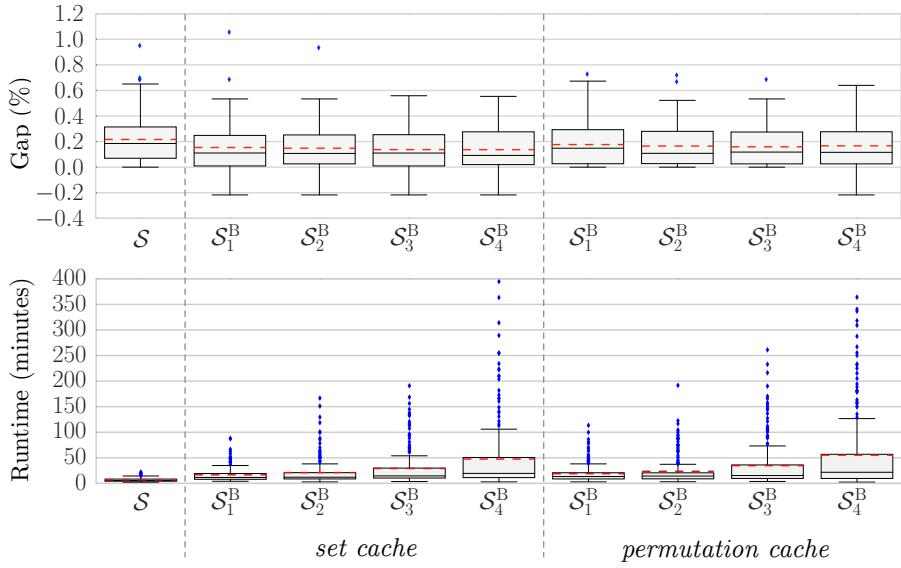


Figure 6.9: Results (solution quality and runtime) of UHGS for different cache strategies and k values

Impact of the dynamic move filter

Figure 6.10 shows the impact of different dynamic move filter values $[\xi^-, \xi^+]$, in addition to filtering all non-proving moves ($\psi = 0$). These experiments employ the *set cache* strategy and $k = 2$. It is noticeable from Figure 6.10 that filtering all moves which do not immediately improve the solution ($\psi = 0$) results in worse solutions. Filtering 95%-97.5% leads to better results in terms of solution quality. Further reducing the filter leads to higher-quality solutions. This is an interesting finding: moves generally discarded in regular local search methods, when applied in combination with B&S lead to better solutions on average. In this particular example, some negative gaps are also witnessed, indicating improvements over the best known solution for some instance(s). However, the runtime increases considerably when less moves are filtered, revealing an evident trade-off. Seeking balance between solution quality and runtime, we selected the range $[90\%, 95\%]$ as default.

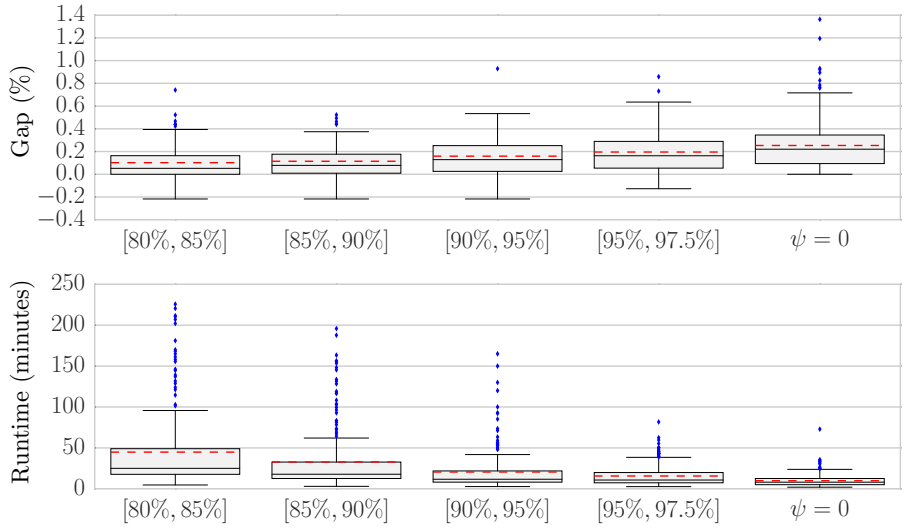


Figure 6.10: Results (solution quality and runtime) of UHGS for different $[\xi^-, \xi^+]$ values

6.4.3 Final results

Results for all 100 instances proposed by Uchoa et al. (2017) are reported in this section. These instances, as indicated by Table 6.1, are grouped into three

categories: *small*, *medium* and *large*.

Table 6.1: Instance groups

Category	Customers	Instances	# Instances
small	100 – 250	X-n101-k25 – X-n247-k47	32
medium	250 – 500	X-n251-k28 – X-n491-k59	36
large	500 – 1000	X-n502-k39 – X-n1001-k43	32

Tables 6.2, 6.3 and 6.4 present the results obtained with the proposed approach, comparing it against the state-of-the-art algorithms in the literature:

ILS : Iterated Local Search method proposed by Subramanian et al. (2013a).

UHGS : Unified Hybrid Genetic Search algorithm proposed by Vidal et al. (2014).

ASB-RR : Adjacent string removal and greedy insertion with blinks heuristic developed by Christiaens and Vanden Berghe (2016).

The tables present the average runtime (in minutes), and average and best solution values for all methodologies. ILS, UHGS and ASB-RR were executed 50 times for each instance. The proposed approach, however, was executed 10 times due to its elevated runtime for some instances, and a runtime limit of 24 hours was imposed for each execution. The maximum number of iterations without improvements within the UHGS was set to 50,000, mimicking Uchoa et al. (2017). Moreover, best results are highlighted in the table, with $\circledast$ indicating an improvement over the best solution produced by ILS, UHGS and ASB-RR. Finally, the average gap to the best solution (including those generated during this research) is presented for each algorithm in each table's final row.

Tables 6.2, 6.3 and 6.4 highlights how exploring $\mathcal{S}_2^{\mathrm{B}}$ resulted in several improvements over the best solutions obtained by the state-of-the-art methods considered. The methodology clearly improves upon the state-of-the-art when *small* and *medium* instances are considered. Moreover, searching in $\mathcal{S}_2^{\mathrm{B}}$ rather than $\mathcal{S}$ (column UHGS) consistently improved final solution quality. The best results for *large* instances, however, were obtained by ASB-RR. In terms of runtime, the proposed approach performed worse than the other heuristics, as expected, reaching the runtime limit of 24h in a few executions.

Among the results, we highlight some unexpected improvements upon the best known solutions for rather small instances. When analyzing the improved solution for X-n256-k16, for example, one interesting characteristic was observed:

Table 6.2: Results for *small* instances from Uchoa et al. (2017)

#	Instance	ILS			UHGS			ASB-RR			Proposed algorithm			
		Time	Average	Best	Time	Average	Best	Time	Average	Best	Time	Average	Best	
1	X-n101-k25	0.1	27591.0	27591	1.4	27591.0	27591	0.8	27591.0	27591	2.4	27591.0	27591	
2	X-n106-k14	2.0	26375.9	26362	4.0	26381.8	26378	1.3	26381.5	26362	17.6	26374.3	26362	
3	X-n110-k13	0.2	14971.0	14971	1.6	14971.0	14971	1.0	14971.1	14971	3.9	14971.0	14971	
4	X-n115-k10	0.2	12747.0	12747	1.8	12747.0	12747	0.2	12747.0	12747	6.4	12747.0	12747	
5	X-n120-k6	1.7	13337.6	13332	2.3	13332.0	13332	1.6	13332.0	13332	38.3	13332.0	13332	
6	X-n125-k30	1.4	55673.8	55539	2.7	55542.1	55539	3.1	55556.3	55542	6.1	55540.0	55539	
7	X-n129-k18	1.9	28998.0	28948	2.7	28948.5	28940	1.5	28948.8	28940	8.4	28940.0	28940	
8	X-n134-k13	2.1	10947.4	10916	3.3	10934.9	10916	2.8	10940.1	10916	20.2	10916.0	10916	
9	X-n139-k10	1.6	13603.1	13590	2.3	13590.0	13590	2.0	13595.4	13590	8.9	13590.0	13590	
10	X-n143-k7	1.6	15745.2	15726	3.1	15700.2	15700	2.1	15705.8	15700	33.1	15700.0	15700	
11	X-n148-k46	0.8	43452.1	43448	3.2	43448.0	43448	2.8	43469.2	43448	5.1	43448.0	43448	
12	X-n153-k22	0.5	21400.0	21340	5.5	21226.3	21220	5.6	21229.4	21220	15.2	21225.6	21220	
13	X-n157-k13	0.8	16876.0	16876	3.2	16876.0	16876	3.7	16878.6	16876	28.4	16876.0	16876	
14	X-n162-k11	0.5	14160.1	14138	3.3	14141.3	14138	3.4	14157.1	14138	12.2	14138.0	14138	
15	X-n167-k10	0.9	20608.7	20562	3.7	20563.2	20557	3.2	20560.8	20557	36.1	20557.0	20557	
16	X-n172-k51	0.6	45616.1	45607	3.8	45607.0	45607	5.3	45619.2	45607	5.7	45607.0	45607	
17	X-n176-k26	1.1	48249.8	48140	7.6	47957.2	47812	5.2	47849.6	47812	16.2	47830.7	47812	
18	X-n181-k23	1.6	25571.5	25569	6.3	25591.1	25569	5.5	25579.8	25569	13.9	25569.4	25569	
19	X-n186-k15	1.7	24186.0	24145	5.9	24147.2	24145	4.0	24178.4	24149	20.2	24145.0	24145	
20	X-n190-k8	2.1	17143.1	17085	12.1	16987.9	16980	9.1	16984.9	16980	161.9	16985.3	16980	
21	X-n195-k51	0.9	44234.3	44225	6.1	44244.1	44225	6.1	44298.5	44241	9.3	44283.8	44225	
22	X-n200-k36	7.5	58697.2	58626	8.0	58626.4	58578	6.7	58636.1	58578	12.0	58615.1	58578	
23	X-n204-k19	1.1	19625.2	19570	5.4	19571.5	19565	4.9	19662.3	19565	15.7	19567.0	19565	
24	X-n209-k16	3.8	30765.4	30667	8.6	30680.4	30656	6.0	30669.4	30656	35.7	30671.3	30656	
25	X-n214-k11	2.3	11126.9	10985	10.2	10877.4	10856	8.7	10908.6	10873	52.3	10872.1	10856	
26	X-n219-k73	0.9	117595.0	117595	7.7	117604.9	117595	8.0	117650.4	117595	18.2	117600.5	117595	
27	X-n223-k34	8.5	40533.5	40471	8.3	40499.0	40437	7.6	40529.9	40448	18.6	40478.4	40437	
28	X-n228-k23	2.4	25795.8	25743	9.8	25779.3	25742	10.5	25790.9	25744	29.0	25768.0	25743	
29	X-n233-k16	3.0	19336.7	19266	6.8	19288.4	19230	8.1	19269.7	19232	34.0	19276.5	19230	
30	X-n237-k14	3.5	27078.8	27042	8.9	27067.3	27042	7.2	27089.7	27042	32.2	27048.8	27042	
31	X-n242-k48	17.8	82874.2	82774	12.4	82948.7	82804	9.9	82884.4	82775	18.1	82920.9	82751	⊛
32	X-n247-k47	2.1	37507.2	37289	20.4	37284.4	37274	18.4	37323.2	37274	27.7	37388.9	37274	
	Average gap:		0.31%	0.12%		0.07%	0.00%		0.11%	0.01%		0.05%	0.00%	

Table 6.3: Results for *medium* instances from Uchoa et al. (2017)

#	Instance	ILS			UHGS			ASB-RR			Proposed algorithm			
		Time	Average	Best	Time	Average	Best	Time	Average	Best	Time	Average	Best	
33	X-n251-k28	10.8	38840.0	38727	11.7	38796.4	38699	9.8	38791.0	38687	20.2	38778.7	38684	⊛
34	X-n256-k16	2.0	18883.9	18880	6.5	18880.0	18880	11.5	18888.9	18880	23.0	18867.7	18839	⊛
35	X-n261-k13	6.7	26869.0	26706	12.7	26629.6	26558	11.8	26642.3	26558	48.6	26618.1	26558	
36	X-n266-k58	10.0	75563.3	75478	21.4	75759.3	75517	10.8	75617.8	75478	29.9	75710.7	75478	
37	X-n270-k35	9.1	35363.4	35324	11.3	35367.2	35303	11.4	35362.2	35323	18.9	35314.6	35303	
38	X-n275-k28	3.6	21256.0	21245	12.0	21280.6	21245	13.3	21268.6	21245	22.7	21255.0	21245	
39	X-n280-k17	9.6	33769.4	33624	19.1	33605.8	33505	17.7	33628.1	33529	136.2	33587.9	33503	⊛
40	X-n284-k15	8.6	20448.5	20295	19.9	20286.4	20227	15.3	20286.6	20240	97.7	20282.1	20228	
41	X-n289-k60	16.1	95450.6	95315	21.3	95469.5	95244	14.3	95352.2	95233	41.7	95447.2	95211	⊛
42	X-n294-k50	12.4	47254.7	47190	14.7	47259.0	47171	14.7	47274.5	47210	27.0	47272.7	47161	⊛
43	X-n298-k31	6.9	34356.0	34239	10.9	34292.1	34231	14.5	34276.0	34234	20.7	34276.3	34231	
44	X-n303-k21	14.2	21895.8	21812	17.3	21850.9	21748	17.3	21776.5	21751	48.4	21811.2	21744	⊛
45	X-n308-k13	9.5	26101.1	25901	15.3	25895.4	25859	25.7	26207.7	25931	112.8	25897.3	25861	
46	X-n313-k71	17.5	94297.3	94192	22.4	94265.2	94093	18.9	94182.4	94063	30.6	94280.4	94045	⊛
47	X-n317-k53	8.6	78356.0	78355	22.4	78387.8	78355	22.0	78392.4	78355	50.3	78385.3	78355	
48	X-n322-k28	14.7	29991.3	29877	15.2	29956.1	29870	16.9	29927.6	29849	27.7	29892.5	29834	⊛
49	X-n327-k20	19.1	27812.4	27599	18.2	27628.2	27564	21.6	27631.4	27608	68.7	27590.8	27532	⊛
50	X-n331-k15	15.7	31235.5	31105	24.4	31159.6	31103	20.4	31128.2	31122	102.1	31126.7	31103	
51	X-n336-k84	21.4	139461.0	139197	38.0	139534.9	139210	22.8	139373.4	139209	66.0	139460.1	139303	
52	X-n344-k43	22.6	42284.0	42146	21.7	42208.8	42099	21.5	42158.5	42079	39.7	42156.1	42056	⊛
53	X-n351-k40	25.2	26150.3	26021	33.7	26014.0	25946	26.5	25982.1	25938	51.5	25981.8	25938	
54	X-n359-k29	48.9	52076.5	51706	34.9	51721.7	51509	23.1	51577.8	51505	112.0	51640.7	51555	
55	X-n367-k17	13.1	23003.2	22902	22.0	22838.4	22814	36.1	22833.4	22814	117.3	22876.2	22814	
56	X-n376-k94	7.1	147713.0	147713	28.3	147750.2	147717	32.0	147783.6	147721	70.3	147740.5	147714	
57	X-n384-k52	34.5	66372.5	66116	40.2	66270.2	66081	25.9	66107.4	65963	56.8	66170.3	65997	
58	X-n393-k38	20.8	38457.4	38298	28.7	38374.9	38269	30.4	38394.1	38331	49.3	38309.3	38260	⊛
59	X-n401-k29	60.4	66715.1	66453	49.5	66365.4	66243	38.0	66248.5	66189	110.2	66359.0	66212	
60	X-n411-k19	23.8	19954.9	19792	34.7	19743.8	19718	58.4	19768.5	19731	126.0	19736.7	19721	
61	X-n420-k130	22.2	107838.0	107798	53.2	107924.1	107798	47.9	107879.2	107817	87.7	107913.7	107798	
62	X-n429-k61	38.2	65746.6	65563	41.5	65648.5	65501	35.0	65593.6	65485	65.6	65661.6	65470	⊛
63	X-n439-k37	39.6	36441.6	36395	34.6	36451.1	36395	42.1	36473.8	36426	57.1	36410.1	36395	
64	X-n449-k29	59.9	56204.9	55761	64.9	55553.1	55378	38.0	55411.2	55272	132.6	55432.7	55330	
65	X-n459-k26	60.6	24462.4	24209	42.8	24272.6	24181	56.5	24242.2	24175	92.9	24226.0	24145	⊛
66	X-n469-k138	36.3	222182.0	221909	86.7	222617.1	222070	48.0	222227.1	221984	142.3	222427.5	222235	
67	X-n480-k70	50.4	89871.2	89694	67.0	89760.1	89535	50.5	89559.2	89458	73.1	89744.7	89513	
68	X-n491-k59	52.2	67226.7	66965	71.9	66898.0	66633	51.4	66645.5	66517	81.9	66794.1	66607	
	Average gap:		0.57%	0.21%		0.28%	0.05%		0.23%	0.05%		0.20%	0.02%	

Table 6.4: Results for *large* instances from Uchoa et al. (2017)

#	Instance	ILS			UHGS			ASB-RR			Proposed algorithm			
		Time	Average	Best	Time	Average	Best	Time	Average	Best	Time	Average	Best	
69	X-n502-k39	80.8	69346.8	69284	63.6	69328.8	69253	60.9	69274.7	69243	177.7	69277.1	69247	
70	X-n513-k21	35.0	24434.0	24332	33.1	24296.6	24201	77.1	24292.1	24238	99.4	24256.2	24201	
71	X-n524-k137	27.3	155005.0	154709	80.7	154979.5	154774	151.4	154807.2	154651	207.3	155038.1	154787	
72	X-n536-k96	62.1	95700.7	95524	107.5	95330.6	95122	74.7	95173.2	95006	144.5	95335.4	95112	
73	X-n548-k50	64.0	86874.1	86710	84.2	86998.5	86822	64.5	86798.0	86710	136.6	86881.0	86778	
74	X-n561-k42	68.9	43131.3	42952	60.6	42866.4	42756	73.8	42868.1	42774	77.2	42860.0	42733	⊛
75	X-n573-k30	112.0	51173.0	51092	188.2	50915.1	50780	113.0	50804.6	50737	782.4	50876.9	50801	
76	X-n586-k159	78.5	190919.0	190612	175.3	190838.0	190543	86.3	190600.7	190484	234.3	190752.4	190442	⊛
77	X-n599-k92	73.0	109384.0	109056	125.9	109064.2	108813	75.4	108688.6	108548	166.9	108993.3	108576	
78	X-n613-k62	74.8	60444.2	60229	117.3	59960.0	59778	88.1	59731.3	59585	103.6	59859.7	59654	
79	X-n627-k43	162.7	62905.6	62783	239.7	62524.1	62366	89.3	62317.1	62219	543.1	62442.9	62254	
80	X-n641-k35	140.4	64606.1	64462	158.8	64192.0	63839	92.5	63850.3	63750	304.4	64105.6	63859	
81	X-n655-k131	47.2	106782.0	106780	150.5	106899.1	106829	109.6	106844.6	106813	253.2	106855.6	106804	
82	X-n670-k126	61.2	147676.0	147045	264.1	147222.7	146705	198.9	146720.4	146451	267.7	147663.9	147163	
83	X-n685-k75	73.9	68988.2	68646	156.7	68654.1	68425	135.1	68369.0	68271	177.0	68596.0	68496	
84	X-n701-k44	210.1	83042.2	82888	253.2	82487.4	82293	122.5	82065.4	81974	368.0	82409.2	82174	
85	X-n716-k35	225.8	44171.6	44021	264.3	43641.4	43525	158.3	43483.8	43426	437.2	43599.9	43498	
86	X-n733-k159	111.6	137045.0	136832	244.5	136587.6	136366	143.2	136389.3	136255	334.2	136607.4	136424	
87	X-n749-k98	127.2	78275.9	77952	313.9	77864.9	77715	146.3	77509.2	77380	308.3	77862.8	77605	
88	X-n766-k71	242.1	115738.0	115443	383.0	115147.9	114683	174.4	114761.1	114590	330.5	115115.9	114812	
89	X-n783-k48	235.5	73722.9	73447	269.7	73009.6	72781	170.2	72660.7	72492	351.2	72892.4	72738	
90	X-n801-k40	432.6	74005.7	73830	289.2	73731.0	73587	137.1	73436.7	73347	424.0	73651.6	73466	
91	X-n819-k171	148.9	159425.0	159164	374.3	158899.3	158611	172.5	158423.0	158305	675.6	158849.0	158592	
92	X-n837-k142	173.2	195027.0	194804	463.4	194476.5	194266	166.8	193976.9	193824	634.9	194504.0	194356	
93	X-n856-k95	153.7	89277.6	89060	288.4	89238.7	89118	160.0	89131.3	89050	314.6	89220.0	89020	⊛
94	X-n876-k59	409.3	100417.0	100177	495.4	99884.1	99715	217.4	99483.2	99388	543.1	99780.3	99610	
95	X-n895-k37	410.2	54958.5	54713	321.9	54439.8	54172	212.5	54085.8	53993	500.2	54407.4	54254	
96	X-n916-k207	226.1	330948.0	330639	560.8	330198.3	329836	215.3	329509.5	329299	1082.5	330153.2	329866	
97	X-n936-k151	202.5	134530.0	133592	531.5	133512.9	133140	412.7	133117.3	133014	1022.2	133729.3	133376	
98	X-n957-k87	311.2	85936.6	85697	432.9	85822.6	85672	202.4	85620.0	85546	307.9	85681.5	85555	
99	X-n979-k58	687.2	120253.0	119994	554.0	119502.1	119194	276.6	119120.4	119065	928.4	119527.7	119188	
100	X-n1001-k43	792.8	73985.4	73776	549.0	72956.0	72742	284.3	72528.1	72415	952.8	72903.3	72629	
	Average gap:		0.90%	0.63%		0.43%	0.19%		0.14%	0.01%		0.38%	0.16%	

only 16 vehicles are used, with the impressive average capacity usage of 99.6%. Clearly, this is a difficult solution to obtain since no other algorithm was capable of generating it before.

6.5 Conclusions and future work

This chapter studied decision-set decompositions for the classic CVRP. It was shown that decomposing the problem into *Assignment* and *Sequencing* and conducting local search in the *Assignment* space ($\mathcal{S}^{\mathrm{A}}$) consistently generates better results than searching in the complete search space $\mathcal{S}$. The *Sequencing* was solved by the *Concorde* solver. However, the additional runtime required for decoding primitive solutions (assignments) into complete solutions prohibited employing this technique within state-of-the-art metaheuristic algorithms. To circumvent this issue the B&S neighborhood was employed to derive an intermediate search space ($\mathcal{S}_k^{\mathrm{B}}$). It was shown that searching in $\mathcal{S}_k^{\mathrm{B}}$ also provides better solutions than searching in $\mathcal{S}$ for a simple local search method.

Different techniques were proposed and evaluated for efficiently exploring space $\mathcal{S}_k^{\mathrm{B}}$: *neighborhood size reduction*, *dynamic move filters*, *concatenation techniques* and *efficient memory structures*. Moreover, memory was also employed to reshape $\mathcal{S}_k^{\mathrm{B}}$ into a search space more and more similar to $\mathcal{S}^{\mathrm{A}}$ as the search progresses. The combination of these techniques within the UHGS solver exploring space $\mathcal{S}_k^{\mathrm{B}}$ resulted in an improved algorithm. Multiple instances from the literature had their best known solution improved. When compared against the state-of-the-art for the CVRP, the proposed algorithm within UHGS obtained higher-quality solutions for small- and medium-size instances.

Future work includes further investigating computational techniques to efficiently explore search space $\mathcal{S}^{\mathrm{A}}$. Employing *Concorde* resulted in extremely long runtimes. However, the resolution of the TSP may be avoided via lower bounds which can be quickly computed. Moreover, *Concorde* is not optimized to handle millions of small cardinality routes and therefore implementing a dedicated TSP-solver also represents one future research direction. Finally, the proposed approach should also be evaluated for immediate generalizations of the CVRP.

Chapter 7

Swap-body Vehicle Routing Problem

This chapter investigates the *Swap-Body Vehicle Routing Problem* (SBVRP), a generalization of the classical Vehicle Routing Problem (VRP) based on real problems faced by industry. The SBVRP was proposed by the EURO Working Group on Vehicle Routing and Logistics Optimization (VeRoLog) and the PTV Group during the First VeRoLog Solver Challenge (Heid et al., 2014). It is a generalization of the classical *Vehicle Routing Problem* (VRP) in which customers are served by vehicles whose sizes may be enlarged via the addition of a swap body (trailer). The inclusion of a swap body doubles vehicle capacity while also increasing its operational cost. However, not all customers may be served by vehicles consisting of two bodies. Therefore swap locations are present where one of the bodies may be temporarily parked, enabling double body vehicles to serve customers requiring a single body. Both total travel time and distance incur costs that should be minimized, while the number of customers visited by a single vehicle is limited both by its capacity and by a maximum travel time.

This chapter is a minor adaptation of Toffolo et al. (2018)[1] and presents the stochastic local search method that won the First VeRoLog Solver Challenge. State of the art VRP approaches do not accommodate SBVRP generalizations

[1] **Toffolo, T. A. M.**, Christiaens, J., Van Malderen, S., Wauters, T., Vanden Berghe, G. (To appear). Stochastic local search with learning automaton for the swap-body vehicle routing problem. *Computers & Operations Research.* 89:68–81 (In press)

well, motivating the investigation of algorithms which take advantage of the swap body characteristic. The proposed algorithm combines both general and dedicated heuristic components with a learning scheme. Classical, problem-specific and subproblem optimization neighborhoods are employed to efficiently explore the solution space. The algorithm improves the best known solution for the majority of the instances proposed during the challenge. Results are also presented for a new set of instances with the aim of stimulating further research on the SBVRP.

This chapter is divided into 5 sections. Section 7.1 introduces the problem, providing a detailed description. Section 7.2 presents a literature overview about the SBVRP and related work. The algorithm is proposed and described within Section 7.3. The neighborhood structures considered for the local search are discussed in Section 7.4, with particular emphasis on those employing subproblem optimization. Section 7.5 presents computational experiments and, finally, Section 7.6 summarizes the conclusions and indicates future research directions.

7.1 Introduction

The classical VRP is one of the most studied problems in combinatorial optimization and is defined under capacity and route length constraints (Cordeau et al., 2007). The SBVRP primarily differs from the VRP insofar as vehicles consist of either one or two bodies (trailers). The lengthened vehicles are called trains and have exactly twice the capacity of the regular vehicles (trucks). Figure 7.1 shows an example of a truck, a swap body (with a trailer) and a train, respectively.

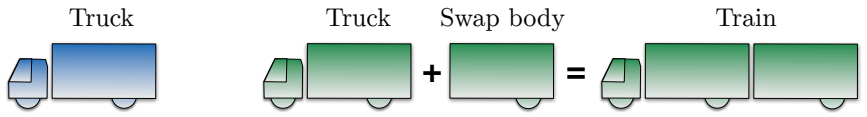


Figure 7.1: Vehicle type examples

Customers have individual demands and must be served by exactly one vehicle. Three types of customers are considered: those who can only be reached by trucks, those who can be served by both trains and trucks, and those whose demands exceed the capacity of a truck and must be attended to by trains.

Customers are geographically dispersed. Travel times and distances between all locations are given.

In addition to the depot and customers' locations, swap locations are present, where one of the bodies of a train may be temporarily left, enabling the vehicle to serve customers with a single body (truck).

The SBVRP considers both total time and distance to derive costs that should be minimized. These costs vary depending on whether the considered vehicle is a train or truck. Furthermore, additional costs for operations at swap locations are also considered. Vehicles routes are limited by both their capacity and a maximum travel duration.

This chapter proposes a stochastic local search heuristic approach to the problem. Initially, a naive solution is quickly built. Different intensification and diversification strategies are subsequently applied to improve the solution. These strategies include a subproblem optimization scheme and different neighborhood structures, both of which are embedded in a metaheuristic framework. A preprocessing procedure reduces the solution space and thus dramatically increases the heuristic's efficiency. The stochastic local search won the First VeRoLog Solver Challenge and continues to outperform all other proposed approaches for the problem.

The approach has significant practical relevance for a range of business activities including production, distribution and also the transportation sector more generally. The delivery of both perishable and urgently-required goods (fuel, for example), which almost always necessitates transportation by road, becomes greatly optimized. Indeed, very often the transportation costs associated with such products are disproportionate when compared against the cost of the products themselves. Furthermore, the approach ensures efficiency with regard to a number of important economic and ecological factors such as: the number of vehicles, number of drivers, travel distance and time, and the environmental impact.

7.1.1 The VeRoLog challenge problem

The SBVRP proposed during the First VeRoLog Solver Challenge is a generalization of the classical VRP and, by consequence, is an $\mathcal{NP}$-Hard problem. It can be defined on a graph $G = (V, A)$, where the vertices V are the locations and the arcs A are the connections between these locations. Three vertex

categories are considered: depot, customers and swap locations. A single depot vertex is defined.

The customers, represented by the subset $C \subset V$, are divided into three groups: truck-only ($C_1 \subseteq C$), flexible ($C_2 \subseteq C$) and train-only ($C_3 \subseteq C$). These groups are defined according to the types of vehicle that can be employed to visit the customers. Truck-only customers can only be attended to by trucks, flexible customers can have their demands satisfied by both trucks and trains, and train-only customers require trains.

All customers $i \in C$ have an associated demand q_i and service time s_i. These demands must be satisfied with exactly one visit. Since the capacity of a swap body is given by constant Q, truck-only and flexible customers' demands must be bounded by Q, such that $q_i \leq Q \ \forall i \in C_1 \cup C_2$. Contrastingly, train-only customers have demands that trucks cannot satisfy, therefore implying $Q < q_i \leq 2Q \ \forall i \in C_3$.

Swap locations, represented by the subset $S \subset V$, are associated with neither demand nor service time. Nevertheless, depending on the operation executed at a swap location, a certain amount of time is consumed. In total, three operations are possible at a swap location, each consuming varying amounts of time:

park : leaves the back swap body of the train at the swap location;

pickup : picks up the swap body that was left at a swap location;

swap : leaves the currently attached swap body and picks up the swap body that was left at the swap location.

The First VeRoLog Solver Challenge organizers also defined a fourth possible operation: *exchange*, which consists of leaving the front swap body of the train at the swap location. However, as Miranda-Bront et al. (2017) have highlighted, the consumed capacity of the bodies may be distributed across routes such that the first action on a swap location is always *park*. Since *exchange* generally requires more time than *park*, it is never utilized.

Each arc $(i, j) \in A$ connects location i to location j, has a distance $d_{i,j}$ and a travel time $t_{i,j}$. Note that the distances and travel times are asymmetric, meaning $d_{i,j}$ and $t_{i,j}$ are not guaranteed to be equal to $d_{j,i}$ and $t_{j,i}$ respectively.

Figure 7.2 shows a graph representation of a small SBVRP instance. Triangles represent swap locations, squares indicate truck-only customers, filled circles denote flexible customers and, finally, open circles identify train-only customers.

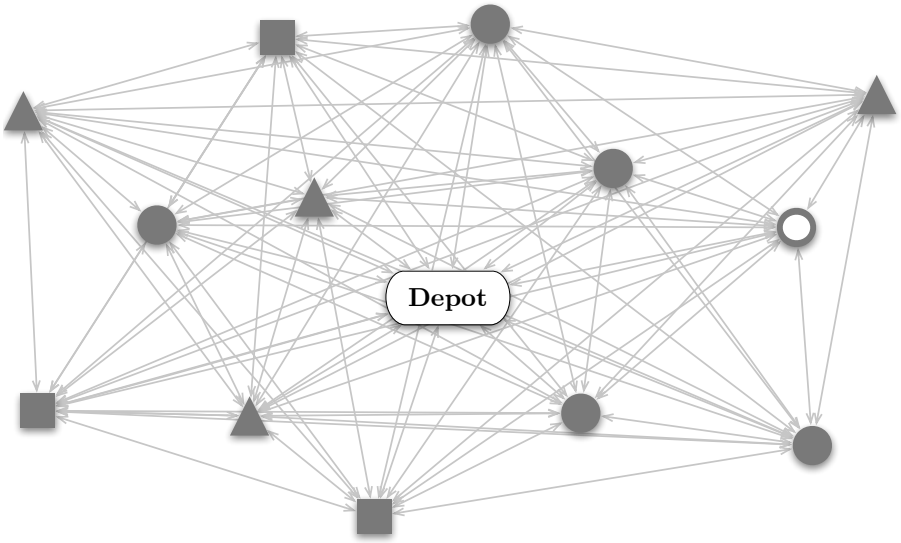


Figure 7.2: Graph representation of a small SBVRP instance

Vehicles must leave and return to the depot with the same swap bodies. Crucially, routes must begin and end in the depot and swap bodies may not be exchanged between vehicles. Therefore, if a vehicle leaves a swap body in a swap location, the body must be retrieved later by the same vehicle. Henceforth, the part of the route that comprises of the customers between the two swap location visits will be referred to as a sub-route.

All routes must respect capacity constraints and a maximum duration T. Each route's duration is given by the sum of its travel times, service times and swap operation times.

In the SBVRP considered the objective is to minimize the total operation cost, given by the sum of two components:

vehicle/driver costs : consisting of a fixed cost for using a vehicle, a cost per kilometer traveled and a cost per hour (driver's cost);

swap body costs : consisting of a fixed cost per additional swap body and a cost per kilometer traveled with it.

A sample solution for the problem depicted by Figure 7.2 is shown in Figure 7.3. This example employs three vehicles: one truck and two trains. Note that one of the routes (route 3) contains a sub-route, therefore indicating it utilizes a swap location. The swap location temporarily stores one of the vehicle's swap bodies, while it visits two truck-only customers. After visiting these two customers (or directly before finishing the sub-route), the vehicle reattaches the parked body and continues towards the next customers.

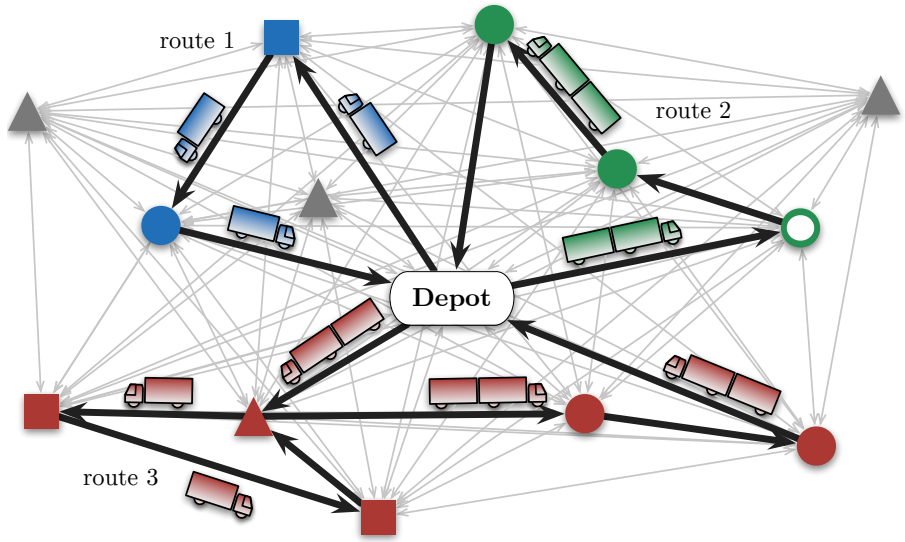


Figure 7.3: Example of a SBVRP solution

7.2 Literature review

The SBVRP considered by this work was introduced recently in the literature. Huber and Geiger (2014) addressed the SBVRP with an iterative Variable Neighborhood Search (VNS) procedure. They employed a cluster-first route-second approach to produce initial solutions. Both sequential and parallel versions of the algorithm were evaluated. Lum et al. (2015) applied a VRP-Reduce algorithm to the SBVRP. They first transform the SBVRP into a classical VRP and then solve the classical problem employing a Simulated Annealing (SA) algorithm. Afterwards, a post-processing procedure produces solutions to the SBVRP, which are subsequently improved with a Variable Neighborhood Descent (VND) method. Miranda-Bront et al. (2017) combined a

cluster-first route-second approach with a Greedy Randomized Adaptive Search Procedure (GRASP). Different constructive heuristics were studied. Absi et al. (2017) proposed a relax-and-repair approach to the SBVRP in which the problem is initially solved as a heterogeneous-fleet VRP with a memetic algorithm. Next, the results are repaired to produce valid solutions for the SBVRP. During the optimization process all solutions are stored to be later used as input to a set-partitioning problem aiming at deriving better solutions. Finally, Todosijević et al. (2017) proposed a method that resembles the one proposed by Huber and Geiger (2014). A cluster-first, route-second constructive heuristic for generating an initial solution, and two General Variable Neighborhood Search (GVNS) heuristics. Both sequential and parallel versions were evaluated. Todosijević et al. (2017) also introduced a mixed integer programming formulation for the SBVRP. Recently, Huber and Geiger (2017) presented a study on the importance of the neighborhoods and their ordering within the VNS algorithm proposed by Huber and Geiger (2014). It was shown that the sequence of neighborhoods matters. In addition, the impact of the synchronization frequency during the parallel execution was also evaluated. Improved results were reported.

Table 7.1 summarizes the characteristics of the published SBVRP methods. Each column represents a different approach: *HG2014/2017* (Huber and Geiger, 2014, 2017), *Lum2015* (Lum et al., 2015), *THUJG2017* (Todosijević et al., 2017), *MB2017* (Miranda-Bront et al., 2017) and *Absi2017* (Absi et al., 2017). Note how all studies addressed the SBVRP with heuristic-based methods. Moreover, although some classic neighborhoods were adapted to consider SBVRP characteristics, the limited number of neighborhoods exploiting problem-specific features is noteworthy.

Other problems bearing many similarities with the SBVRP have been studied by various authors. Gerdessen (1996), for instance, presented a study on the *Vehicle Routing Problem with Trailers* (VRPT). Like the SBVRP, the VRPT considers two vehicle configurations: trucks and trucks with an attached trailer (trains). Although all customers may be served by trains in the VRPT, it proves inconvenient to visit some customers with a large vehicle, such as those located in the city center. The degree of inconvenience is measured by what Gerdessen calls *manoeuvring time*, which consists of the additional time required by trains, as opposed to trucks, when serving a specific customer. Trailers can be parked at any customer site with two additional simplifications considered: firstly, each trailer is parked exactly once and, secondly, all customers have unit demands. Gerdessen proposed three constructive heuristic algorithms and an improvement heuristic based on local search.

Table 7.1: Overview of previous SBVRP strategies in the literature

	Challenge participants				Others
Feature / characteristics	*HG2014/2017*	*Lum2015*	*THUJG2017*	*MB2017*	*Absi2017*
Construction heuristic	Cluster-first route-second	SA (VRP)	Cluster-first route-second	Cluster-first route-second	Heterogeneous fleet VRP
Metaheuristic strategy	VNS	SA, VND	GVNS	GRASP	Multiple strategies
Population based	-	-	-	-	✓
Parallel computing	✓	-	✓	✓	✓
Intra-route neighborhoods					
relocate	✓	-	✓	-	✓
swap	-	-	-	-	✓
2-opt	✓	✓	✓	✓	✓
3-opt	✓	-	-	-	-
Inter-route neighborhoods					
relocate	✓	-	✓	✓	✓
swap	✓	✓	✓	✓	✓
multiple swap	-	✓	✓	-	-
3-exchange	✓	-	-	-	-
or-opt	-	✓	-	✓	-
Problem-specific neighborhoods					
Change swap location	✓	-	-	-	-
Truck-only customer migration	-	✓	-	-	-
Route downgrade (single truck)	-	-	-	✓	-
Repair procedure(s)	-	-	-	-	✓

Another similar problem is the *Truck and Trailer Routing Problem* (TTRP), introduced by Chao (2002). The TTRP is a real-world extension of the VRP in which a limited fleet of trucks and trailers with fixed capacities serve a set of customers from a central depot. Note that, in contrast to the SBVRP, obtaining a feasible solution for the TTRP is not trivial, on account of the limited number of vehicles. The objective is to satisfy customers' demands while minimizing the total travel distance. Similarly to the SBVRP, customers are divided in groups: (*i*) vehicle customers, reachable by either a complete vehicle (truck and trailer) or by a truck alone, and (*ii*) truck customers, reachable only by trucks alone. As with the SBVRP and the VRPT, trailers may be temporarily parked, enabling truck customers to be served exclusively by trucks. Any customer site may serve as a parking place for trailers. Chao (2002) proposed a three-step constructive heuristic and a Tabu Search (TS) method for the TTRP. The algorithm obtained feasible solutions for all instances, despite the high demand-to-capacity ratio (above 90%).

Several papers elaborate upon the TTRP literature. Scheuerer (2006) proposed two constructive algorithms and a TS method, improving the best known results for all instances. Later, Lin et al. (2009) proposed a Simulated Annealing algorithm with a two-level solution representation which employed dummy depots/roots, in conjunction with random neighborhood structures employing three different types of moves. Lin et al. (2009) further improved the best known results for several instances. Caramia and Guerriero (2009) proposed a very interesting hybrid approach based on local search and mathematical programming. They heuristically decompose the problem and use CPLEX to solve two subproblems sequentially. First, each customer is assigned to a route with the objective of minimizing the fleet size. Next, by considering the customers assigned to each vehicle, the individual routes are optimized by minimizing their total tour length. The hybrid algorithm adds constraints to the formulation during each iteration while using a tabu-like customer-route matrix to avoid previously analyzed allocations. Some new best results were produced, in addition to lower bounds for assessing solution quality. Villegas et al. (2013) present a two-phase matheuristic approach to the TTRP which employs locally optimal routes as columns in a set-partitioning formulation. Routes are generated with a metaheuristic consisting of a hybrid GRASP and ILS. Route pool sizes may be controlled, offering a trade-off between solution quality and running time. Very competitive results were obtained compared to previous methods.

Other papers focus on variants of TTRP such as the relaxed TTRP with

unlimited availability of trucks and trailers and the *TTRP with Time Windows* (TTRPTW), introduced by Lin et al. (2010) and Lin et al. (2011) respectively. Recently, the TTRPTW was further studied by Parragh and Cordeau (2017), who proposed a tailor-made branch-and-price algorithm capable of optimally solving instances of up to 100 customers.

Much like with the SBVRP, the majority of the literature on the VRPT and the TTRP address the problem with local search based algorithms. In fact, the majority of the best results for these problems were obtained by metaheuristics relying on local search. This observation, coupled with the $\mathcal{NP}$-hardness of the SBVRP, further motivated the development of local search algorithms when approaching the problem.

7.3 Local search algorithm

A stochastic local search based algorithm is proposed for the SBVRP. The algorithm begins by building a naive feasible solution. Once an initial solution is obtained, the algorithm proceeds to the local search phase that considers several neighborhood structures. A learning algorithm is responsible for choosing the neighborhood to apply at each iteration. To enable escaping from local optima, a hybridization of the metaheuristics Iterated Local Search (ILS) (Lourenço et al., 2003) and Late Acceptance Hill-Climbing (LAHC) (Burke and Bykov, 2017) is considered.

The algorithm's components are explained throughout the following sections. Section 7.3.1 details the constructive algorithm while Section 7.3.2 introduces the hybrid algorithm combining ILS and LAHC. The neighborhood structures employed and the learning mechanism are discussed later in Section 7.4.

7.3.1 Constructive algorithm

A simple and straightforward constructive algorithm is considered for quickly producing a feasible solution. The algorithm generates separate routes from the depot to each customer. Both truck-only and flexible customers are initially served by trucks, whereas train-only customers are served by trains.

The produced solution is expected to be very poor in terms of cost. However, it can be generated quickly in $\mathcal{O}(|C|)$, where $|C|$ is the number

of customers. Additionally, providing good feasible solutions to threshold acceptance algorithms, such as the LAHC, may limit the search space and result in poor final solutions, especially if the initial solution happens to be a local optimum. It is possible to avoid this drawback by feeding the LAHC with a cost larger than the initial solution's. This may, however, result in several worsening modifications at the execution's beginning, eventually wasting the additional effort required to produce a good initial solution.

7.3.2 Hybrid local search algorithm

The proposed algorithm is a hybridization of ILS and LAHC procedures. The ILS metaheuristic was introduced by Lourenço et al. (2003) and relies upon perturbations to escape from local optima. The main principle behind the hybridization is to apply these ILS-like perturbations to the best solution obtained by the LAHC, before re-executing LAHC on the perturbed solution.

The LAHC is a metaheuristic introduced by Burke and Bykov (2008) and further discussed by Burke and Bykov (2017). It is an adaptation of the classic Hill-Climbing heuristic in which the quality of a *late* solution, obtained l iterations before the current, determines whether a new solution is accepted or rejected. This permits the algorithm to accept worsening solutions, enabling it to eventually escape from local optima. Successful applications of the LAHC have been reported by Özcan et al. (2009), Verstichel and Vanden Berghe (2009), Abuhamdah (2010), Goerler et al. (2013) and Yuan et al. (2015), among others.

The LAHC approach to the SBVRP is presented in Algorithm 7.1. Three arguments are required: (i) an initial solution S, (ii) the list size l and (iii) the maximum number of consecutively rejected neighbors m. The algorithm begins by filling the late acceptance list, F, with the initial solution cost (lines 1-2). Following this, the best solution is stored and counters p and r are initialized (lines 3-4), where p is a cyclic pointer to a position in the late acceptance list and r the current number of consecutive rejections. The main loop (line 5) begins by selecting a neighborhood (line 6). The learning algorithm is responsible for this selection. Afterwards, a new neighboring solution is generated (line 7). If this solution is at least as good as the previous or has a lower objective value than the considered entry in the late acceptance list, it is accepted (line 8). Note that counter r is reset only if S' is an improving solution over S (lines 9-10). Solution S is then updated (line 11). If the best solution S^* is improved, it is also updated (lines 12-13). Next, the late acceptance list is updated (line

14) as well as counters p and r (lines 15-16). Finally, the best solution obtained is returned once the main loop finishes (line 17).

Algorithm 7.1: Late Acceptance Hill-Climbing

```
   Input: Initial solution S, list size l and maximum consecutive rejections m
   LAHC(S, l, m)
1  |  foreach p ∈ {0, ..., l − 1} do
2  |  |  F_p ← f(S) // LAHC list is initialized with initial solution's cost
3  |  S* ← S
4  |  p ← r ← 0
5  |  while r ≤ m and time limit is not reached do
6  |  |  N ← selected neighborhood structure
7  |  |  S' ← random neighbor N(S)
8  |  |  if f(S') ≤ f(S) or f(S') ≤ F_p then
9  |  |  |  if f(S') < f(S) then
10 |  |  |  |  r ← 0 // since solution is an improvement, r is reset
11 |  |  |  S ← S'
12 |  |  |  if f(S) < f(S*) then
13 |  |  |  |  S* ← S
14 |  |  F_p ← f(S) // LAHC list is updated with current solution's cost
15 |  |  p ← (p + 1) mod l // p is updated to point to the next list position
16 |  |  r ← r + 1
17 |  return S*
```

The hybrid algorithm combining ILS and LAHC is presented in Algorithm 7.2. It begins by producing the initial solution (line 1) and setting the perturbation level ρ to 1 (line 2). The main loop (line 3) first calls the LAHC algorithm (line 4). If the solution produced by LAHC is better than the current best solution, the best solution is updated and the perturbation level is reset to 1 (lines 5-7). Otherwise, the current solution is reset to the best solution (lines 8-9). Afterwards, the perturbation is executed (lines 10-12), which corresponds to applying ρ random moves to the current solution. Next, the perturbation level ρ is increased and the loop repeated. Note that the given maximum perturbation level ρ_{max} is never exceeded: ρ is reset to 1 after ρ_{max} is reached. Once the time limit is reached, the best solution produced is returned.

7.4 Neighborhood structures

Several neighborhoods were developed to explore the search space of the SBVRP, being categorized within three groups: (*i*) neighborhood structures based on classical VRP moves, (*ii*) neighborhood structures based on SBVRP

Algorithm 7.2: Hybrid Algorithm (ILS and LAHC)

Input: LAHC list size l, maximum consecutive rejections m and maximum perturbation level ρ_{max}

SBVRP_Solver(l, m, ρ_{max})

```
1  S* ← S ← initial naive solution
2  ρ ← 1
3  while time limit is not reached do
4      S ← LAHC(S, l, m)
5      if f(S) < f(S*) then
6          S* ← S
7          ρ ← 1 // perturbation level is reset
8      else
9          S ← S* // solution S is rejected and therefore replaced by S*
10     for i = 0 to ρ do
11         N ← selected neighborhood structure
12         S ← random neighbor N(S)
13     ρ ← (ρ mod ρ_max) + 1
14 return S*
```

specific moves and, finally, (*iii*) neighborhood structures based on subproblem optimization. On the one hand, the first two groups generate neighbors by applying move(s) to the solution. These neighborhoods are sampled randomly rather than being fully explored. The neighborhoods based on subproblem optimization, on the other hand, generate neighbors by solving a subproblem. In this case, randomization occurs when generating the subproblem.

All different neighborhoods are considered together as possible approaches in generating new (neighboring) solutions. The probability of selecting each neighborhood in a given iteration is determined by a learning automaton.

Section 7.4.1 begins by introducing the neighborhood size reduction procedure, responsible for pruning away potentially less attractive neighbors. Next, the different neighborhood structures are explained. Classical VRP neighborhoods are discussed in Section 7.4.2, those based on SBVRP specific moves are presented in Section 7.4.3 and those based on subproblem optimization are described in Section 7.4.4. Finally, Section 7.4.5 details the learning automaton.

7.4.1 Neighborhood size reduction

A neighborhood size reduction procedure is applied to limit the number of solution neighbors. This procedure is equivalent to that presented in Chapter 6

(see Section 6.3.2, page 146) and avoids considering solutions in which certain pairs of customers are visited consecutively in one route. The primary motivation behind this approach is that two geographically distant customers tend not to be visited one after the other, and thus analyzing them as consecutive customers in a route likely results in wasted time.

The procedure requires a preprocessing step. First, a list of possible preceding and succeeding customers in the same route is computed for each customer. Both time and capacity constraints are considered. For instance, two train-only customers can never be in the same route since the sum of their capacities exceeds the capacity of a train. Once the list is built, it is sorted according to the cost of traveling either from or to each other customer, whichever is smaller. This cost is computed as the weighted sum of time and distance costs.

Once the ordered lists are built for each customer, neighborhood reduction may be applied. The procedure limits possible customer allocations to the first Γ in each customer list (ignoring the other possibilities). Note that $\Gamma \in \mathbb{Z}^+$ with $\Gamma \leq |C|$. Clearly, no neighborhood reduction is applied when $\Gamma = |C|$.

During ILS perturbations, Γ is always set to $|C|$. This disables neighborhood reduction and may lead to diverse solutions, which is the primary objective of the perturbation. In the other contexts, each neighborhood may have a different value for Γ. Depending on its value and on the instance characteristics, the procedure may cut the optimal solution away from the search space. Hence, setting an appropriate value for Γ is crucial for the proposed algorithm's efficiency.

A neighborhood is defined by both its structure and the value of Γ. Therefore, one neighborhood structure is considered in multiple neighborhoods, each with a different value for the neighborhood reduction parameter Γ. The principle behind this is that the learning algorithm should select the most appropriate neighborhoods, giving very low probabilities for inefficient selections.

7.4.2 Classical neighborhood structures

Neighborhood structures based on four classical VRP moves are considered. Whenever possible, neighborhood reduction (Section 7.4.1) is applied. The moves are as follows:

Relocate move. This move consists of relocating one or more customers into random routes and positions.

Swap move. The *swap move* consists of simply swapping two customers in the solution. The customers may belong to the same or to different routes.

2-opt move. This move is the classical 2-opt move proposed by Croes (1958). It is important to note, however, that when dealing with the SBVRP both routes and sub-routes are considered, with sub-routes being treated as independent routes.

Ejection-chain move. This move is a standard ejection-chain (Glover, 1991, 1996), and consists of re-arranging a chain of consecutive customers in a route. The *size* of the chain is given as a parameter.

7.4.3 Problem-specific neighborhood structures

Five neighborhood structures were developed to explore specific characteristics of the SBVRP. Again, neighborhood reduction (Section 7.4.1) is applied whenever possible. The neighborhood structures are based on the following five moves:

Change swap location move. Given a route that includes a swap location, this move essentially changes the vertex (location) of one of the route's swap locations. Note that moving to certain swap locations may render the solution infeasible due to the limit imposed on route durations. To circumvent this issue, only swap locations close enough (so as to maintain feasibility) are considered. The new vertex is randomly selected from one of these locations.

Convert to route move. This move consists of converting a sub-route into a new (truck) route. The improvement probability of this move is limited, nevertheless it proves useful as a diversification tool.

Add sub-route move. The *add sub-route move* consists of adding a swap location to a train route, resulting in a new sub-route inside the route. Two steps must be executed: (*i*) defining the vertex (location) of the swap location and (*ii*) defining the positions in which the swap location will be added. This move requires all customers within the sub-route to be truck-only or flexible

customers. Two operations are performed at the swap location: (*i*) park and then (*ii*) pickup. Capacity and time constraints must also be checked.

Split to sub-routes move. This move is similar to the *add sub-route* move, but instead of adding one sub-route, it adds two sub-routes at the same swap location, such that each sub-route is conducted with a different swap body. Three operations are defined at the swap location: (*i*) park, (*ii*) swap, and finally (*iii*) pickup.

Merge routes move. This move consists of merging two truck routes into a train route. Initially, two truck routes are selected. Then the swap location closest to both routes is selected and a new route including this swap location is created. Afterwards, the two truck routes are added to the route as sub-routes. As with the split to sub-routes move, three operations are performed at the swap location: (*i*) park, (*ii*) swap, and finally (*iii*) pickup.

7.4.4 Subproblem optimization scheme

Additional neighborhood structures based on a subproblem optimization scheme have been developed. The subproblems require the reallocation of a subset $P \in C$ of the customers while minimizing the solution cost. In other words, subproblems are SBVRP problems in which a subset $\bar{P}$ of customers, $\bar{P} = C \backslash P$, have their routes predefined.

Two questions arise from this approach. Firstly, how many customers may be added to P without negatively impacting on the subproblem computation time? Secondly, which customers should be added to P? Both questions are addressed in this section.

The initial idea of the subproblem optimization was to solve the subproblems to optimality employing an MIP approach. However, only very small subproblems (small P) appear solvable in short runtimes by currently available MIP solvers such as CPLEX and Gurobi. Experiments revealed that heuristic (sub-optimal) solutions for larger subproblems yielded much better final results than optimal solutions for very small problems. The difference is more profound when the runtime limit imposed by the VeRoLog challenge of 600 seconds is considered. Hence, the approach proposed in this chapter does not guarantee optimality of subproblem solutions. They are produced by a straightforward best-fit constructive heuristic, henceforth called *Cheapest Inserter* procedure.

The *Cheapest Inserter* procedure operates as follows. First, P is defined as an ordered set and its customers are removed from the solution. Next, they are sequentially inserted following P's order into the routes and positions incurring the lowest cost increase. This procedure executes $O(|P| \times |\bar{P}|)$ operations in the worst case. When applying the neighborhood reduction presented in Section 7.4.1, only the Γ closest customers are considered by the *Cheapest Inserter*. This speeds up the procedure by considerably reducing the number of comparisons. The size of P therefore does not prevent the algorithm from quickly producing a solution to the subproblem.

Six strategies have been developed for customer selection in a subproblem. Each strategy requires the maximum number of customers (η) as a parameter in addition to the value for the neighborhood size reduction Γ, applied within the *Cheapest Inserter*. Note that all strategies have a certain degree of randomization.

Ruin and recreate strategy. This strategy randomly selects η customers and adds them to P.

Remove chains strategy. In this strategy, small chains of consecutive customers are added to P. The total number of customers is bounded by the parameter η, while the number of chains is bounded by the parameter k. The customers may be added to P in three possible orders: (i) random, (ii) original, or (iii) inverse original route. One of the three sorting criteria is randomly selected.

Remove route strategy. All customers from one or more randomly selected routes are added to P. Similar to the previous strategy, customers may be disposed in three possible orders: (i) random, (ii) route, and (iii) inverse route.

Remove sub-routes strategy. This strategy is similar to the *remove routes strategy* except that, instead of routes, only sub-routes are considered.

Limited ruin and recreate strategy. This strategy is similar to the *ruin and recreate strategy*, but only routes close to one another are considered when selecting customers. Note that the parameter Γ (Section 7.4.1) is employed when defining close routes, meaning that only routes containing geographically-nearby customers are selected. This strategy has two parameters: η, which indicates the number of customers to be added to P, and k, which indicates

the maximum number of routes to be considered.

Ruin and randomized recreate strategy. This strategy is similar to the *ruin and recreate strategy* except for the differing behavior of the *Cheapest Inserter*. Rather than always selecting the cheapest insertion, the algorithm may select another insertion based on a probability given by a Heuristic-Biased Stochastic Sampling (HBSS) (Bresina and Bresina, 1996). Any customer may potentially be chosen, with the best ones having a much higher probability of selection. The chances of selecting the i-th cheaper insertion is given by $f(i) = e^{-i}$.

7.4.5 Learning automaton

A learning automaton (Narendra and Thathachar, 1989) provides a simple reinforcement learning approach commonly used in single state environments. It maintains and updates a probability distribution γ for all possible actions.

In the proposed algorithm, the learning automaton determines the probabilities of selecting each of the available neighborhoods. Therefore, for each neighborhood k a probability $\gamma_{k,e}$ is assigned to each event (or iteration) e. The value of $\gamma_{k,e}$ is updated according to the feedback provided by the local search algorithm from the previous event, $e-1$. Generally, the probabilities are updated according to Equations (7.1) and (7.2). Equation (7.1) is applied to update the probability of selecting the current neighborhood while Equation (7.2) is applied to update the probabilities of the remaining neighborhoods.

$$\gamma_{k,e+1} = \gamma_{k,e} + \alpha^{+}\beta_e(1-\gamma_{k,e}) - \alpha^{-}(1-\beta_e)\gamma_{k,e} \tag{7.1}$$

$$\gamma_{k,e+1} = \gamma_{k,e} - \alpha^{+}\beta_e\gamma_{k,e} + \alpha^{-}(1-\beta_e)\left(\frac{1}{|\mathcal{N}|-1} - \gamma_{k,e}\right) \tag{7.2}$$

In Equations (7.1) and (7.2), constants $\alpha^{+} \in [0,1]$ and $\alpha^{-} \in [0,1]$ are the reward and penalty parameters, respectively. $\beta_e \in [0,1]$ indicates the improvement gained by event e and, finally, $|\mathcal{N}|$ is the number of available neighborhoods. The objective of this update scheme is to increase the probability of a neighborhood in case of success and to decrease it in case of failure. A linear reward-penalty (L_{R-P}) strategy is employed, meaning that the reward and penalty parameters are equal: $\alpha^{+} = \alpha^{-}$. The improvement gain calculation is also simplified: β_e is defined such that $\beta_e \in \{0,1\}$. It assumes a value of one if successful and

zero in case of failure. This approach was shown to be successful for improving heuristic search algorithms by Wauters (2012).

7.5 Computational Experiments

The proposed algorithm was coded in Java 1.8 and executed on Intel(R) Xeon E5-2680v3 CPU @ 2.5GHz computers with 64GB of RAM memory running Red Hat Enterprise Linux ComputeNode 6.5. All computational experiments were performed according to the VeRoLog challenge rules: the running time was limited to 600 seconds per instance and at most 4 CPU cores were used per execution.

The LAHC list size and the maximum number of consecutively rejected neighbors were manually tuned. After a significant number of experiments considering different values for the parameters, four sets were selected based on their performance: (*i*) l=5000 and m=5000; (*ii*) l=12500 and m=12500; (*iii*) l=25000 and m=25000; (*iv*) l=25000 and m=50000. Each set is considered by the solver in a different thread.

This section begins by presenting the instances provided by the VeRoLog Solver Challenge in Section 7.5.1. Section 7.5.2 presents a study comparing the impact of the different neighborhood groups. Section 7.5.3 shows the behavior induced by the learning automata by presenting the evolution of certain neighborhood selection probabilities. The results obtained for the VeRoLog datasets are presented in Section 7.5.4 and, finally, Section 7.5.5 introduces new SBVRP instances aimed at stimulating future research.

7.5.1 VeRoLog challenge datasets

In total, six instance sets have been published by the VeRoLoG challenge committee: *small*, *medium*, *large*, *presel*, *final* and *final_random* datasets. The first three datasets were introduced prior to the challenge. The *presel* dataset was used to evaluate algorithms during the pre-selection phase, while the remaining two datasets, *final* and *final_random*, were considered to determine the challenge winner. Each dataset contains three instances: one in which all customers can be visited by trains (*all_with*), one containing all customer types (*normal*) and one in which no customer can be visited by trains (*all_without*). Table 7.2

shows the characteristics of these instances. The nomenclature presented during Section 7.1.1 is employed and $\sum q_i$ represents the sum of the demands of all customers $i \in C$. Note that the number of train-only customers ($|C_3|$) is either one or zero. All instance files are available online[2].

Table 7.2: Characteristics of the VeRoLog challenge instances

Instance	$\|C\|$	$\|C_1\|$	$\|C_2\|$	$\|C_3\|$	$\|S\|$	T	Q	$\sum q_i$
small_all_with	57	0	56	1	20	8h	500	8445
small_normal	57	15	41	1	20	8h	500	8445
small_all_without	57	57	0	0	20	8h	500	7950
medium_all_with	206	0	206	0	41	11h	1000	24790
medium_normal	206	20	186	0	41	11h	1000	24790
medium_all_without	206	206	0	0	41	11h	1000	24000
large_all_with	548	0	548	0	99	11h	1000	65080
large_normal	548	50	498	0	99	11h	1000	65080
large_all_without	548	548	0	0	99	11h	1000	63090
presel_all_with	550	0	550	0	101	11h	1000	58845
presel_normal	550	50	500	0	101	11h	1000	58845
presel_all_without	550	550	0	0	101	11h	1000	58845
final_normal_all_with	549	0	549	0	102	18h	1000	58785
final_normal	549	50	499	0	102	18h	1000	58785
final_normal_all_without	549	549	0	0	102	18h	1000	58785
final_rand_all_with	549	0	549	0	102	18h	1000	327388
final_rand	549	50	499	0	102	18h	1000	331239
final_rand_all_without	549	549	0	0	102	18h	1000	325815

7.5.2 Neighborhood groups

Experiments isolating each neighborhood group (Section 7.4) were performed to evaluate their impact on the final solution quality, thereby enabling one to assess the importance of the subproblem optimization scheme. Let $\mathcal{N}$ be the set containing all developed neighborhoods. These neighborhoods are divided into three subsets: $\mathcal{N}_1$ represents the classical VRP neighborhoods, $\mathcal{N}_2$ the problem-specific neighborhoods, and $\mathcal{N}_3$ the subproblem optimization neighborhoods. Table 7.3 compares the results obtained by the algorithm considering different combinations of neighborhood subgroups. The average gap to the solutions obtained with all neighborhoods are presented. The values are aggregated by instance type: all_with, normal and all_without. Each line shows the average

[2]The VeRoLog Challenge organizers gently provided the instance files and permitted us to publish them at http://benchmark.gent.cs.kuleuven.be/sbvrp

gap for 60 executions, 10 for each instance. A runtime limit of 600 seconds was imposed for all executions. The overall average gap is also presented providing a rough ranking with regard to the different strategies.

Table 7.3: Average gap obtained by employing different neighborhood group combinations

Neighborhoods	Results per instance group			Average
	all_with	normal	all_without	
$\mathcal{N}_1 \cup \mathcal{N}_2 \cup \mathcal{N}_3$	0.00%	0.00%	0.00%	0.00%
$\mathcal{N}_1 \cup \mathcal{N}_2$	0.90%	2.29%	9.06%	4.08%
$\mathcal{N}_1 \cup \mathcal{N}_3$	-0.06%	1.21%	9.45%	3.54%
$\mathcal{N}_2 \cup \mathcal{N}_3$	0.15%	0.21%	0.42%	0.26%
$\mathcal{N}_1$	0.76%	2.09%	9.72%	4.19%
$\mathcal{N}_2$	66.90%	68.81%	74.41%	70.04%
$\mathcal{N}_3$	0.12%	1.56%	10.21%	3.97%

Table 7.3 enables one to draw some interesting conclusions. Firstly, considering the entire neighborhood set appears to be the best strategy on average. Secondly, it is clear that the subset of problem-specific neighborhoods is incapable of independently producing good quality solutions, as expected. Thirdly, the characteristic of the instances largely impact the relationship between the neighborhoods. Note how excluding the problem-specific neighborhoods was beneficial when dealing with "all_with" instances. These instances enable visiting all customers with trains and therefore the swap locations tend to be less frequently employed, since swap operations become optional. Contrastingly, huge gaps were obtained when the problem-specific neighborhood set was ignored for the "all_without" instances. These instances prohibit visiting customers with trains. Swap locations consequently become essential for storing bodies of trains, as without them only trucks may be used. Finally, the table reveals how including neighborhoods employing decomposition and performing subproblem optimization were on average more beneficial than including classical VRP neighborhoods in almost all situations.

7.5.3 Learning automaton and neighborhoods

The learning automaton algorithm presented in Section 7.4.5 has been employed in two different contexts: first as an offline tuning tool to assess the initial probabilities of selecting each neighborhood, and second as an online fine-tuning tool to update these probabilities.

The learning automaton has one parameter: reward-penalty α, also known as the *learning rate*. This parameter defines how much the probabilities may change in each iteration. For offline tuning the reward-penalty was set to $\alpha = 10^{-5}$ and the resulting probabilities were employed to help determining initial neighborhood probabilities. Two main criteria were considered for each neighborhood: its time complexity, given by the number of operations executed to generate a neighbor, and the average probability given by the learning automaton after several rounds of LAHC. The principle behind this is that fast and relatively efficient neighborhoods should initially have higher chances of selection than slow ones. Due to very low probabilities resulting from the offline tuning phase, some neighborhoods were dropped.

Table 7.4 shows the selected neighborhoods and their respective *importance*, given by v. The probability of each neighborhood is given by its importance divided by the sum of all neighborhoods' importance. The initial importance values considered are $v \in \{1, 5, 10, 20, 50, 100\}$. The table also details the values for parameters η, k and Γ, when required by a neighborhood. If the value for Γ is hidden, then no neighborhood reduction is applied ($\Gamma = |C|$).

Table 7.4: Initial probabilities and considered neighborhoods

Neighborhood	v
Relocate(η=1)	50
Relocate(η=2)	100
Relocate(η=3)	50
Relocate(η=4)	10
Swap(Γ=50)	5
TwoOpt()	10
ChangeSwapLocation()	10
ConvertToRoute()	10
AddSubRoute(Γ=50)	10
SplitToSubRoute()	10
MergeRoutes()	10
EjectionChain(η=3, Γ=50)	20
EjectionChain(η=4, Γ=50)	20
EjectionChain(η=5, Γ=50)	20
EjectionChain(η=10, Γ=50)	20
EjectionChain(η=15, Γ=50)	10
EjectionChain(η=20, Γ=75)	10
EjectionChain(η=25, Γ=75)	10
EjectionChain(η=30, Γ=75)	10
RuinRecreate(η=1, Γ=50)	5
RuinRecreate(η=2, Γ=50)	5
RuinRecreate(η=3, Γ=50)	5
RuinRecreate(η=4, Γ=50)	5
RuinRecreate(η=5, Γ=50)	5

Neighborhood	v
RuinRecreate(10, Γ=50)	5
RuinRecreate(15, Γ=50)	5
RuinRecreate(20, Γ=50)	5
RuinRecreate(25, Γ=50)	5
RuinRecreate(35, Γ=50)	5
RuinRecreate(50, Γ=50)	5
RemoveRoute(Γ=150)	1
RemoveSubRoute()	10
RemoveSubRoute(Γ=150)	10
RemoveChains(η=1, k=100, Γ=50)	10
RemoveChains(η=2, k=100, Γ=50)	10
RemoveChains(η=3, k=100, Γ=50)	10
RemoveChains(η=4, k=100, Γ=50)	10
RemoveChains(η=5, k=100, Γ=50)	10
RemoveChains(η=6, k=100, Γ=50)	10
RemoveChains(η=7, k=100, Γ=50)	10
RemoveChains(η=8, k=100, Γ=50)	10
LimRuinRecreate(η=4, k=4, Γ=50)	10
LimRuinRecreate(η=2, k=5, Γ=50)	10
LimRuinRecreate(η=5, k=2, Γ=50)	10
RuinRandRecreate(η=1)	5
RuinRandRecreate(η=2)	5
RuinRandRecreate(η=3)	5
RuinRandRecreate(η=4)	5

The online fine-tuning phase, present in the final algorithm, considers a lower reward-penalty $\alpha = 10^{-6}$. The goal is to avoid drastic changes in the

probabilities while still fine-tuning them. After each execution of the LAHC algorithm, the probabilities are reset to their initial values.

7.5.4 Results

Table 7.5 presents the average and best results from 20 algorithm executions for all instances published by the First VeRoLog Challenge. These results are compared against the best results reported in the literature:

HG2014 : Huber and Geiger (2014) executed their approach 30 times on an Intel Xeon X5650 2.66 GHz;

Lum2015 : Lum et al. (2015) reported the best results obtained on a 2012 MacBook Air (precise computer unspecified);

Absi2017 : Absi et al. (2017) employed an Intel Xeon 2.8 GHz and ran their relax-and-repair heuristic 10 times;

THUJG2017 : Todosijević et al. (2017) experimented both their Parallel and Sequential GVNS on an Intel i7-4900MQ 2.80 GHz;

MB2017 : Miranda-Bront et al. (2017) executed their algorithm 10 times on an Intel Core i5-3320M;

HG2017 : Huber and Geiger (2017) utilized an Intel Xeon X5650 2.66 GHz. They report the best solutions out of a very large set of experiments considering different parameters and algorithm versions. Each of these experiments consists of 30 algorithm executions.

With the exception of Absi et al. (2017), who included results from 20-minute runs, runtimes were limited to 10 minutes by all other authors. Table 7.5 highlights how the algorithm proposed here clearly outperforms all approaches described in the literature. For the majority of the instances, the average over all runs provides a better value than the best solution generated by other algorithms. The produced solutions are available online[3].

Figure 7.4 presents boxplot graphs illustrating the proposed approach's performance for all instance groups. The dashed lines indicate previous best solutions. For 12 out of 18 instances, every solution produced by the proposed algorithm are at least as good as the previous best solution, while for 9 of these instances every solution lies below the dashed line, meaning all solutions improve upon the previous best. This further supports the claim that the proposed approach outperforms all methods in the literature. Figure 6 also enables one to

[3]http://benchmark.gent.cs.kuleuven.be/sbvrp

Table 7.5: Results for VeRoLog challenge instances

Instance	Proposed Algorithm		*HG2014*	*Lum2015*	*Absi2017*	*THUJG2017*	*MB2017*	*HG2017*
	Average	Best						
small_all_with	4716.50	4715.78	4730.92	4873.05	4716.58	4731.02	4728.93	4716.58
small_normal	4797.69	4797.55	4804.97	4959.00	4802.38	4847.63	4806.97	4797.55
small_all_without	4858.53	4839.64	4839.64	5356.36	4981.70	5249.18	4855.62	4839.64
medium_all_with	7745.80	7708.63	7755.43	8335.57	7763.13	7754.39	7847.30	7734.61
medium_normal	7818.29	7803.54	7817.83	8297.25	7810.93	7834.78	7942.22	7795.98
medium_all_without	8007.11	7980.01	8045.47	8628.37	8058.89	8382.80	8169.69	7982.76
large_all_with	19851.04	19806.75	20215.26	21317.00	20495.70	20066.40	20516.70	20058.99
large_normal	20039.92	19985.88	20524.54	22051.40	20760.30	20496.40	20738.50	20298.82
large_all_without	20728.43	20640.72	21255.51	22419.40	21580.60	22310.60	21522.50	21003.51
presel_all_with	24475.56	24402.67	25072.36	26658.10	25021.70	24965.10	25573.20	24767.63
presel_normal	24859.76	24800.16	25425.85	26712.40	25529.50	25443.20	25894.40	25069.86
presel_all_without	25510.38	25448.55	25835.85	26712.40	25975.50	26515.90	26524.50	25719.19
final_all_with	33118.08	33014.03	-	-	-	-	34997.90	33753.48
final_normal	34254.51	33927.04	-	-	-	-	36305.80	34649.78
final_all_without	36574.23	36347.28	-	-	-	-	38826.20	36814.84
final_random_all_with	129356.15	129049.18	-	-	-	-	131445.00	129257.44
final_random	132758.74	132341.83	-	-	-	-	135509.00	132295.68
final_random_all_without	144588.22	144331.84	-	-	-	-	152587.00	144725.57
Gap from best result:	0.33%	0.01%	1.36%	6.83%	1.97%	2.93%	3.28%	0.81%

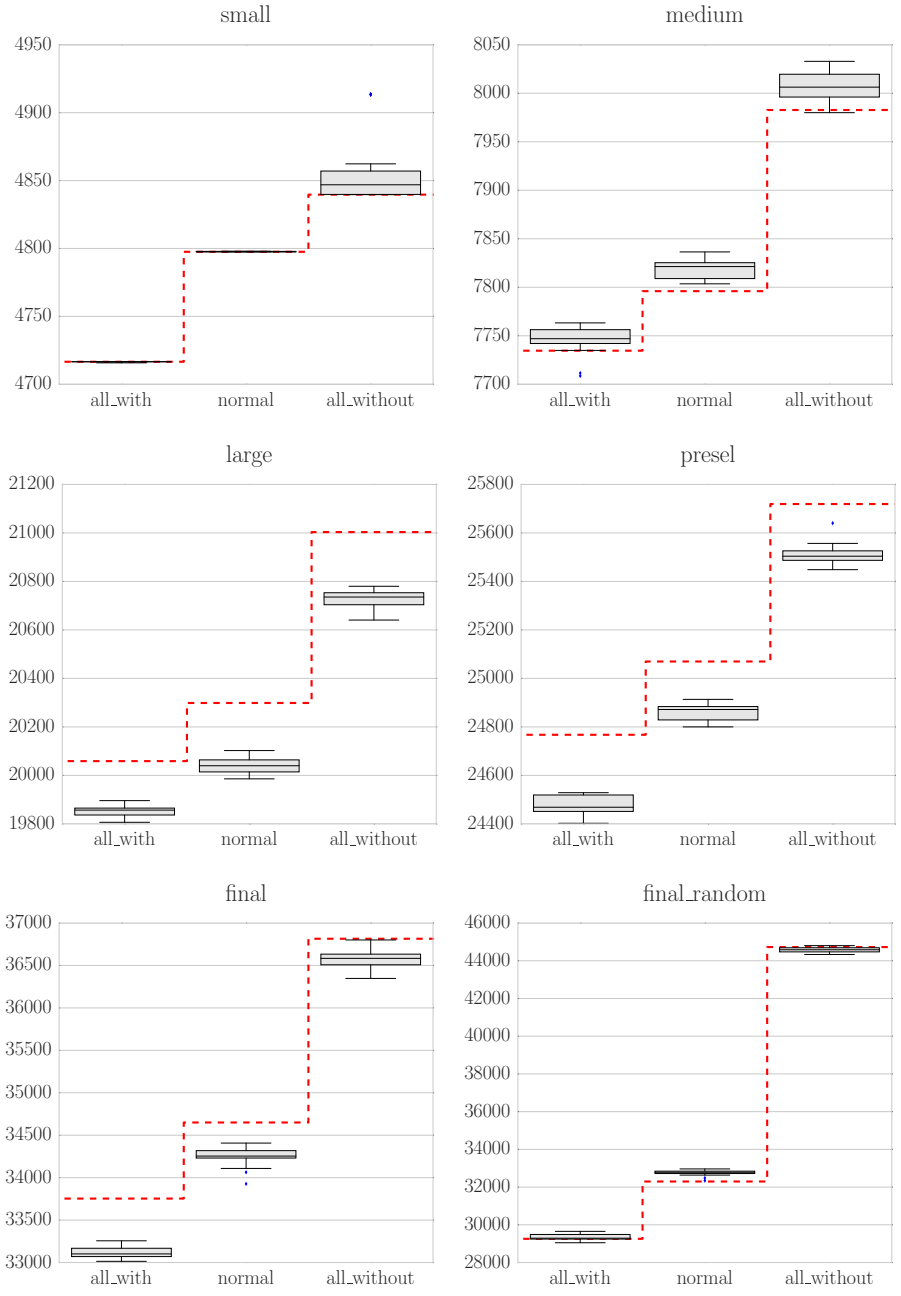


Figure 7.4: Boxplots comparing the solutions obtained with the proposed approach and the best results reported in the literature for all instances

conclude that the algorithm is very robust. In fact, the maximum gap between two solutions obtained by the algorithm after 600 seconds is 1.5% (instance *small_all_without*: costs of 4839.64 and 4913.39). The average gap to the best solution when considering all 360 solutions produced for the 18 instances is only 0.33%.

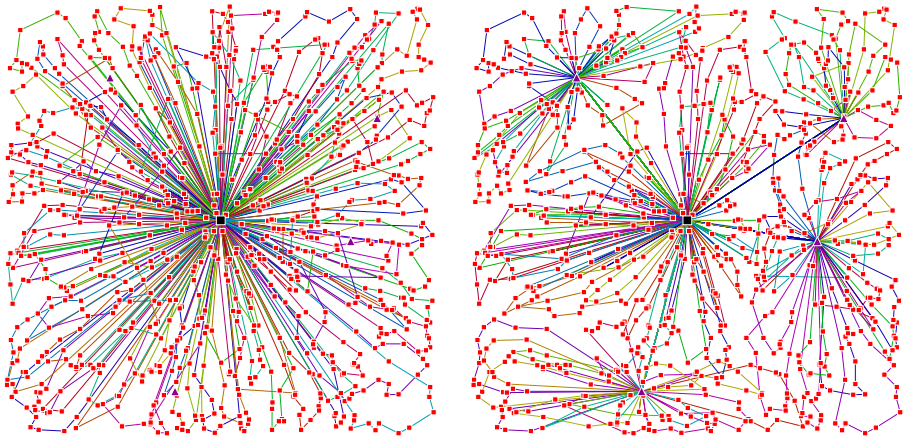
7.5.5 Additional instances

New instances are proposed to stimulate additional research in the field, varying many characteristics in contrast to those proposed by the VeRoLog Solver Challenge. These new instances are generated from Uchoa et al. (2017)'s benchmark CVRP instances which are produced by distributing customers within a 1000×1000 grid in accordance with various strategies. Uchoa et al. (2017)'s instances are modified via the addition of either 4, 20 or 100 swap locations and the definition of costs for trains and trucks. Swap locations are generated by first selecting a random customer before randomly selecting a point on a circle of radius 100 from the selected customer's location. If this point lies within the CVRP's 1000×1000 grid then the point is accepted as a swap location. This process continues until the required number of swap locations have been added to the instance. The cost of a truck is fixed at one per distance unit. The cost of a train is given by c per distance unit, with $c \in \{1.2, 1.4, 1.6\}$. Customers are distributed within truck-only and flexible customers, such that $|C_1| = \lfloor k \times |C| \rfloor$ and $|C_2| = |C| - |C_1|$, where $|C_1|$ and $|C_2|$ are the number of truck-only and flexible customers, respectively. No train-only customers are generated. Moreover, swap body capacity simply corresponds to a vehicle's capacity in the CVRP instances.

Note that these new instances have significant differences from those proposed during the VeRoLog Solver Challenge. The instances proposed here have symmetric distances, impose no limits on route durations and do not include service times at customers or operation times at swap locations.

An example[4] comparing CVRP and SBVRP is shown in Figure 7.5. Instance sbvrp-n936-s4 adds four swap locations to instance X-n936-k151. Customers may be visited only by trucks, and the train's cost is set to 120% of the truck's cost. Note how the four swap locations are utilized extensively within this

[4]For more examples, the reader is directed to http://benchmark.gent.cs.kuleuven.be/sbvrp, where solution files and the software for visualizing them are provided.



(a) CVRP solution for X-n936-k151 (b) SBVRP solution for sbvrp-n936-s4

Figure 7.5: Example of CVRP and SBVRP solutions

example, resulting in a 2.5% reduction in the objective function value (129633.4 against 132926.0).

Table 7.6 presents the characteristics of the generated instances (including characteristics of the original CVRP instances). The columns indicate:

- $|C|$: number of customers;
- Dep : placement of the depot, which can be at the center (C), at a random position (R) or at the corner of the grid (E);
- $Cust$: placement of the customers, which may be random (R), in n clusters (C(n)), or "randomly-clustered" (RC(n));
- Dem : demand distribution, which may be completely unitary (U), uniformly distributed in a range between n_1 and n_2 (n_1-n_2), depending on quadrant (Q) or with "many small and few large values" (SL);
- Q : vehicle capacity (or capacity of each swap body);
- $|S|$: number of swap locations;
- c : cost of a train (trucks have cost fixed to 1.0 per distance unit);
- k : coefficient of customers categorization, $|C_1| = \lfloor k \times |C| \rfloor$ and $|C_2| = |C| - |C_1|$.

Detailed information concerning the characteristics of the original CVRP instances may be found within Uchoa et al. (2017). Table 7.6 also presents the average and best results out of 20 algorithm executions with a runtime limit of 10 minutes and 1 hour, respectively. Note that the proposed algorithm was executed using the same parameters as those employed in the experiments of Section 7.5.4.

Table 7.6: Characteristics of the proposed instances and computational results for 10min and 1h

#	Instance	Characteristics (Uchoa et al., 2017)					SB-VRP			Results (10min)		Results (1h)	
		$\|C\|$	Dep	$Cust$	Dem	Q	$\|S\|$	c	k	Avg.	Best	Avg.	Best
1	sbvrp-n101-s100	100	R	RC(7)	1-100	206	100	1.2	0.9	22540.70	22494.80	22506.68	22492.80
2	sbvrp-n106-s20	105	E	C(3)	50-100	600	20	1.6	1.0	23661.37	23615.40	23643.09	23629.60
3	sbvrp-n110-s4	109	C	R	5-10	66	4	1.4	0.5	15920.79	15891.60	15897.86	15891.60
4	sbvrp-n115-s100	114	C	R	SL	169	100	1.4	0.1	12302.95	12191.00	12223.85	12186.20
5	sbvrp-n120-s4	119	E	RC(8)	U	21	4	1.6	0.0	13503.40	13484.20	13486.12	13390.60
6	sbvrp-n125-s20	124	R	C(5)	Q	188	20	1.2	0.1	36515.15	36461.60	36476.56	36439.20
7	sbvrp-n129-s4	128	E	RC(8)	1-10	39	4	1.6	1.0	30469.39	30386.60	30408.99	30355.40
8	sbvrp-n134-s100	133	R	C(4)	Q	643	100	1.4	0.9	9960.20	9896.60	9918.03	9890.40
9	sbvrp-n139-s20	138	C	R	5-10	106	20	1.2	0.0	12679.34	12647.60	12660.45	12647.60
10	sbvrp-n143-s100	142	E	R	1-100	1190	100	1.4	0.5	14824.00	14736.40	14754.40	14736.40
11	sbvrp-n148-s4	147	R	RC(7)	1-10	18	4	1.6	0.9	45448.99	44096.80	44030.44	43252.00
12	sbvrp-n153-s20	152	C	C(3)	SL	144	20	1.2	0.0	14815.98	14770.80	14797.89	14773.20
13	sbvrp-n157-s20	156	R	C(3)	U	12	20	1.6	0.1	15477.34	15426.80	15417.12	15376.00
14	sbvrp-n162-s100	161	C	RC(8)	50-100	1174	100	1.4	0.5	13807.98	13719.40	13745.48	13719.40
15	sbvrp-n167-s4	166	E	R	5-10	133	4	1.2	1.0	19815.63	19767.20	19777.78	19695.20
16	sbvrp-n172-s100	171	C	RC(5)	Q	161	100	1.2	0.0	30032.52	30024.00	30026.21	30024.00
17	sbvrp-n176-s4	175	E	R	SL	142	4	1.6	0.9	49130.29	46707.60	46866.35	45484.00
18	sbvrp-n181-s20	180	R	C(6)	U	8	20	1.4	0.5	22143.19	22091.00	22104.73	22056.80
19	sbvrp-n186-s4	185	R	R	50-100	974	4	1.2	1.0	25651.45	25553.40	25491.21	25131.80
20	sbvrp-n190-s100	189	E	C(3)	1-10	138	100	1.6	0.1	15463.47	15359.40	15372.54	15301.00
21	sbvrp-n195-s20	194	C	RC(5)	1-100	181	20	1.4	0.1	37900.65	37695.80	37658.32	37471.20
22	sbvrp-n200-s100	199	R	C(8)	Q	402	100	1.6	1.0	50723.48	50592.40	50579.62	50427.00
23	sbvrp-n204-s4	203	C	RC(6)	50-100	836	4	1.2	0.5	20133.14	20068.60	20079.93	20043.40
24	sbvrp-n209-s20	208	E	R	5-10	101	20	1.4	0.0	26738.25	26658.80	26668.98	26622.40
25	sbvrp-n214-s100	213	C	C(4)	1-100	944	100	1.6	0.9	10822.29	10758.00	10763.21	10704.80
26	sbvrp-n219-s4	218	E	R	U	3	4	1.2	0.9	101295.38	100729.00	100978.25	100649.00
27	sbvrp-n223-s20	222	R	RC(5)	1-10	37	20	1.4	0.5	36704.39	36459.00	36516.64	36417.00
28	sbvrp-n228-s20	227	R	C(8)	SL	154	20	1.2	1.0	22649.59	22431.20	22515.39	22427.60
29	sbvrp-n233-s4	232	C	RC(7)	Q	631	4	1.4	0.1	19258.70	18954.60	19107.78	18941.80
30	sbvrp-n237-s100	236	E	R	U	18	100	1.6	0.0	25454.56	25234.20	25279.33	25205.40
31	sbvrp-n242-s20	241	E	R	1-10	28	20	1.2	1.0	62482.20	62171.20	62266.32	62108.60
32	sbvrp-n247-s100	246	C	C(4)	SL	134	100	1.4	0.1	28221.79	28058.60	28058.08	27911.40
33	sbvrp-n251-s4	250	R	RC(3)	5-10	69	4	1.6	0.9	41014.91	37218.60	37626.60	37010.80

(continued on next page)

Table 7.6 continued: Characteristics of the proposed instances and computational results for 10min and 1h

#	Instance	Characteristics (Uchoa et al., 2017)					SB-VRP			Results (10min)		Results (1h)	
		$\|C\|$	Dep	$Cust$	Dem	Q	$\|S\|$	c	k	Avg.	Best	Avg.	Best
34	sbvrp-n256-s4	255	C	C(8)	50-100	1225	4	1.6	0.5	22023.10	21687.80	21639.21	19426.80
35	sbvrp-n261-s100	260	E	R	1-100	1081	100	1.2	0.0	21136.92	20986.60	21041.61	20986.80
36	sbvrp-n266-s20	265	R	RC(6)	5-10	35	20	1.4	1.0	63312.60	63079.20	63118.29	62917.80
37	sbvrp-n270-s20	269	C	RC(5)	50-100	585	20	1.2	0.5	31361.16	31195.00	31243.23	31118.80
38	sbvrp-n275-s4	274	R	C(3)	U	10	4	1.4	0.0	18940.07	18888.80	18903.71	18871.80
39	sbvrp-n280-s100	279	E	R	SL	192	100	1.6	0.9	32197.82	31886.20	31959.69	31722.20
40	sbvrp-n284-s100	283	R	C(8)	1-10	109	100	1.6	0.1	19381.85	19202.60	19235.35	19110.60
41	sbvrp-n289-s20	288	E	RC(7)	Q	267	20	1.4	0.0	71748.75	71639.00	71542.48	71434.60
42	sbvrp-n294-s4	293	C	R	1-100	285	4	1.2	0.9	50338.40	47632.80	47703.31	45380.20
43	sbvrp-n298-s100	297	R	R	1-10	55	100	1.4	1.0	31715.41	31502.60	31485.55	31246.20
44	sbvrp-n303-s4	302	C	C(8)	1-100	794	4	1.6	0.1	22832.06	22711.40	22717.58	22645.20
45	sbvrp-n308-s20	307	E	RC(6)	SL	246	20	1.2	0.5	23563.78	23324.00	23185.37	22928.60
46	sbvrp-n313-s4	312	R	RC(3)	Q	248	4	1.2	0.5	75694.01	73087.00	72755.57	72562.20
47	sbvrp-n317-s100	316	E	C(4)	U	6	100	1.4	0.0	58438.25	58312.80	58380.68	58279.40
48	sbvrp-n322-s20	321	C	R	50-100	868	20	1.6	0.1	31153.38	30819.20	30791.26	30634.00
49	sbvrp-n327-s4	326	R	RC(7)	5-10	128	4	1.2	1.0	26987.69	26801.80	26763.68	26489.60
50	sbvrp-n331-s100	330	E	R	U	23	100	1.4	0.9	28513.69	28260.20	28105.76	27788.20
51	sbvrp-n336-s20	335	E	R	Q	203	20	1.6	0.5	123044.32	122533.40	122361.76	121921.00
52	sbvrp-n344-s4	343	C	RC(7)	5-10	61	4	1.2	0.9	45302.97	41666.20	43216.76	40842.00
53	sbvrp-n351-s20	350	C	C(3)	1-100	436	20	1.6	1.0	26112.07	26021.00	26010.77	25919.00
54	sbvrp-n359-s100	358	E	RC(7)	1-10	68	100	1.4	0.0	42537.92	42286.60	42303.67	41983.60
55	sbvrp-n367-s20	366	R	C(4)	SL	218	20	1.6	0.1	22322.04	22101.60	21996.53	21656.80
56	sbvrp-n376-s100	375	E	R	U	4	100	1.4	0.1	111893.74	111476.20	111517.40	111264.00
57	sbvrp-n384-s4	383	R	R	50-100	564	4	1.2	0.0	45656.69	45470.40	45441.98	45304.80
58	sbvrp-n393-s100	392	C	RC(5)	5-10	78	100	1.6	1.0	36832.35	36519.40	36552.43	36286.60
59	sbvrp-n401-s20	400	E	C(6)	Q	745	20	1.4	0.9	53737.96	53452.00	53495.55	53338.80
60	sbvrp-n411-s4	410	R	C(5)	SL	216	4	1.2	0.5	19278.39	18803.60	18770.09	18603.60
61	sbvrp-n420-s4	419	C	RC(3)	1-10	18	4	1.2	0.5	95750.59	93630.60	91132.09	90620.80
62	sbvrp-n429-s100	428	R	R	50-100	536	100	1.4	0.9	56865.53	56636.00	56494.73	56307.80
63	sbvrp-n439-s20	438	C	RC(8)	U	12	20	1.6	0.0	35549.16	35016.40	35294.73	35009.60
64	sbvrp-n449-s100	448	E	R	1-100	777	100	1.2	0.1	41953.47	41570.80	41412.53	40982.20
65	sbvrp-n459-s20	458	C	C(4)	Q	1106	20	1.4	1.0	25046.39	24685.40	24520.72	24187.20
66	sbvrp-n469-s4	468	E	R	50-100	256	4	1.6	0.1	190409.37	189868.80	189656.37	189426.60
67	sbvrp-n480-s4	479	R	C(8)	5-10	52	4	1.6	0.5	97438.81	89458.20	87876.17	86695.20

(continued on next page)

Table 7.6 continued: Characteristics of the proposed instances and computational results for 10min and 1h

#	Instance	Characteristics (Uchoa et al., 2017)					SB-VRP			Results (10min)		Results (1h)	
		$\|C\|$	Dep	$Cust$	Dem	Q	$\|S\|$	c	k	Avg.	Best	Avg.	Best
68	sbvrp-n491-s20	490	R	RC(6)	1-100	428	20	1.2	0.0	46023.17	45883.40	45806.53	45679.60
69	sbvrp-n502-s100	501	E	C(3)	U	13	100	1.4	1.0	54480.57	54261.80	54327.01	54200.00
70	sbvrp-n513-s100	512	C	RC(4)	1-10	142	100	1.6	0.9	24686.73	24472.40	24460.20	24326.80
71	sbvrp-n524-s4	523	R	R	SL	125	4	1.4	0.5	133072.77	131259.60	129852.68	128261.40
72	sbvrp-n536-s20	535	C	C(7)	Q	371	20	1.2	0.9	71373.16	71191.00	71130.96	70945.80
73	sbvrp-n548-s100	547	E	R	U	11	100	1.2	0.1	63100.78	62789.20	62610.65	62343.00
74	sbvrp-n561-s4	560	C	RC(7)	1-10	74	4	1.6	1.0	52142.10	51781.60	51764.28	51511.60
75	sbvrp-n573-s20	572	E	C(3)	SL	210	20	1.4	0.0	39728.09	39521.00	39515.33	39401.00
76	sbvrp-n586-s20	585	R	RC(4)	5-10	28	20	1.2	0.9	145081.50	144678.00	144430.92	144269.80
77	sbvrp-n599-s4	598	R	R	50-100	487	4	1.6	0.0	96508.84	96209.40	95975.48	95793.60
78	sbvrp-n613-s100	612	C	R	1-100	523	100	1.4	1.0	54242.47	53891.00	53516.76	53113.40
79	sbvrp-n627-s4	626	E	C(5)	5-10	110	4	1.6	0.1	60551.98	59217.40	58944.11	58026.20
80	sbvrp-n641-s20	640	E	RC(8)	50-100	1381	20	1.4	0.5	56985.91	56630.00	56207.74	55939.40
81	sbvrp-n655-s100	654	C	C(4)	U	5	100	1.2	0.5	73269.46	73186.60	72968.71	72824.00
82	sbvrp-n670-s4	669	R	R	SL	129	4	1.2	0.1	98832.02	98178.20	98238.34	97818.20
83	sbvrp-n685-s20	684	C	RC(6)	Q	408	20	1.6	0.0	64405.40	63970.60	63740.19	63284.20
84	sbvrp-n701-s100	700	E	RC(7)	1-10	87	100	1.4	0.9	69105.02	68307.80	68172.87	67454.60
85	sbvrp-n716-s20	715	R	C(3)	1-100	1007	20	1.4	1.0	39078.70	38738.60	38450.13	38168.20
86	sbvrp-n733-s4	732	C	R	1-10	25	4	1.6	0.9	155822.25	141618.40	146352.27	135869.80
87	sbvrp-n749-s100	748	R	C(8)	1-100	396	100	1.2	0.1	56116.02	55881.00	55723.82	55502.80
88	sbvrp-n766-s100	765	E	RC(7)	SL	166	100	1.6	1.0	106061.99	105462.80	104980.10	104365.80
89	sbvrp-n783-s4	782	R	R	Q	832	4	1.4	0.0	61655.59	61231.80	60879.23	60608.80
90	sbvrp-n801-s20	800	E	R	U	20	20	1.2	0.5	61576.71	60632.80	60179.16	59891.00
91	sbvrp-n819-s100	818	C	C(6)	50-100	358	100	1.4	0.5	125217.12	124521.60	124092.90	123375.80
92	sbvrp-n837-s4	836	R	RC(7)	5-10	44	4	1.2	0.1	134346.01	133712.00	132976.66	132397.20
93	sbvrp-n856-s20	855	C	RC(3)	U	9	20	1.6	1.0	88029.31	87743.80	86955.92	85840.20
94	sbvrp-n876-s20	875	E	C(5)	1-100	764	20	1.6	0.9	90739.96	90187.60	89977.46	89734.80
95	sbvrp-n895-s4	894	R	R	50-100	1816	4	1.2	0.0	42107.46	41907.60	41747.53	41656.80
96	sbvrp-n916-s100	915	E	RC(6)	5-10	33	100	1.4	0.0	241110.22	240702.40	240140.84	239844.80
97	sbvrp-n936-s4	935	C	R	SL	138	4	1.2	1.0	158350.03	154993.20	133332.20	129633.40
98	sbvrp-n957-s100	956	R	RC(4)	U	11	100	1.4	0.9	73653.74	73133.60	72926.08	72555.80
99	sbvrp-n979-s20	978	E	C(6)	Q	998	20	1.6	0.5	109252.51	107932.20	107572.13	106815.60
100	sbvrp-n1001-s4	1000	R	R	1-10	131	4	1.6	0.1	76996.90	76172.80	75520.17	74188.80

7.6 Conclusions and future work

This chapter investigated the SBVRP, a relevant VRP generalization which poses interesting challenges with respect to developing optimization approaches. The stochastic local search algorithm for the SBVRP that won the First VeRoLog Solver Challenge was presented.

The presented algorithm explores the problem structure and clearly outperforms all published approaches to the SBVRP. It has improved the best reported solution in the literature for the majority of the instances introduced during the First VeRoLog Solver Challenge. For *large*, *presel* and *final* instance sets, the worst solution obtained after 20 algorithm runs is better than the previous best known result, indicating the algorithm's superior performance. To encourage further research on the SBVRP, the CVRP instances proposed by Uchoa et al. (2017) were adapted to the SBVRP and an automated benchmark website[5] was produced including instances, solutions and a visualization tool.

Experiments showed the importance of the subproblem optimization in the final solution quality. Moreover, all algorithmic components were discussed and analyzed, including the hybrid metaheuristic, the considered neighborhoods, and the learning automaton. These combined analyses resulted in various insights concerning the development of competitive local search algorithms to the SBVRP.

Future research directions include simplifying the presented algorithm. This chapter described the winning approach of the First VeRoLog Challenge as it was implemented. However, the large number of neighborhoods and components hinder the understanding of the algorithm's behavior and prove a challenge in terms of reproduction. Additionally, core components such as the learning automaton may be further explored to prune neighborhoods, enabling simplification without significant loss in terms of solution quality. Finally, some of the ideas here presented should be applied to related problems, such as the VRPT and the TTRP.

[5] http://benchmark.gent.cs.kuleuven.be/sbvrp

Chapter 8

Multiple Container Loading Problem

This chapter addresses a real-world unsolved *Multiple Container Loading Problem* (MCLP) introduced by Renault and the EURO Special Interest Group on Cutting and Packing (ESICUP) on the occasion of the 2014/2015 ESICUP challenge. Renault's problem requires a large number of different items to be packed into containers of different types and sizes. Items must be grouped into stacks and respect the *this side up* constraint, meaning that items can only be rotated around the vertical axis. The primary objective is to minimize the volume of shipped containers.

This chapter is a minor adaptation of Toffolo et al. (2017b)[1], and presents a decomposition approach embedded in a multi-phase heuristic for the problem. Feasible solutions are generated quickly, and subsequently improved by local search and post-processing procedures. Experiments revealed that the approach generates optimal solutions for two instances, in addition to high-quality solutions for those remaining from the Renault set. The algorithmic contribution is, however, not restricted to the Renault instances. Experiments on less constrained container loading problems from the literature demonstrate the approach's general applicability and competitiveness.

[1] **Toffolo, T. A. M.**, Esprit, E., Wauters, T., and Vanden Berghe, G. (2017). A two-dimensional heuristic decomposition approach to a three-dimensional multiple container loading problem. *European Journal of Operational Research*, 257(2):526 – 538.

This chapter is organized as follows. Section 8.1 presents an introduction on container loading problems and offers a more detailed definition of the ESICUP challenge problem. A lower bound methodology is introduced in Section 8.2. The proposed decomposition-based heuristic is described by Section 8.3. The different subproblems are enumerated and the algorithm to address them are presented. The different phases of the proposed algorithm, resulting ultimately in a local search method incorporating all algorithms is also described. Computational experiments are presented in Section 8.4. The chapter is concluded by Section 8.5, which presents a summary and discussions concerning the proposed algorithms and future research.

8.1 Introduction

Efficient container loading is a key element within the continuously evolving domain of logistics and transportation. However, most studies tend not to focus on real-world problems. Bortfeldt and Wäscher (2013) highlighted this issue by presenting a comprehensive and insightful study of practically-relevant constraints of container loading problems, identifying and categorizing them. They argued how research in this field has dealt mainly with standard problems, often neglecting relevant issues encountered in practice. This claim is further supported by Zhao et al. (2016).

In this context, this chapter addresses a real-world multiple container loading problem introduced by Renault for the 2014/2015 ESICUP challenge (Clautiaux et al., 2015). This problem somewhat differs from those commonly addressed in the literature. In brief, it considers the merging of items into layers and combining these layers into stacks before packing these stacks into containers. Each individual problem defines a set of available container types. Layers contain rows of similar size items and, likewise, stacks should be composed of similar size layers. The objective is to minimize the total container volume required for packing the items, excluding the container with the smallest load volume, which is left behind for the next shipment.

Besides the constraints commonly associated with container loading, such as those related to the size and weight of items, a number of specific constraints must be taken into account. The first set of constraints determines how items should be packed into stacks. This set includes, among others, ‘this side up’, guillotine-like and weight bearing constraints. Other constraints restrict the

materials that may be combined in a stack. More specific constraints are associated with the container left behind. Shipment may only be postponed for a small percentage of the items of each product type and the container left behind must be the one holding the smallest volume (Clautiaux et al., 2015).

Considering the real-world constraints identified by Bortfeldt and Wäscher (2013), the problem proposed by Renault includes weight limits, orientation, stacking, stability and complexity constraints. In the typology suggested by Wäscher et al. (2007), the problem belongs to the category of *input minimization problems*. More specifically, it represents a type of *Multiple Stock-Size Cutting Stock Problem* (MSSCSP), since all instances provided by the 2014/2015 ESICUP challenge consist of weakly heterogeneous sets of items and containers.

Research concerning the MSSCSP is currently scarce, but very desirable. Zhao et al. (2016) analyzed 113 papers on container loading problems, and declared that their literature examination highlights three possible fruitful avenues for future research, with multiple container packing problems being one of them. Furthermore, the authors also state that "there are significantly fewer papers examining the multiple container packing problem when compared to the single container loading problem".

Most papers addressing the MSSCSP employ heuristic approaches. Ivancic et al. (1989) presented a placement heuristic based on integer programming in addition to several benchmarks instances. Eley (2002) developed an approach in which containers are filled by homogeneous blocks composed of identically-oriented items. A greedy constructive heuristic together with a tree-search algorithm improved the results reported by Ivancic et al. (1989). Later, Brunetta and Grégoire (2005) presented another tree-search heuristic algorithm to maximize the average volumetric utilization of containers. Their approach was applied to a real-world problem concerning a biscuit factory in France. Ren et al. (2011a) presented a priority-considering algorithm and improved some results for the instances of Ivancic et al. (1989). The algorithm assigns large items a higher priority to prevent their late placement, thereby improving the containers' utilization. Che et al. (2011) addressed a multiple container loading cost minimization problem with heuristics for a set cover formulation that builds on the work of Eley (2002). They further improved the results for the instances proposed by Ivancic et al. (1989).

The MSSCSP problem introduced by Renault, henceforth referred to as the ESICUP challenge problem, may be modeled as a logic decomposition of subproblems. Note that problem subdivision strategies are commonly applied

to container loading problems, as can be seen, for instance, in the approaches of Araya and Riff (2014), Parreño et al. (2008) and Bortfeldt et al. (2003). In this chapter a heuristic approach was produced, consisting of constructive and local search phases. The algorithm incorporates different components, each dealing with one of the decomposition subproblems. The algorithm's primary component is the *bin builder*, which is responsible for packing stacks into containers. Provided that a stack's height does not exceed the container's height, this subproblem may be solved as a 2D bin packing problem with weight constraints. The stack building subproblem, which mainly consists of piling up layers built by a dynamic programming algorithm, is solved by another component of the algorithm. Likewise, row building and layer building are handled by separate components. After a feasible solution is obtained, different intensification and diversification strategies are applied in parallel. These strategies include a ruin-and-recreate method for rebuilding parts of the solution, a multi-start and a procedure to insert or rearrange items in a container.

The ESICUP challenge had 17 registered participants representing institutions from 14 different countries. The challenge consisted of two tracks in two phases. Each track had a different runtime limit for solving a complete instance set: one hour for the *SHORT* and six hours for the *LONG* track. One set of instances was released for the first phase (*InstancesA*). For the second phase, another set of instances (*InstancesB*) was made public and a hidden set (*InstancesX*) was used to evaluate the different approaches. The presented algorithm competed in both tracks and garnered laudable comments from the organizing committee. For now, and to the best of our knowledge, it may be considered the strongest publicly available approach to the ESICUP challenge problem.

8.1.1 The ESICUP challenge problem

The ESICUP challenge problem is a combinatorial optimization problem that generalizes 3D packing. Given a set of different three-dimensional container types J, the purpose is to pack a set I of three-dimensional items into these containers while simultaneously minimizing the sum of the volume of the containers used.

Container types $j \in J$ have three spatial dimensions (W_j, L_j, H_j) and a maximum allowed weight R_j. Each container type is available in unlimited quantities. Items $i \in I$ have three dimensions (w_i, l_i, h_i), weight r_i and type

$t_i \in \{\text{metal, cardboard, wood}\}$. Furthermore, they are associated with a product p_i and their orientation is defined by $o_i \in \{\text{fixed, free}\}$, where 'fixed' means no rotation is allowed and 'free' means 90^o rotation around the z-axis ('this side up' constraint) is permitted.

The problem includes two inherent constraints that must be respected: (i) all items must lie entirely within a container and (ii) the items must not overlap. However, the items cannot be put directly into containers. They must be arranged into rows, layers and stacks. Figure 8.1 presents an example of a row, layer and stack. Note that a row is a set of items, while a layer is a set of rows and a stack is a set of layers.

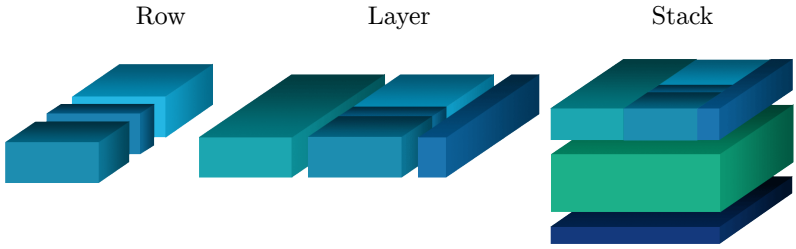


Figure 8.1: Representation of Row, Layer and Stack

The row/layer/stack structure introduces guillotine-like constraints, as the following rules must be considered:

- items within a particular row must have the same width (or length), within a tolerance;
- rows within a particular layer must have the same width (or length), within a tolerance;
- there is a limit on the number of items that rows and layers may have;
- the layers forming a stack must have the same width and length, within a tolerance, except for the stack's top layer;
- all items within a layer must have the same height, except when the layer is the top layer of the stack.

Additionally, rules concerning item weight must also be respected:

- stack layers must be sorted by weight, such that heavier layers are under lighter layers;

- the density of non-metal stacks, given by the total weight divided by ground area, is limited;
- the weight atop the bottom-most layer of non-metal stacks should not exceed a maximum value.

Most of the aforementioned constraints, such as guillotine, orientation and weight bearing, are commonplace within packing problems. However the problem also imposes additional less common constraints, concerning metal items and the smallest volume container (entitled Bin-0).

Metal items are subject to specific constraints. Firstly, stacks containing metal products must be formed exclusively by metal items. In these stacks, rows and layers must contain exactly one item and two constraints are relaxed: there is no maximum density and there is no maximum weight atop the bottom-most layer.

The items in Bin-0, the container holding the smallest volume, are left behind. There is, however, a maximum percentage of each product permitted within this container.

The problem's main objective function is the minimization of the total container volume excluding the volume of Bin-0. The volume in Bin-0 is only considered as a tie-break, meaning that when two solutions have the same main objective value then the one with the smallest volume inside Bin-0 is preferred. Two other tie-breaks are also introduced: (i) to minimize the lengths of the bin configurations, such that items are compacted at the back of the containers and (ii) to minimize the weighted sum of the items x-coordinates to also position as much weight as possible at the back of the container. Generally, however, the first tie break (Bin-0's volume) proved enough to resolve most cases of ties.

The official description of the ESICUP challenge problem (Clautiaux et al., 2015) provides further details concerning the problem and its motivation.

8.2 Lower bounds

In order to measure the solution quality produced by the presented algorithm, lower bounds for the problem instances have been computed.

A lower bound on the total container volume of a solution may be computed considering the set of items and set of available bin types. However, as the objective is to minimize the total container volume excluding Bin-0, the volume of Bin-0 should not be included in the lower bound. The maximum number of items in Bin-0 can be determined for each product type, thus providing an upper bound for the volume and weight of this bin's items. Hence, a valid lower bound for the instance is obtained by subtracting the volume of Bin-0 from the total required volume.

In certain cases, depending on the instance characteristics, lower bounds may be determined by the required weight capacity or minimum area. The required weight is the sum of all items' weight, while the minimum area is the area of the bottom layer necessary for satisfying the weight bearing constraint. Some instances contain very heavy items, with these requiring placement at the bottom layer.

A simple Mixed Integer Program (MIP) is solved to obtain a lower bound. Considering a problem with J container types and auxiliary constants n_j indicating the maximum number of containers of type $j \in J$, the following variables are defined:

$x_{j,k}$: binary variable that is equal to 1 if the k-th container of type j is used and 0 otherwise;

$a_{j,k}$: total area assigned to the k-th container of type j;

$y_{j,k}$: total volume assigned to the k-th container of type j;

$z_{j,k}$: total weight assigned to the k-th container of type j;

ba : total bottom area assigned to Bin-0 limited by A^+;

by : total volume assigned to Bin-0 limited by V^+;

bz : total weight assigned to Bin-0 limited by R^+.

Formulation (8.1)-(8.10) presents the lower bound MIP. The following notation is used: $K_j = \{1, ..., n_j\}$ is the set of available containers of type j; V_j, A_j and R_j are the volume, total area and weight capacity of containers of type j; $\tilde{A}$ is the minimum area required; and, finally, $\tilde{V}$ and $\tilde{R}$ denote the sum of the volume and weight of the items, respectively. Therefore, $\tilde{V} = \sum_{i \in I} w_i \, l_i \, h_i$ and $\tilde{R} = \sum_{i \in I} r_i$.

Minimize:

$$\sum_{j \in J} \sum_{k \in K_j} V_j x_{j,k} \tag{8.1}$$

Subject to:

$$\sum_{j \in J} \sum_{k \in K_j} a_{j,k} + ba = \tilde{A} \tag{8.2}$$

$$\sum_{j \in J} \sum_{k \in K_j} y_{j,k} + by = \tilde{V} \tag{8.3}$$

$$\sum_{j \in J} \sum_{k \in K_j} z_{j,k} + bz = \tilde{R} \tag{8.4}$$

$$by \leq y_{j,k} + \max_{j \in J} V_j (1 - x_{j,k}) \qquad \forall\, j \in J, k \in K_j \tag{8.5}$$

$$a_{j,k} \leq A_j x_{j,k},\ y_{j,k} \leq V_j x_{j,k},\ z_{j,k} \leq R_j x_{j,k} \qquad \forall\, j \in J, k \in K_j \tag{8.6}$$

$$a_{j,k} \geq a_{j,k+1},\ y_{j,k} \geq y_{j,k+1},\ z_{j,k} \geq z_{j,k+1} \qquad \forall\, j \in J, k \in K_j \backslash \{n_j\} \tag{8.7}$$

$$0 \leq ba \leq A^+,\ 0 \leq by \leq V^+,\ 0 \leq bz \leq R^+ \tag{8.8}$$

$$a_{j,k} \geq 0,\ y_{j,k} \geq 0,\ z_{j,k} \geq 0 \qquad \forall\, j \in J, k \in K_j \tag{8.9}$$

$$x_{j,k} \in \{0, 1\} \qquad \forall\, j \in J, k \in K_j \tag{8.10}$$

The objective function (8.1) exclusively considers the problem's main objective (minimizing the total container volume used). Constraints (8.2), (8.3) and (8.4) ensure that the required area, volume and weight match the allocation to the containers, respectively. Constraints (8.5) guarantee that Bin-0 is the bin with the smallest volume. Constraints (8.6) verify that the bottom area, volume and weight capacities of containers are not exceeded, while linking variables $a_{j,k}$, $y_{j,k}$ and $z_{j,k}$ to $x_{j,k}$. Symmetry among the different containers of each type is broken by Constraints (8.7). Constraints (8.8) limit the values for ba, by and bz, and Constraints (8.9) impede negative values for variables $a_{j,k}$, $y_{j,k}$ and $z_{j,k}$. Finally, Constraints (8.10) establish the binary nature of variables $x_{j,k}$.

The solution of Formulation (8.1)-(8.10) generally provides a weak lower bound, since many of the complex constraints are simplified or not even taken into account. Stronger lower bounds can be achieved by treating the problem as a 1D bin packing problem, separately considering the entire set of items in terms

of volume, weight and area. However, since the item size is relatively small compared to the size of the containers, there is little or no gain in terms of bound quality.

8.3 Decomposition-based heuristic

The row/layer/stack structure presented in Section 8.1.1 is exploited for heuristically decomposing the problem. The main idea is to consider different stacks as 2D items, dividing the problem in two subproblems: (*i*) *pack items into stacks* and (*ii*) *pack stacks into containers*.

This decomposition enables the creation of a straightforward two-step algorithm: first produce a set of feasible stacks and then pack them into containers. However, this approach has certain drawbacks. It may be necessary to build a large number of candidate stacks, since different container types (and dimensions) are considered. Furthermore, for a given set of stacks, the algorithm must ensure that an item is included at most once. Therefore, many produced stacks may potentially end up not even being considered.

Such drawbacks are managed by changing *when* stacks are built. Rather than producing the stacks beforehand, they are produced on-the-fly, while packing stacks into containers. To enable this, items are initially converted to 2D items, and a 2D bin packing problem is solved. Then, for each packed item, a stack is built considering its width and length, in addition to the height of the selected container. Note that only available items are considered when dynamically constructing the stacks, thus preventing the addition of an item to multiple stacks.

The two subproblems resulting from the decomposition are solved by different algorithms. The *stack builder* is responsible for packing items into stacks, while the *bin builder* is responsible for packing stacks into containers. The *stack builder* and *bin builder* algorithms are presented in Sections 8.3.1 and 8.3.2, respectively. Next, the local search algorithm is presented in Section 8.3.3.

8.3.1 Stack builder

Building a stack consists of arranging a set of layers with as much volume as possible, given upper limits for length, width, height and weight, defined

according to the stack's base item and the container type under consideration. A straightforward constructive algorithm is applied to this problem: layers are successively built and, for each new layer, the dimensional and updated weight limits are respected. Whenever a feasible layer is produced, it is added to the stack. Once no more layers can be produced, the resulting stack is returned.

Building a layer consists in, given a set of items I', dimension limits (w, l, h) and a maximum weight r, returning a feasible subset of positioned items. Note that these limits are defined according to the current container and stack states (their remaining height and weight capacity), in addition to the length and width of the stack base.

Layers are divided in two types: regular and relaxed. Regular layers must consist of items with equal height, while minimum values for length and width must be respected. Relaxed layers, however, are not subjected to such constraints. Note that the construction of relaxed layers is optional, but a relaxed layer may only be placed at the top of a stack. Therefore, a stack may have at most one relaxed layer, which must be the lightest layer of the stack, since layers are sorted by weight.

Building a regular layer

When building regular layers, upper dimensions (w, l, h) and weight (r) limits are considered, as well as lower limits on width and length. Lower limits are expressed in terms of a tolerance τ, with $0 \leq \tau \leq 1$. Thus, the width of the layer must lie between τw and w, and the length between τl and l. Given that upper limits are fixed, so are lower limits.

Regular layers are built by a dynamic programming algorithm. This approach benefits from the limits imposed concerning the size of rows and layers, and also from the weakly heterogeneous characteristic of the instances. Size limitations in combination with these characteristics significantly reduces the number of possible combinations. Nevertheless, it is worthwhile taking a few precautions by limiting the number of combinations, and thus avoiding a worst case scenario. This limit is given by a parameter α, with $\alpha \in \mathbb{Z}^+$.

The algorithm begins by filtering possible items into sets $C_h \subseteq I'$, such that every item $i \in C_h$ has height h while simultaneously respecting the dimensional and weight limits. Once the items are filtered, there are two special scenarios:

(1) there is a set C_h containing at least one item $i \in C_h$ such that $w_i \geq \tau w$ and $l_i \geq \tau l$;

(2) there is a set C_h with a subset $C_{h,l} \subseteq C_h$, $C_{h,l} \neq \emptyset$ such that $l_i \geq \tau l \; \forall i \in C_{h,l}$ and/or a subset $C_{h,w} \subseteq C_h$, $C_{h,w} \neq \emptyset$ such that $w_i \geq \tau w \; \forall i \in C_{h,w}$.

The first special scenario enables one to solve an easier layer building problem by utilizing the fact that individual items are themselves valid layers. Note that this is a necessary condition for metal layers, since they must contain exactly one item. When this scenario arises, the layer is formed by a single item $i \in C_h$ with $w_i \geq \tau w$ and $l_i \geq \tau l$.

The second special scenario also enables simplifying the layer building process. Note that a subset of items $X \subseteq C_{h,l}$ with $\tau w \leq \sum_{i \in X} w_i \leq w$ forms a valid layer. Analogously, a subset of items $X' \subseteq C_{h,w}$ also represents a solution if $\tau l \leq \sum_{i \in X'} l_i \leq l$. Therefore, there potentially exists a single-row solution (layer). In this scenario, a dynamic programming algorithm is applied to form a single-row layer. One of the $C_{h,l}$ or $C_{h,w}$ sets for which the condition is true is randomly selected, and only those items within the selected set are considered. If a valid layer is produced, it is returned. If not, the procedure is re-executed considering the remaining possible sets. Figure 8.2 depicts examples of typical single-row widthwise and lengthwise layers built by this dynamic programming algorithm.

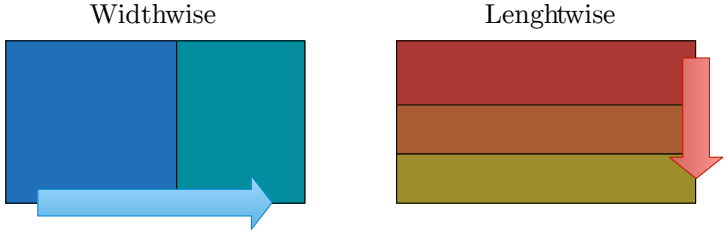


Figure 8.2: Example of single-row widthwise and lengthwise layers

The procedure to build single-row layers requires two arguments: a (partial) layer L and the ordered list of candidate items C. This list is composed of items with equal height, which is a condition for regular layers. Therefore, C contains items from the selected set ($C_{h,l}$ or $C_{h,w}$). Crucially, C must be sorted such that items with larger values for the considered dimension come first. For instance, if a widthwise layer is being built, then items with the largest width must come first.

The recursive single-row layer building procedure is presented by Algorithm 8.1. The procedure begins by guaranteeing that layer L can be enlarged and that the set C is not empty (line 1). Afterwards, the first item is removed from the candidates list (lines 2-3) and added to the layer (line 4). If a valid layer is obtained while respecting the requirements for dimension and weight, it is returned (lines 5-6). Otherwise a new set, C', is created (line 7). This set contains the feasible candidates that may be added to layer L without violating any constraints. The procedure is then recursively called with the new layer and set C'. If it succeeds in building a solution, the produced layer L is returned (lines 8-9). Otherwise, item i is removed from the layer and the next item is considered for incorporation in a recursive call (lines 10-12). If a solution is produced, it is returned (line 13). Otherwise, item i is put back into the candidate list and the algorithm backtracks by returning $\emptyset$ (lines 14-15).

Algorithm 8.1: Layer building algorithm

Let L be an empty layer, C be the sorted set of feasible items, and M be the maximum number of items allowed in a row or layer

BuildSingleRowLayer(L**,** C**):**

```
1  if |L| < M and C ≠ ∅ then
2      i ← first item in C
3      C ← C\{i}
4      L ← L + {i}
5      if L is a valid layer then
6          return L
7      C' ← set of items in C that may be added to layer L
8      if BuildSingleRowLayer(L, C') ≠ ∅ then
9          return L
10     else
11         L ← L\{i}
12         if BuildSingleRowLayer(L, C) ≠ ∅ then
13             return L
14     C ← {i} + C
15 return ∅
```

When neither special scenarios arise, Algorithm 8.1 operates within two separate contexts when creating a multiple-row layer. Initially, it is applied to obtain the set of base items to form the layer rows. Then it is applied individually to each of these items, expanding them towards valid rows.

Due to its performance, the creation of multiple-row layers was disabled in the

final version of the proposed algorithm. As expected, the creation of these layers can be very time consuming. Section 8.4 presents an experimental analysis documenting the impact of disabling multiple-row layers.

Additional pruning

Additional stopping criteria have been developed for building regular layers. Since the maximum number of items in a row or layer is known, and the candidate list is sorted, it is possible to prune the search tree at an early stage. Let d_i be the considered dimension of the first item in the candidate list, d_L the layer's current dimension, d the target dimension, M the maximum number of items in a row or layer and, finally, $|L|$ the number of items in the current row. If the condition given by Equation (8.11) does not hold, the algorithm can backtrack by returning $\emptyset$. Given that d_i is the maximum dimension of the possible items, and if adding $(M - |L|)$ times the dimension d_i does not make the layer large enough, then more than M items are necessary. Since this violates one of the problem's constraints, the algorithm may backtrack.

$$d_L + (M - |L|) \times d_i \geq d \tag{8.11}$$

Building a relaxed layer

The dynamic programming algorithm for regular layers can also be applied when building relaxed layers. The key differences are that items with different heights are considered, additional pruning is disabled, and the layer with the largest area is stored. If a layer with dimensions (w', l') where $\tau w \leq w' \leq w$ and $\tau l \leq l' \leq l$ is obtained, the algorithm returns it. Otherwise, the algorithm stops when the maximum number of iterations α is reached, returning the produced layer with largest area.

8.3.2 Bin builder

The *bin builder* is the highest-level component of the proposed decomposition. The purpose of the *bin builder* is to pack stacks in a container in a feasible manner. This is achieved by solving a classic 2D bin packing problem with weight constraints. The method relies heavily on the *stack builder* (Section 8.3.1) for handling the constraints related to layers and stacks.

The *bin builder* consists of a best-fit constructive algorithm (Burke et al., 2004). The implementation is based on the work of Imahori and Yagiura (2010), to which multiple sorting rules are added: (*i*) area, (*ii*) weight and (*iii*) volume. A random ordering of the items is also applied in addition to different placement policies: leftmost, rightmost, tallest neighbor, smallest neighbor and nearest neighbor.

Algorithm 8.2 presents the best-fit algorithm. The algorithm considers both length and width and is applied such that the gap indicates the container's largest dimension. Thus, if the length is greater than the width, the algorithm is applied width-wise with the gap indicating the length. The algorithm begins by creating an empty bin (line 1). While items may be added to this bin (line 2), the algorithm searches for one to add to the lowest gap (lines 3-4). If no item fits, the gap is elevated and the candidate set updated (lines 5-6). Otherwise, the first perfect fit candidate item is added to the bin (lines 7-9). It is important to note that Algorithm 8.2 aims at finding a perfect fit candidate, considering the size of the gap. However, if no perfect fit candidate is available, the first candidate which fits in the gap is considered. Once a candidate item is added, the *stack builder* procedure is called to transform it into a stack (line 10). Finally, the set of candidate items C is updated (line 11) with the items added and the state of the bin (its lowest gap and available weight capacity). Once no more items can be added to the bin, the algorithm returns (line 12).

Algorithm 8.2: Best-fit algorithm

Let C be the sorted set of candidate items, j be the container type, and p be the placement policy
BestFit(C, j, p)**:**
1 $b \leftarrow$ empty bin with dimensions of container type j
2 **while** $C \neq \emptyset$ **do**
3 $g \leftarrow$ lowest gap in b
4 **if** no candidate $c \in C$ fits in g **then**
5 elevate g to its shortest neighbor and update the gaps of b
6 $C \leftarrow$ items in C that may still fit in b
7 **else**
8 $c \leftarrow$ best candidate $c' \in C$ that fits in gap g
9 add candidate c to b according to policy p and update gaps of b
10 call ***stack builder*** procedure to turn item c into a stack
11 $C \leftarrow$ items in C that were not added and may still fit in b
12 return b

8.3.3 Local search algorithm

The local search algorithm can be divided in two phases: *(i)* a construction phase and *(ii)* a refinement phase. The refinement phase consists of either a multi-start approach, in which new solutions are generated from scratch, or a ruin-and-recreate local search. Figure 8.3 presents an outline of the algorithm. Its components are detailed in this section.

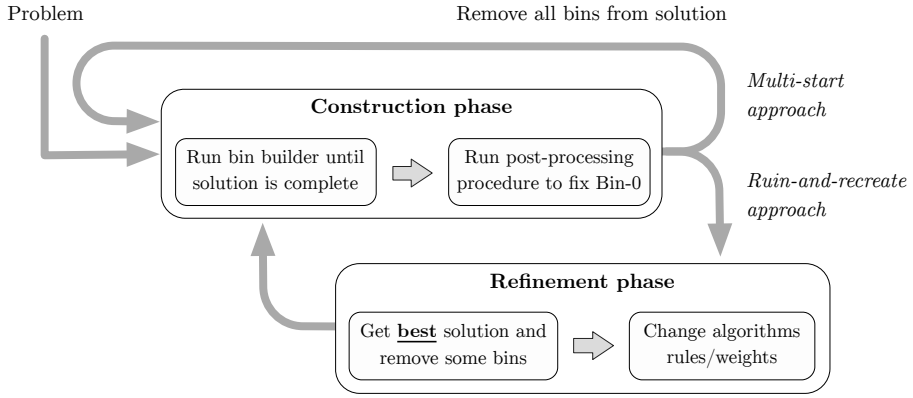


Figure 8.3: Local search algorithm outline

Construction phase

The construction phase begins by packing the first container, applying the *bin builder*. This procedure returns a feasibly-packed container configuration. The algorithm is executed for each available container type. Therefore, given an instance containing $|J|$ different container types, the result is a set of $|J|$ different containers filled with items. One item can be placed in multiple containers, but only one of these filled containers can be included in the solution. The container with the highest occupation rate ν is greedily selected, while the others are discarded. Equation (8.12) defines the occupation rate ν of each container, with X_b representing the set of items in container b, v_i the volume of item i and V_b the volume capacity of container b.

$$\nu = \frac{\sum_{i \in X_b} v_i}{V_b} \tag{8.12}$$

After a container is added to the solution, the procedure repeats considering the remaining items, until all items are packed and an initial solution is obtained.

The initial solution is not guaranteed feasible. Bin-0 may contain more items of a specific product than permitted. When infeasibility occurs, a simple repair procedure is executed.

Bin-0 repair procedure

The constructive procedure does not avoid infeasible solutions regarding the Bin-0 constraint, since it does not prohibit Bin-0 from exceeding the permitted limit of specific products. Whenever such a limit is not respected, the following procedure is executed. First a non-Bin-0 container b in the solution is randomly selected. Next, stacks from b are iteratively moved to Bin-0. The stacks to move are selected such that products causing infeasibility are moved first. Note, however, that some stacks in b may prove incompatible with the current Bin-0. For instance, the height or weight of a stack may impede moving it from b to Bin-0. These stacks are ignored and the procedure continues until no more stacks can be moved from b to Bin-0. If at this point the volume inside b is smaller than or equal to the volume inside Bin-0, b becomes the new Bin-0. If the infeasibility is fixed, the procedure finishes. Otherwise, another bin is selected and the procedure repeated until the solution is either feasible or all bins have been investigated. If the solution remains infeasible after investigating all bins, an empty bin is added to the solution, restoring feasibility at the expense of deteriorating the objective value. Figure 8.4 illustrates the repairing procedure.

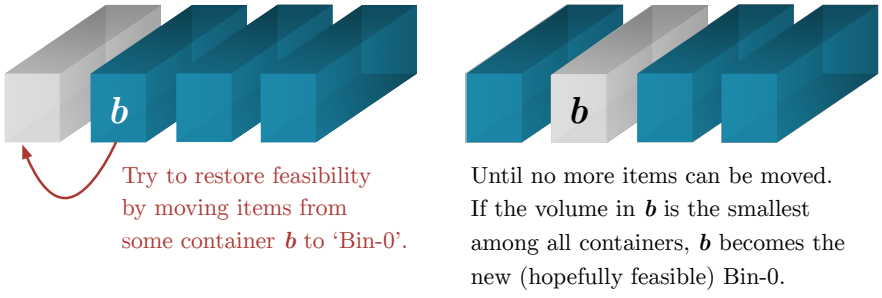


Figure 8.4: Visual example of the *Bin-0 repair procedure*

Refinement phase

Once a feasible initial solution is obtained, different intensification strategies are applied in parallel until the time limit is reached. A simple multi-start approach continuously rebuilds a solution from scratch in a greedy-randomized way. The randomization occurs during the container selection. Rather than greedily choosing the container with the highest occupation rate, the algorithm employs a roulette approach whereby containers with higher occupation rates have higher probabilities of being selected.

A ruin-and-recreate method is also applied, which consists of removing a random subset of containers to be rebuilt later using different container types and sorting criteria. The number of selected containers γ is chosen according to Equation (8.13):

$$\gamma = 1 + rand(min(B - 1, \eta)) \tag{8.13}$$

where η is an upper bound on the number of selected bins and B is the number of bins in the current solution. The upper bound η was introduced to limit the calculations on larger instances.

Different refinement strategies are executed in parallel, exchanging best solutions. This approach intensifies the search for better solutions, but renders the final algorithm non-deterministic.

8.4 Computational experiments

Computational experiments were performed in accordance with the ESICUP competition rules. The running time for solving each instance set was limited to one hour in the *SHORT* run and six hours in the *LONG* run. The test system was an Intel(R) Core(TM) i7-3770 CPU @ 3.4GHz computer with 16GB of RAM memory running Ubuntu Linux 12.04 LTS[2]. The maximum number of bins removed per iteration was set to $q = 5$ and the maximum number of iterations of the dynamic programming algorithm was set to $\alpha = 10,000$.

This section is organized as follows. The instances and their characteristics are presented in Section 8.4.1. Section 8.4.2 evaluates the algorithm components

[2]The challenge organizers used an Intel(R) Core(TM) i5-3570S @ 3.1GHz computer for the benchmarks.

and their impact. The best results obtained for the competition instances are presented in Section 8.4.3. Section 8.4.4 presents results obtained when applying the decomposition approach to other container loading problem instances from the literature.

8.4.1 Instances

The ESICUP challenge sets '*InstancesA*', '*InstancesB*' and '*InstancesX*', employed during the experiments, consist of real-world data. For all instances, at most 10% of each product is permitted in Bin-0, while the maximum weight on top of the bottom layer is set to 750 weight units. Tables 8.1, 8.2 and 8.3 show the following individual characteristics of these instances:

- the number of different container types ($|J|$);
- the main characteristics of the items: the average weight, number of item types (categorized considering the length, width and height), number of different products and the total number of items;
- the limit imposed on the number of items per row and per layer;
- the lower bounds obtained with the approach described in Section 8.2.

The lower bounds were computed very quickly (less than one second per instance) using SCIP (Achterberg, 2009). Note that only the set '*InstancesB*' contains instances with a single container type. The largest instance is 'ING_RIR_PEX_container' from set '*InstancesX*' with 2,597 items, while instance 'CVU_GX_PEX_container' from set '*InstancesA*' contains the highest number of item types (54).

Table 8.1: Characteristics analysis of *InstancesA* set

Instance	$\|J\|$	Items				Items limit		Lower bounds	
		Weight	Types	Prods.	Total	Row	Layer	Obj_1	Obj_2
ARG_BUR_STK_container	2	209.0	24	73	219	10	50	228.43	30.85
AVF_RIR_STK_container	2	303.9	25	201	805	10	50	1111.62	32.26
AVF_RIR_STK_truck	2	232.9	19	177	415	10	50	353.12	69.90
AVT_BUR_STK_container	2	180.1	25	67	145	10	50	152.29	33.37
AVZ_RIR_STK_container	2	295.2	17	143	675	10	50	548.23	31.91
AVZ_RIR_STK_truck	2	362.5	21	95	327	10	50	353.12	29.50
CHE_GC_PEX_container	2	169.6	19	63	211	10	50	206.81	25.83
CHE_RIR_STK_container	2	441.0	19	145	482	10	50	371.73	28.27
CME_GC_PEX_container	2	406.1	14	46	259	10	50	239.80	26.99
CVP_GC_PEX_container	2	490.6	29	103	736	10	50	973.45	55.20
CVU_GC_PEX_container	2	266.7	54	543	2447	10	50	4614.06	32.44
FSI_GC_PEX_container	2	284.8	40	316	967	10	50	1198.09	57.97
IND_RIR_STK_container	2	414.3	13	88	202	10	50	206.81	32.22
MED_GC_PEX_container	2	463.8	29	117	320	10	50	449.28	64.23
NSA_BUR_STK_container	2	260.3	21	43	117	10	50	152.29	41.34
NSA_RIR_STK_container	2	298.1	15	82	196	10	50	149.76	58.41
RSM_GC_PEX_container	2	794.5	15	40	151	10	50	332.51	31.12
SOM_BUR_STK_container	2	166.1	9	26	149	10	50	152.29	26.84
SOM_RIR_STK_container	2	182.8	34	219	811	10	50	1048.33	33.66
TNG_RIR_STK_container	2	418.3	15	70	211	10	50	215.73	26.30
Total	-	332.0	-	-	9845	-	-	13057.72	768.60

Table 8.2: Characteristics analysis of *InstancesB* set

Instance	$\|J\|$	Items				Items limit		Lower bounds	
		Weight	Types	Prods.	Total	Row	Layer	Obj_1	Obj_2
ACI_RIR_PEX_truck	2	607.82	8	19	162	4	12	353.12	1.72
ALG_RIR_PEX_container	1	119.13	26	100	1378	4	12	1797.13	40.06
AVF_RIR_PEX_container	1	373.75	17	145	346	4	12	374.40	43.99
AVZ_RIR_PEX_container	1	297.01	25	79	602	4	12	673.93	36.17
CHE_GC_PEX_container	3	197.94	23	80	315	4	12	275.35	61.05
CHE_RIR_PEX_container	1	206.06	21	85	771	4	12	673.93	49.30
CHE_VLD_PEX_container	1	720.56	10	31	249	4	12	599.04	20.90
CVP_GC_PEX_container	1	309.00	26	81	428	4	12	449.28	45.63
CVU_GC_PEX_container	1	292.46	47	326	863	4	12	1422.73	51.34
CVU_VLD_PEX_container	1	175.44	14	80	506	4	12	449.28	59.29
FSI_GC_PEX_container	1	275.63	37	258	597	4	12	599.04	66.00
FSI_VLD_PEX_container	1	309.12	21	140	705	4	12	898.57	52.11
IKO_RIR_PEX_truck	2	231.14	15	38	359	4	12	353.12	33.80
MED_GC_PEX_container	1	514.57	33	116	321	4	12	449.28	58.23
MJV_VLD_PEX_truck	3	346.31	7	13	172	4	12	264.84	61.72
NSA_RIR_PEX_container	1	351.85	24	142	788	4	12	1198.09	38.62
RSM_GC_PEX_container	1	533.68	18	32	248	4	12	449.28	4.42
SOM_VLD_PEX_container	1	442.79	25	90	1283	4	12	2471.06	2.32
TAK_RIR_PEX_truck	2	78.17	5	8	208	4	12	176.56	30.00
Total	-	335.92	-	-	9060	-	-	13928.04	756.67

Table 8.3: Characteristics analysis of *InstancesX* set

Instance	$\|J\|$	Items				Items limit		Lower bounds	
		Weight	Types	Prods.	Total	Row	Layer	Obj_1	Obj_2
ALG_RIR_PEX_container	2	119.83	20	65	552	4	12	685.29	49.99
ARG_BUR_PEX_container	2	241.92	29	240	825	4	12	913.71	59.37
AVF_RIR_PEX_container	2	362.27	21	118	991	4	12	1066.00	57.98
AVT_BUR_PEX_container	2	154.48	27	112	643	4	12	989.86	35.21
BAX_COR_PEX_container	2	262.90	14	52	616	4	12	821.76	29.56
CHE_RIR_PEX_container	2	385.46	18	129	545	4	12	523.20	31.03
CME_GC_EXP_container	2	406.92	14	54	361	4	12	341.53	30.37
CVP_GC_EXP_container	2	504.43	25	125	700	4	12	989.86	53.29
FSI_GC_EXP_container	2	276.24	39	415	1356	4	12	1725.68	32.54
MED_GC_EXP_container	2	401.13	20	152	648	4	12	911.49	31.18
NSA_RIR_PEX_container	2	341.45	23	91	362	4	12	456.86	68.23
RSM_GC_EXP_container	2	550.24	8	71	176	4	12	284.98	30.89
SOM_GC_EXP_container	2	311.13	17	71	566	4	12	685.29	61.71
SOM_RIR_PEX_container	2	216.43	25	193	1098	4	12	1522.86	66.64
TAN_BUR_PEX_container	2	207.34	20	115	561	4	12	533.00	40.77
TNG_RIR_PEX_container	2	229.29	23	237	2597	4	12	2937.29	32.89
VAX_COR_PEX_container	2	323.04	5	17	159	4	12	228.43	36.75
Total	-	311.44	-	-	12756	-	-	15617.07	748.42

8.4.2 Algorithm components

At the core of the *bin builder*, a 2D packing algorithm adds items (combined into stacks) to the bins. This section compares the performance of the implemented best-fit algorithm (Section 8.3.2) with a default bottom-left-fill algorithm (Chazelle, 1983), typically used for 2D packing problems. Table 8.4 shows the gap to the best total Obj_1 value obtained when best-fit and bottom-left-fill are employed as *bin-builder* algorithms. The results are organized by constructive phase and refinement phase result after 1 and 6 hours of computation time. All other components are equal for both runs. The table highlights how the best-fit algorithm performs significantly better than the bottom-left-fill. In fact, even after 6 hour of runtime, bottom-left-fill did not reach the quality of the initial solutions produced by best-fit.

Additional tests were conducted to analyze the effect of the strategies used to build layers. Table 8.5 presents the impact of: (*i*) disabling the dynamic programming and generating only single-item layers, (*ii*) generating only single-row layers and (*iii*) generating layers with multiple rows. The α parameter, indicating dynamic programming algorithm's iteration limit, was set to $10,000$ for the experiments. The table shows the gap to the best total Obj_1 value of each execution. Each phase's best results are highlighted in bold. Disabling the

Table 8.4: Gap to the best generated solutions among best-fit and bottom-left-fill as *bin builder* algorithms

Phase	*InstancesA*		*InstancesB*		*InstancesX*	
	Best-fit	BLF	Best-fit	BLF	Best-fit	BLF
Constructive phase	6.72%	21.21%	6.75%	22.40%	8.63%	14.61%
Refinement phase (1h)	0.88%	12.20%	1.27%	15.72%	0.99%	7.59%
Refinement phase (6h)	0.00%	11.32%	0.00%	15.08%	0.00%	6.59%

dynamic programming component causes a considerable increase in the number of iterations the algorithm can execute within the runtime limit. Nevertheless, the additional iterations are not sufficient to produce better solutions. In fact, the algorithm incorporating the dynamic programming procedure for single-row layers yielded better solutions in 1h than the algorithm considering only single-item layers did in 6h. The single-row strategy including the dynamic programming yielded better solutions than the multiple-row approach. The multiple-row approach tends to produce better layers, while requiring additional time for their generation. This additional time reduces the number of local search iterations within the time limit and renders it unworthy, since the single-row layers approach resulted in better solutions. Results were obtained for sets '*InstancesA*', '*InstancesB*' and '*InstancesX*'.

Table 8.5: Gap to the best generated solutions among single-item, single-row and multiple-row layers

Phase	Strategy	*InstancesA*	*InstancesB*	*InstancesX*
Constructive Phase	Single-item	7.45%	8.99%	9.10%
	Single-row	6.72%	6.75%	8.63%
	Multiple-row	**5.52%**	**6.28%**	**7.61%**
Refinement Phase (1h)	Single-item	2.13%	1.70%	1.14%
	Single-row	**0.88%**	**1.27%**	**0.99%**
	Multiple-row	3.46%	3.83%	2.05%
Refinement Phase (6h)	Single-item	1.28%	1.42%	0.32%
	Single-row	**0.00%**	**0.00%**	**0.00%**
	Multiple-row	3.46%	3.42%	1.43%

Interestingly, most rows formed by the single or multiple-row approach contain one item. However, cases in which more items are added to a row consistently impact the solution quality (see Table 8.5). Although the average number of items in a row is close to 1, some best solutions contain rows that reach

the number of items limit. An additional noteworthy observation is that by reducing the value of α, no multiple-row layers are generated for any of the considered instances, even with the multiple-row strategy enabled. Therefore, the multiple-row approach is not indicated for situations where the number of iterations (or runtime) is very limited.

8.4.3 Results for ESICUP instances

The final results for the ESICUP instances are shown in Tables 8.6, 8.7 and 8.8. Instances and solution files are available at the automated benchmark website[3] we developed for MCLP. These tables present the best known solution (BKS) for each individual instance, collected from the executions of all different approaches submitted to the competition[4] (including the solver presented in this chapter) in 1h of runtime, and the values for Obj_1 and Obj_2 obtained after 1h and 6h of runtime respectively with the presented approach. Solutions marked with a $\circledast$ were proven to be optimal for the first objective based on the calculated lower bounds. The best solutions are highlighted. The time dedicated to each instance is also shown in the tables. Equation (8.14) is employed to partition the total runtime among the instances, where t_k is the total time devoted to instance k, T is the total runtime, Z is the set of instances, I_z is the set of items in instance z and v_i is the volume of item i.

$$t_k = \frac{T}{2}\left(\frac{|I_k|}{\sum\limits_{z\in Z}|I_z|} + \frac{\sum\limits_{i\in I_k} v_i}{\sum\limits_{z\in Z}\sum\limits_{i\in I_z} v_i}\right) \tag{8.14}$$

The principle is to allot each instance an amount of computation time proportional to its number of items and the sum of the items' volumes, based on two assumptions: (i) an instance with many items tends to be harder to solve, and to require more computational time than an instance with fewer items; and (ii) an instance with a large total item volume generally impacts the overall objective function more than others with less total item volume.

Tables 8.6, 8.7 and 8.8 enable one to conclude that the presented approach is very competitive. The proposed approach improved the best known result for six instances and equaled the best known solution for 30 instances. In addition,

[3] http://benchmark.gent.cs.kuleuven.be/mclp

[4] Results extracted from the challenge website, http://challenge-esicup-2015.org, where the values for Obj_1 after 1h of runtime are available.

the lower bound approach (Section 8.2) enabled proving that three of the best known solutions are optimal for the first objective function (Obj_1).

Table 8.6: Results for *InstancesA* set

Instance	BKS - 1h	SHORT - 1h Runtime			LONG - 6h Runtime		
	Obj_1	Obj_1	Obj_2	Time	Obj_1	Obj_2	Time
ARG_BUR_STK_container	284.98	284.98	19.01	109.0	275.18	25.35	654.5
AVF_RIR_STK_container	1312.19	1312.19	21.39	257.0	1312.19	24.47	1542.9
AVF_RIR_STK_truck	441.40	441.40	53.73	147.1	441.40	50.86	882.9
AVT_BUR_STK_container	218.63	218.63	2.72	93.6	208.84	15.32	562.2
AVZ_RIR_STK_container	620.44	653.42	24.10	192.3	644.51	22.86	1154.6
AVZ_RIR_STK_truck	441.40	441.40	39.18	132.8	441.40	33.09	797.6
CHE_GC_PEX_container	224.64	224.64	45.08	105.7	224.64	40.13	634.8
CHE_RIR_STK_container	470.68	494.75	12.28	153.2	470.68	15.75	919.9
CME_GC_PEX_container	281.69	290.61	18.28	114.5	281.69	24.81	687.7
CVP_GC_PEX_container	1157.08	1240.88	8.32	238.6	1231.96	21.25	1432.5
CVU_GC_PEX_container	5159.62	5293.34	12.98	760.3	5255.01	23.76	4564.8
FSI_GC_PEX_container	1347.85	1387.07	25.14	286.4	1375.48	17.47	1719.6
IND_RIR_STK_container	248.71	248.71	25.11	105.2	248.71	23.16	631.6
MED_GC_PEX_container	578.54	587.45	17.23	143.3	587.45	9.25	860.6
NSA_BUR_STK_container	208.84	208.84	22.75	90.9	208.84	21.71	545.9
NSA_RIR_STK_container	215.73	224.64	24.31	101.8	224.64	20.13	611.1
RSM_GC_PEX_container	374.40	374.40	31.88	109.8	374.40	36.15	659.1
SOM_BUR_STK_container ⊛	152.29	185.46	26.80	93.6	185.46	26.80	561.7
SOM_RIR_STK_container	1216.81	1228.40	25.62	252.3	1210.57	20.04	1515.1
TNG_RIR_STK_container	248.71	272.78	14.68	106.5	272.78	17.69	639.7
Total	15204.63	15613.99	470.59	3593.9	15475.83	490.05	21578.8

Table 8.7: Results for *InstancesB* set

Instance	BKS - 1h	SHORT - 1h Runtime			LONG - 6h Runtime		
	Obj_1	Obj_1	Obj_2	Time	Obj_1	Obj_2	Time
ACI_RIR_PEX_truck ⊛	353.12	353.12	1.78	94.2	353.12	1.78	565.5
ALG_RIR_PEX_container	2021.78	2021.78	20.98	378.8	2021.78	10.16	2274.4
AVF_RIR_PEX_container	449.28	449.28	25.32	138.7	449.28	22.01	832.8
AVZ_RIR_PEX_container	823.69	823.69	46.80	193.2	823.69	46.80	1159.8
CHE_GC_PEX_container	342.17	342.17	34.71	128.2	342.17	32.46	769.5
CHE_RIR_PEX_container	748.81	823.69	15.28	213.9	823.69	12.66	1284.5
CHE_VLD_PEX_container	673.93	673.93	27.17	144.5	673.93	27.17	867.5
CVP_GC_PEX_container	524.16	524.16	47.15	154.7	524.16	38.16	928.9
CVU_GC_PEX_container	1572.49	1647.37	2.66	288.2	1572.49	49.77	1730.2
CVU_VLD_PEX_container	524.16	524.16	36.20	164.9	524.16	33.65	990.3
FSI_GC_PEX_container	673.93	748.81	2.48	188.8	673.93	59.67	1133.4
FSI_VLD_PEX_container	1198.09	1198.09	42.07	225.5	1198.09	34.40	1353.9
IKO_RIR_PEX_truck	366.20	441.40	10.16	137.6	372.73	77.82	825.9
MED_GC_PEX_container	599.04	599.04	23.28	143.3	599.04	20.41	860.6
MJV_VLD_PEX_truck	353.12	353.12	32.68	110.7	353.12	31.65	664.6
NSA_RIR_PEX_container	1347.85	1347.85	41.41	259.4	1347.85	31.86	1557.2
RSM_GC_PEX_container	524.16	524.16	28.02	129.9	524.16	28.02	779.7
SOM_VLD_PEX_container	2995.22	3444.51	39.01	395.0	3444.51	20.12	2371.7
TAK_RIR_PEX_truck ⊛	176.56	176.56	54.29	104.7	176.56	54.29	628.8
Total	16267.76	17016.89	531.45	3594.2	16798.46	632.86	21579.2

Table 8.8: Results for *InstancesX* set

Instance	BKS - 1h	SHORT - 1h Runtime			LONG - 6h Runtime		
	Obj_1	Obj_1	Obj_2	Time	Obj_1	Obj_2	Time
ALG_RIR_PEX_container	761.43	761.43	46.28	176.2	761.43	45.42	1057.6
ARG_BUR_PEX_container	1066.00	1073.57	20.40	219.2	1069.78	17.09	1316.0
AVF_RIR_PEX_container	1202.48	1212.27	13.37	245.8	1202.48	23.36	1476.0
AVT_BUR_PEX_container	1136.13	1155.72	8.95	205.9	1132.35	24.22	1236.3
BAX_COR_PEX_container	977.83	987.63	11.24	190.7	983.85	21.02	1144.7
CHE_RIR_PEX_container	599.35	616.71	23.55	162.2	609.14	43.85	974.1
CME_GC_EXP_container	380.71	394.30	25.96	131.6	380.71	43.68	790.2
CVP_GC_EXP_container	1176.87	1243.22	16.63	212.6	1219.84	25.29	1276.4
FSI_GC_EXP_container	1891.55	1911.14	18.89	326.6	1901.34	12.86	1960.6
MED_GC_EXP_container	1063.77	1073.57	20.92	200.4	1073.57	21.08	1202.9
NSA_RIR_PEX_container	579.75	579.75	21.07	142.9	566.17	30.01	858.1
RSM_GC_EXP_container	361.12	361.12	14.75	110.1	361.12	14.01	661.2
SOM_GC_EXP_container	811.97	825.55	19.40	178.3	811.97	22.80	1070.6
SOM_RIR_PEX_container	1864.39	1895.33	18.36	290.0	1871.96	26.39	1740.9
TAN_BUR_PEX_container	665.69	655.90	13.04	165.2	646.10	18.81	991.7
TNG_RIR_PEX_container	3224.49	3232.73	19.80	531.9	3232.73	9.50	3193.1
VAX_COR_PEX_container	298.56	304.57	23.70	104.8	304.57	21.76	629.3
Total	18062.09	18284.51	336.31	3594.4	18129.11	421.15	21579.7

8.4.4 General applicability

To enable a comparison between the proposed algorithm and those proposed by other container loading studies, experiments were performed on the LN01-LN15 instances introduced by Loh and Nee (1992) and the MPV instances by Martello et al. (2000) and Lodi et al. (2002b). These instances are the most directly comparable to those of ESICUP challenge given how items are weakly heterogeneous and the presence of the 'this side up' constraint, whereby rotations are not allowed around the horizontal axes. Table 8.9 presents a comparison between these two instance sets and the ESICUP challenge instances. The typology by Wäscher et al. (2007) is employed when characterizing them. Comparisons with other container loading instances is impractical given the absence of the 'this side up' constraint, which is a requirement of the dynamic programming component developed.

Table 8.9: Characteristics of the different instance sets

Instance set	Category	Weight	Material	Support	Row/Layer/Stack
ESICUP Challenge 2015	MSSCSP	✓	✓	✓	✓
Loh and Nee (1992)	SLOPP	-	-	-	-
Martello et al. (2000)	SSSCSP	-	-	-	-

It is important to note that, except for weight and material (metal-related) restrictions, the proposed approach was applied to these instances respecting all the constraints present in the ESICUP challenge problem. Therefore, in contrast to other approaches, the support and row/layer/stack constraints introduced by the ESICUP challenge (Clautiaux et al., 2015) are respected by the presented approach for all the other instances. Interestingly, both the addition of these constraints and the decomposition itself do not appear to considerably affect the quality of the final solution for some of these instances.

Table 8.10 presents the average number of bins required for each instance group from Martello et al. (2000) and Lodi et al. (2002b) by both the proposed algorithm and those in the literature, with the best results highlighted. The developed algorithm was executed for 10 minutes for each instance. While the presented approach results in more bins being required, which is expected given the additional constraints imposed, it is competitive with some approaches from the literature (Martello et al., 2000, Lodi et al., 2002a), whereas more recent works resulted in better solutions (Faroe et al., 2003, Crainic and Toulouse, 2008, Crainic et al., 2009).

Table 8.11 compares the volume occupation obtained by the developed approach against those from the literature for Loh and Nee (1992) instances. The best results are, again, highlighted. During this experiment, the developed algorithm was also executed for 10 minutes. An additional container was necessary to pack all the items for two instances and, therefore, only the occupation of the most-utilized container is presented. The table shows how the proposed approach is very competitive with the state-of-the-art (Ngoi et al., 1994, Bischoff et al., 1995, Bischoff and Ratcliff, 1995, Gehring and Bortfeldt, 1997, Bortfeldt and Gehring, 2001, Bortfeldt et al., 2003, Moura and Oliveira, 2005, He and Huang, 2010, Dereli and Das, 2011, Ren et al., 2011b, Lim and Rus, 2012) for these instances. Optimality was achieved for 13/15 of the instances (all items were placed in a single container), with occupation rates of over 93% achieved for the remaining two instances. In fact, only four of the dedicated approaches perform better: the tabu search methods (CBUSE) by Bortfeldt and Gehring (1998) and Bortfeldt et al. (2003), the CDFA algorithm by He and Huang (2010), the tree-search method by Ren et al. (2011b) and the iterated construction with dynamic prioritization by Lim and Rus (2012).

To facilitate future comparisons with other algorithms, experiments considering relaxed versions of the ESICUP challenge instances were also performed. Only three constraints remain: (*i*) all items must lie entirely within a container, (*ii*)

Table 8.10: Average number of used bins for the MPV instances by Martello et al. (2000)

#	Items	Martello et al. (2000)			Lodi et al. (2002b)	Faroe et al. (2003)	Crainic and Toulouse (2008)	Crainic et al. (2009)	Presented approach (*)
		A	B	C					
1	50	15.3	13.6	13.5	13.4	13.4	13.7	13.4	14.1
	100	27.4	27.3	29.5	26.6	26.7	27.2	26.7	27.8
	150	40.4	38.2	38.0	36.7	37.0	37.7	37.0	38.6
	200	55.6	52.3	52.3	51.2	51.2	51.9	51.1	52.3
2	50	-	-	-	13.8	-	-	-	14.4
	100	-	-	-	25.7	-	-	-	26.7
	150	-	-	-	37.2	-	-	-	38.7
	200	-	-	-	50.1	-	-	-	51.3
3	50	-	-	-	13.3	-	-	-	14.3
	100	-	-	-	26.0	-	-	-	27.0
	150	-	-	-	37.7	-	-	-	38.8
	200	-	-	-	50.5	-	-	-	52.0
4	50	29.8	29.4	29.4	29.4	29.4	29.4	29.4	29.5
	100	60.0	59.1	59.0	59.0	59.0	59.0	58.9	59.4
	150	87.9	87.2	87.3	86.8	86.8	86.8	86.8	87.3
	200	120.3	119.5	119.3	118.8	119.0	118.8	118.8	119.1
5	50	10.2	9.2	9.1	8.4	8.3	8.4	8.3	9.1
	100	17.6	17.5	17.0	15.0	15.1	15.1	15.2	15.9
	150	24.0	24.0	23.7	20.4	20.2	21.0	20.1	22.1
	200	31.7	31.8	31.7	27.6	27.2	28.1	27.4	29.0
6	50	11.2	9.8	11.0	9.9	9.8	10.1	9.8	10.0
	100	24.5	19.4	22.3	19.1	19.1	19.6	19.1	21.0
	150	35.0	29.6	32.4	29.4	29.4	29.9	29.2	31.0
	200	42.3	38.2	40.8	37.7	37.7	38.5	37.7	40.0
7	50	9.3	8.2	8.2	7.5	7.4	7.5	7.4	8.4
	100	15.3	15.3	13.9	12.5	12.3	13.2	12.3	13.8
	150	20.1	19.7	18.1	16.1	15.8	17.0	15.8	17.9
	200	28.7	28.1	28.0	23.9	23.5	25.1	23.5	25.9
8	50	11.3	10.1	9.9	9.3	9.2	9.4	9.2	10.3
	100	21.7	20.2	20.2	18.9	18.9	19.5	18.8	20.2
	150	28.3	27.3	26.8	24.1	23.9	25.2	23.9	26.3
	200	35.0	34.9	34.0	30.3	29.9	31.3	30.0	32.3
Average		33.5	32.1	32.3	30.8	**30.4**	31.0	**30.4**	32.0

(*) Note that the presented approach imposes additional constraints.

the items must not overlap and (*iii*) some items may not be rotated while others may be rotated around the z-axis only ('this side up' constraint). A single objective function was considered: to minimize the sum of container volume used. Tables 8.12, 8.13 and 8.14 present the results obtained with the proposed decomposition approach. Instance names were abbreviated, with *co* standing for *container*, *tr* for trucks and *re* for relaxed. Note that the approach considers the row/layer/stack constraints introduced by the ESICUP challenge, despite their absence in these instances. The goal is to provide a means for future

Table 8.11: Results obtained for the 15 instances from Loh and Nee (1992). Values represent the volume utilization (%).

	Ngoi et al. (1994)	Bischoff et al. (1995)	Bischoff and Ratcliff (1995)	Gehring and Bortfeldt (1997)	Bortfeldt and Gehring (1998)	Bortfeldt and Gehring (2001)	Moura and Oliveira (2005)	He and Huang (2010)	Dereli and Das (2011)	Ren et al. (2011b)	Lim and Rus (2012)	Presented approach (*)
LN01	62.5	62.5	62.5	62.5	62.5	62.5	62.5	62.5	62.5	62.5	62.5	62.5
LN02	80.7	89.7	90.0	90.7	96.7	89.8	92.6	97.9	86.3	97.9	96.4	93.2
LN03	53.4	53.4	53.4	53.4	53.4	53.4	53.4	53.4	53.4	53.4	53.4	53.4
LN04	55.0	55.0	55.0	55.0	55.0	55.0	55.0	55.0	55.0	55.0	55.0	55.0
LN05	77.2	77.2	77.2	77.2	77.2	77.2	77.2	77.2	77.2	77.2	77.2	77.2
LN06	88.7	89.5	83.1	91.1	96.3	92.4	91.7	96.7	89.2	96.3	93.5	93.0
LN07	81.8	83.9	78.7	82.7	84.7	84.7	84.7	84.7	83.2	84.7	84.7	84.7
LN08	59.4	59.4	59.4	59.4	59.4	59.4	59.4	59.4	59.4	59.4	59.4	59.4
LN09	61.9	61.9	61.9	61.9	61.9	61.9	61.9	61.9	61.9	61.9	61.9	61.9
LN10	67.3	67.3	67.3	67.3	67.3	67.3	67.3	67.3	67.3	67.3	67.3	67.3
LN11	62.2	62.2	62.2	62.2	62.2	62.2	62.2	62.2	62.2	62.2	62.2	62.2
LN12	78.5	76.5	78.5	78.5	78.5	78.5	78.5	78.5	78.5	78.5	78.5	78.5
LN13	84.1	82.3	78.1	85.6	85.6	85.6	85.6	85.6	85.6	85.6	84.9	85.6
LN14	62.8	62.8	62.8	62.8	62.8	62.8	62.8	62.8	62.8	62.8	62.8	62.8
LN15	59.5	59.5	59.5	59.5	59.5	59.5	59.5	59.5	59.5	59.5	59.5	59.5
Average	69.0	69.5	68.6	70.0	70.9	70.1	70.3	**71.0**	69.6	70.9	70.6	70.4

(*) Note that the presented approach imposes additional constraints.

evaluation of the row/layer/stack impact within the decomposition. Expectedly, the results differ considerably from those obtained for the original instances. These results, in addition to instances and solutions files, are available online[5].

Table 8.12: Results for relaxed *InstancesA* set

Instance	LB	UB	Instance	LB	UB
ARG_BUR_STK_co_re	261.60	304.57	CVU_GC_PEX_co_re	4647.04	5329.89
AVF_RIR_STK_co_re	1144.60	1369.24	FSI_GC_PEX_co_re	1258.71	1408.47
AVF_RIR_STK_tr_re	441.40	529.68	IND_RIR_STK_co_re	239.80	281.69
AVT_BUR_STK_co_re	199.04	242.01	MED_GC_PEX_co_re	515.25	623.11
AVZ_RIR_STK_co_re	581.22	656.10	NSA_BUR_STK_co_re	199.04	242.01
AVZ_RIR_STK_tr_re	441.40	454.47	NSA_RIR_STK_co_re	215.73	248.71
CHE_GC_PEX_co_re	239.80	272.78	RSM_GC_PEX_co_re	365.49	407.39
CHE_RIR_STK_co_re	404.71	449.28	SOM_BUR_STK_co_re	185.46	218.63
CME_GC_PEX_co_re	272.78	314.68	SOM_RIR_STK_co_re	1082.20	1276.53
CVP_GC_PEX_co_re	1030.50	1201.65	TNG_RIR_STK_co_re	248.71	281.69

Table 8.13: Results for relaxed *InstancesB* set

Instance	LB	UB	Instance	LB	UB
ACI_RIR_PEX_tr_re	176.56	176.56	FSI_GC_PEX_co_re	673.93	748.81
ALG_RIR_PEX_co_re	1872.02	2096.66	FSI_VLD_PEX_co_re	973.45	1272.97
AVF_RIR_PEX_co_re	449.28	524.16	IKO_RIR_PEX_tr_re	441.40	441.40
AVZ_RIR_PEX_co_re	748.81	898.57	MED_GC_PEX_co_re	524.16	673.93
CHE_GC_PEX_co_re	338.74	389.56	MJV_VLD_PEX_tr_re	353.12	441.40
CHE_RIR_PEX_co_re	748.81	823.69	NSA_RIR_PEX_co_re	1272.97	1422.73
CHE_VLD_PEX_co_re	673.93	748.81	RSM_GC_PEX_co_re	449.28	524.16
CVP_GC_PEX_co_re	524.16	599.04	SOM_VLD_PEX_co_re	2171.54	2995.22
CVU_GC_PEX_co_re	1497.61	1647.37	TAK_RIR_PEX_tr_re	264.84	264.84
CVU_VLD_PEX_co_re	524.16	599.04			

Table 8.14: Results for relaxed *InstancesX* set

Instance	LB	UB	Instance	LB	UB
ALG_RIR_PEX_co_re	735.82	827.77	MED_GC_EXP_co_re	944.66	1106.74
ARG_BUR_PEX_co_re	974.05	1089.38	NSA_RIR_PEX_co_re	526.99	589.55
AVF_RIR_PEX_co_re	1124.11	1235.65	RSM_GC_EXP_co_re	318.15	370.92
AVT_BUR_PEX_co_re	1026.81	1179.10	SOM_GC_EXP_co_re	749.41	845.14
BAX_COR_PEX_co_re	854.94	1017.02	SOM_RIR_PEX_co_re	1590.76	1905.13
CHE_RIR_PEX_co_re	556.38	642.32	TAN_BUR_PEX_co_re	579.75	679.27
CME_GC_EXP_co_re	374.70	427.47	TNG_RIR_PEX_co_re	2970.46	3265.90
CVP_GC_EXP_co_re	1044.18	1216.06	VAX_COR_PEX_co_re	265.39	337.74
FSI_GC_EXP_co_re	1758.85	1934.52			

[5]http://benchmark.gent.cs.kuleuven.be/mclp.

8.5 Conclusions and future work

This chapter focused on a real-world 3D container loading problem. A decomposition-based heuristic algorithm was introduced, inspired by the ESICUP 2014/2015 challenge problem characteristics. Rather than directly approaching a complex and very constrained 3D container loading problem, the problem is decomposed into multiple, simpler-to-solve, 2D bin packing problems whose solutions are combined to generate a solution for the overall problem.

The performance of the algorithm's individual components was evaluated and discussed. Furthermore, several experiments were conducted, devoting attention to both the ESICUP challenge instances and to less constrained container loading instances from the literature. In both cases, the proposed approach proved competitive with current best approaches, obtaining best known solutions for 36 out of 56 ESICUP challenge instances. It was also demonstrated as being applicable to other instance sets, producing high quality solutions for the instances of Loh and Nee (1992). All known optimal values were obtained by the decomposition approach. For the remaining instances, occupation rates above 93% were reached, despite the incorporation of specific ESICUP challenge constraints. When considering the instances of Martello et al. (2000) and Lodi et al. (2002b), obtained solutions were, on average, 5% worse than the best known.

Future research directions include investigating alternative strategies for building stacks and layers. The current algorithm, for instance, only considers a single item's dimensions as the target width and length of a stack. Other algorithms for the *bin builder* should also be analyzed, given the negative impact resulting from the adoption of bottom-left-fill instead of best-fit constructive heuristics.

Chapter 9

Conclusions

This thesis investigated decomposition-based algorithms for different combinatorial problems, resulting in significant scientific and practical contributions which were thoroughly discussed throughout previous chapters. These contributions directly impact future research concerning multiple problem domains, particularly when heuristic algorithms are considered.

While the thesis' primary focus concerns decomposition-based heuristic algorithms, the methodologies presented are not restricted to heuristic approaches. The structure of the TUP, for instance, was exploited to derive a decomposition-based exact algorithm, which was capable of optimally solving numerous unsolved problem instances. In fact, instances whose solution previously required over 24h of computational time were solved within only a few seconds. However, like any exact approach for the problems studied throughout this thesis, the exact algorithm proposed was unable to independently generate good solutions for large instances. Despite presenting superior performance when compared against commercial MIP solvers, both approaches belong to the same category due to one particular characteristic: their exponential time complexity gives the impression they are inadequate for addressing large problems. Such an impression was, however, disproved throughout the thesis by employing decomposition techniques.

The generally prohibitively long runtimes required by exact (and exponential worst-case time complexity) algorithms when solving large combinatorial problems are often used to motivate the investigation of problem-specific

heuristic methods. Indeed, such a motivation is valid. However, this thesis showed how simple decomposition strategies enable the employment of exact algorithms for producing high-quality solutions without the burden of extremely long runtimes. Moreover, the proposed approaches improved upon state-of-the-art results obtained by problem-specific methodologies. In summary, exponential-time algorithms are not only applicable for large problems but may also outperform more conventional methodologies in terms of solution quality.

It is true, however, that when extremely tight runtime limits are imposed, the role of such exponential-time algorithms may be questioned. We took this challenge upon ourselves, and implemented algorithms which participated in two international competitions. The initial goal was to achieve state-of-the-art performance employing exponential-time algorithms, but the runtime limits were indeed too tight. It proved necessary to investigate (faster) heuristic approaches for solving subproblems under this circumstance. Note, however, that the decomposition approaches were maintained, resulting in state-of-the-art algorithms even when subproblems are not solved to optimality. The combination of classic metaheuristic approaches with decomposition strategies culminated in award-winning algorithms for the two competitions. In both cases, the best publicly available approaches remain those proposed within this thesis.

Many combinatorial problems constitute the combination of different interconnected problems. These problems can often be decoupled into different decision sets, which may be exploited when developing decompositions approaches. One such example of this type of problem is the CVRP, for which a decision-set decomposition was studied. Fundamental decisions concerning *Assignment* and *Sequencing* were isolated and investigated. The objective was to evaluate the impact of taking optimal *Sequencing* decisions at all times, focusing instead upon the exploration of different *Assignment* decisions. Solving *Sequencing* subproblems to optimality proved very time consuming, and so an intermediate heuristic approach to address subproblems was investigated. Again, improvements over the state-of-the-art were observed.

Besides the aforementioned discussion concerning a problem's decision sets, this manuscript also investigated the generality of specific decomposition approaches. Part I addressed three different problems and proved how the competitive decomposition-based algorithms developed for the TUP, NRP and PSP are general. In fact, a general framework was built, one applicable not only to these three problems. By successfully solving an additional academic problem,

whose decomposition approach employs a few different principles, the generality of the framework was confirmed. It currently constitutes a powerful tool for evaluating decomposition strategies for a wide range of combinatorial optimization problems.

One of the primary conclusions arising from this thesis is that many problem-specific heuristics are replaceable by general decomposition-based algorithms. When compared against such problem-specific heuristics, decomposition-based approaches present interesting advantages. Take for instance the framework discussed throughout Chapter 5. Problems may be easily modified or extended, often requiring no algorithmic modifications. Such flexibility is rarely seen within heuristic algorithms, and constitutes a particularly interesting property for software developers seeking products capable of addressing a large number of problem variations. Moreover, the framework directly benefits from advances concerning the subproblem solver which, in our case, is a general MIP solver. As the boundaries of what can be optimally solved is pushed forward, these decomposition-based methodologies may become more relevant. Part I showed how larger subproblems tend to lead to higher-quality solutions. Therefore, as subproblem solvers become capable of addressing larger problems, decomposition-based algorithms tend to produce better and better solutions.

The findings presented by this manuscript resulted in many research questions which should be addressed in future work. Many questions were discussed throughout previous chapters, with some in particular deserving additional attention. One such example concerns the generality of certain decomposition-based heuristic algorithms. It was shown that the algorithms proposed in Part I are generalized by the general framework described in Chapter 5, however it is unclear to what degree this general framework can be extended. Future work includes investigating the challenges concerning automatic problem structure detection towards an automatic decomposition-based algorithm. This includes studying additional problems for further identifying the traits of successful decompositions. Another question concerns detecting to what extent algorithms may be simplified without significant loss of performance, both in terms of solution quality and computational time. The algorithms proposed for the two international challenges represent two cases in which this research question is relevant. Furthermore, decision-set decompositions should be investigated in greater detail. These decomposition approaches are particularly valuable for real-world industry applications composed of multiple interconnected problems, in which more than one decision set is present.

Finally, the contributions of this thesis go beyond proposing and evaluating decomposition methods. They also go beyond the proposition of general approaches and the development of reusable open source code. This thesis presented evidence that, rather than immediately proposing problem-specific heuristics, researchers should consider combining exact algorithms and decomposition approaches to derive their algorithms. Even when very constrained environments restrict optimally solving subproblems, the advantages of decomposition-based algorithms cannot be ignored. They are predominantly more flexible, general and future-proof. Furthermore, they performed greatly in terms of solution quality for all the problems addressed throughout this manuscript.

Bibliography

Absi, N., Cattaruzza, D., Feillet, D., and Housseman, S. (2017). A relax-and-repair heuristic for the Swap-Body Vehicle Routing Problem. *Annals of Operations Research*, 253(2):957–978.

Abuhamdah, A. (2010). Experimental result of late acceptance randomized descent algorithm for solving course timetabling problems. In *IJCSNS - International Journal of Computer Science and Network Security*, volume 10, pages 192–200.

Achterberg, T. (2009). SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41. http://mpc.zib.de/index.php/MPC/article/view/4.

Achterberg, T. and Wunderling, R. (2013). *Mixed Integer Programming: Analyzing 12 Years of Progress*, pages 449–481. Springer Berlin Heidelberg, Berlin, Heidelberg.

Ahuja, R., Ergun, Ö., Orlin, J., and Punnen, A. (2002). A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102.

Alba, E. and Chicano, J. F. (2007). Software Project Management with GAs. *Information Sciences*, (177):2380–2401.

Applegate, D., Bixby, R., Chvatal, V., and Cook, W. (2003). CONCORDE TSP Solver. http://www.math.uwaterloo.ca/tsp/concorde.html.

Araujo, J. A. S., Santos, H. G., Baltar, D. D., Toffolo, T. A. M., and Wauters, T. (2016). Neighborhood composition strategies in stochastic local search. In Blesa, M. J., Blum, C., Cangelosi, A., Cutello, V., Di Nuovo, A., Pavone, M., and Talbi, E.-G., editors, *Hybrid Metaheuristics: 10th International*

Workshop, HM 2016, Plymouth, UK, June 8-10, 2016, Proceedings, pages 118–130, Cham. Springer International Publishing.

Araya, I. and Riff, M. C. (2014). A beam search approach to the container loading problem. *Computers and Operations Research*, 43(1):100–107.

Artigues, C., Demassey, S., and Néron, E. (2013). *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE. Wiley.

Asta, S., Karapetyan, D., Kheiri, A., Özcan, E., and Parkes, A. J. (2016). Combining monte-carlo and hyper-heuristic methods for the multi-mode resource-constrained multi-project scheduling problem. *Information Sciences*, 373:476–498.

Awadallah, M. A., Al-Betar, M. A., Khader, A. T., Bolaji, A. L., and Alkoffash, M. (2017). Hybridization of harmony search with hill climbing for highly constrained nurse rostering problem. *Neural Computing and Applications*, 28(3):463–482.

Balas, E. and Simonetti, N. (2001). Linear time dynamic-programming algorithms for new classes of restricted TSPs: A computational study. *INFORMS Journal on Computing*, 13(1):56–75.

Balinski, M. and Quandt, R. (1964). On an integer program for a delivery problem. *Operations Research*, 12(2):300–304.

Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., and Vance, P. H. (1998). Branch-and-price: column generation for solving huge integer programs. *Operations Research*, 46:316–329.

Beasley, J. (1983). Route first-cluster second methods for vehicle routing. *Omega*, 11(4):403–408.

Benders, J. F. (1962). Partitioning procedures for solving mixed-variables programming problems. *Numer. Math.*, 4(1):238–252.

Berkelaar, M., Eikland, K., and Notebaert, P. (2017). Multi-platform, pure ANSI C/POSIX source code, Lex/Yacc based parsing. Online at http://lpsolve.sourceforge.net/5.5.

Bilgin, B., Demeester, P., Misir, M., Vancroonenburg, W., and Vanden Berghe, G. (2012). One hyper-heuristic approach to two timetabling problems in health care. *Journal of Heuristics*, 18(3):401–434.

Bischoff, E., Janetz, F., and Ratcliff, M. (1995). Loading pallets with non-identical items. *European Journal of Operational Research*, 84(3):681–692.

Bischoff, E. and Ratcliff, M. (1995). Issues in the development of approaches to container loading. *Omega*, 23(4):377–390.

Błażewicz, J., Lenstra, J., and Kan, A. R. (1983). Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24.

Bodin, L. and Berman, L. (1979). Routing and scheduling of school buses by computer. *Transportation Science*, 13(2):113.

Bompadre, A. (2012). Exponential Lower Bounds on the Complexity of a Class of Dynamic Programs for Combinatorial Optimization Problems. *Algorithmica*, 62:659–700.

Bortfeldt, A. and Gehring, H. (1998). Ein tabu search-verfahren für containerbeladeprobleme mit schwach heterogenem kistenvorrat. *Operations-Research-Spektrum*, 20(4):237–250.

Bortfeldt, A. and Gehring, H. (2001). A hybrid genetic algorithm for the container loading problem. *European Journal of Operational Research*, 131(1):143–161.

Bortfeldt, A., Gehring, H., and Mack, D. (2003). A parallel tabu search algorithm for solving the container loading problem. *Parallel Computing*, 29(5):641–662. Parallel computing in logistics.

Bortfeldt, A. and Wäscher, G. (2013). Constraints in container loading: A state-of-the-art review. *European Journal of Operational Research*, 229(1):1–20.

Boschetti, M. A., Maniezzo, V., Roffilli, M., and Bolufé Röhler, A. (2009). *Matheuristics: Optimization, Simulation and Control*, pages 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bresina, J. and Bresina, L. (1996). Heuristic-Biased Stochastic Sampling. In *AAAI-96 Proceedings*, pages 271–278.

Brunetta, L. and Grégoire, P. (2005). A general purpose algorithm for three-dimensional packing. *INFORMS Journal on Computing*, 17(3):328–338.

Brunner, J. O. (2010). *Flexible Shift Planning in the Service Industry: The Case of Physicians in Hospitals*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag Berlin Heidelberg.

Burke, E. K. and Bykov, Y. (2008). A Late Acceptance Strategy in Hill-Climbing for Exam Timetabling Problems. In *Proccedings of PATAT 2008 conference*, Montreal, Canada.

Burke, E. K. and Bykov, Y. (2017). The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70–78.

Burke, E. K. and Curtois, T. (2014). New approaches to nurse rostering benchmark instances. *European Journal of Operational Research*, 237(1):71–81.

Burke, E. K., Kendall, G., and Whitwell, G. (2004). A New Placement Heuristic for the Orthogonal Stock-Cutting Problem. *Operations Research*, 52(4):655–671.

Caramia, M. and Guerriero, F. (2009). A heuristic approach for the truck and trailer routing problem. *Journal of the Operational Research Society*, 61(7):1168–1180.

Chao, I. M. (2002). A tabu search method for the truck and trailer routing problem. *Computers and Operations Research*, 29(1):33–51.

Chazelle, B. (1983). The bottomn-left bin-packing heuristic: An efficient implementation. *IEEE Transactions on Computers*, 32(8):697–707.

Che, C., Zhang, Z., and Lim, A. (2011). A memetic algorithm for solving multiperiod vehicle routing problem with profit. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 45–46. ACM.

Christiaens, J. and Vanden Berghe, G. (2016). A fresh ruin & recreate implementation for the capacitated vehicle routing problem. Technical report, KU Leuven, Belgium.

Clarke, G. and Wright, J. W. (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581.

Clautiaux, F., Nguyen, A., and Brenaut, J.-P. (2015). Model for the challenge Renault/ESICUP: version 1.3. http://challenge-esicup-2015.org/doc/modele_renault.pdf.

Coelho, L., Cordeau, J.-F., and Laporte, G. (2011). Consistency in Multi-Vehicle Inventory-Routing. *CIRRELT Working Paper*.

Cordeau, J.-F., Laporte, G., Savelsbergh, M., and Vigo, D. (2007). Vehicle Routing. In Barnhart, C. and Laporte, G., editors, *Transportation*, pages 367–428. Elsevier, North-Holland, Amsterdam.

Crainic, T. and Toulouse, M. (2008). Explicit and Emergent Cooperation Schemes for Search Algorithms. In Maniezzo, V., Battiti, R., and Watson, J.-P., editors, *Learning and Intelligent Optimization*, volume 5315 of *LNCS*, pages 95–109, Berlin, Heidelberg. Springer-Verlag.

Crainic, T. G., Le Cun, B., and Roucairol, C. (2006). *Parallel Branch-and-Bound Algorithms*, pages 1–28. John Wiley & Sons, Inc.

Crainic, T. G., Perboli, G., and Tadei, R. (2009). TS2PACK: A two-level tabu search for the three-dimensional bin packing problem. *European Journal of Operational Research*, 195(3):744–760.

Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812.

Danna, E., Rothberg, E., and Le Pape, C. (2003). Integrating mixed integer programming and local search: A case study on job-shop scheduling problems. In *Proceedings Third International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming (CPAIOR'03)*, Montreal, Canada.

Dantzig, G. B. and Wolfe, P. (1960). Decomposition Principle for Linear Programs. *Operations Research*, 8(1):101–111.

Davis, M. and Loveland, D. (1962). A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397.

de Oliveira, L., de Souza, C. C., and Yunes, T. (2014). Improved bounds for the traveling umpire problem: A stronger formulation and a relax-and-fix heuristic. *European Journal of Operational Research*, 236(2):592–600.

de Oliveira, L., de Souza, C. C., and Yunes, T. (2016). Lower bounds for large traveling umpire instances: New valid inequalities and a branch-and-cut algorithm. *Computers & Operations Research*, 72:147–159.

Deineko, V. G. and Woeginger, G. J. (2000). A study of exponential neighborhoods for the Travelling Salesman Problem and for the Quadratic Assignment Problem. *Mathematical Programming*, 87:519–542.

Demeulemeester, E. L. and Herroelen, W. S. (2002). *Project Scheduling: A Research Handbook*. Kluwer Academic Publishers.

Dereli, T. and Das, G. S. (2011). A hybrid bee(s) algorithm for solving container loading problems. *Applied Soft Computing*, 11(1):2854–2862.

Eley, M. (2002). Solving container loading problems by block arrangement. *European Journal of Operational Research*, 141:393–409.

Faroe, O., Pisinger, D., and Zachariasen, M. (2003). Guided Local Search for the Three-Dimensional Bin-Packing Problem. *INFORMS Journal on Computing*, 15(3):267–283.

Fischetti, M. and Lodi, A. (2003). Local branching. *Mathematical Programming*, 98(1-3):23–47.

Fischetti, M., Lodi, A., and Salvagnin, D. (2010). Just mip it! In Maniezzo, V., Stützle, T., and Voü, S., editors, *Matheuristics*, volume 10 of *Annals of Information Systems*, pages 39–70. Springer US.

Fisher, M. and Jaikumar, R. (1981). A generalized assignment heuristic for vehicle routing. *Networks*, 11(2):109–124.

Fonseca, G. H. G., Santos, H. G., Toffolo, T. A. M., Brito, S. S., and Souza, M. J. F. (2016). Goal solver: a hybrid local search based solver for high school timetabling. *Annals of Operations Research*, 239(1):77–97.

Foster, B. and Ryan, D. (1976). An integer programming approach to the vehicle scheduling problem. *Operational Research Quarterly*, 27(2):367–384.

Gamrath, G. and Lübbecke, M. E. (2010). *Experiments with a Generic Dantzig-Wolfe Decomposition for Integer Programs*, pages 239–252. Springer Berlin Heidelberg, Berlin, Heidelberg.

Gehring, H. and Bortfeldt, A. (1997). A genetic algorithm for solving the container loading problem. *International Transactions in Operational Research*, 4(5-6):401–418.

Geiger, M. (2013). Iterated variable neighborhood search for the resource constrained multi-mode multi-project scheduling problem. some comments on our contribution to the mista 2013 challenge. *Multidisciplinary International Scheduling Conference (MISTA) 2013 Proceedings*, 27-29:807–811.

Gendron, B. and Crainic, T. G. (1994). Parallel branch-and-branch algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066.

Geoffrion, A. (1970). Elements of large-scale mathematical programming: Part I: Concepts. *Management Science*, 16(11):652–675.

Gerdessen, J. C. (1996). Vehicle routing problem with trailers. *European Journal of Operational Research*, 93(1):135–147.

Glover, F. (1991). Multilevel tabu search and embedded search neighborhoods for the traveling salesman problem. Technical report, Leeds School of Business, University of Colorado, Boulder.

Glover, F. (1996). Ejection chains, reference structures and alternating path methods for traveling salesman problems. *Discrete Applied Mathematics*, 65(1-3):223–253.

Goel, A. and Vidal, T. (2014). Hours of service regulations in road freight transport: an optimization-based international assessment. *Transportation Science*, 48(3):391–412.

Goerler, A., Schulte, F., and Voß, S. (2013). An application of late acceptance hill-climbing to the traveling purchaser problem. In Pacino, D., Voß, S., and Jensen, R., editors, *Computational Logistics*, volume 8197 of *Lecture Notes in Computer Science*, pages 173–183. Springer Berlin Heidelberg.

Gomes, R. A. M., Toffolo, T. A. M., and Santos, H. G. (2017). Variable neighborhood search accelerated column generation for the nurse rostering problem. *Electronic Notes in Discrete Mathematics*, 58:31–38. 4th International Conference on Variable Neighborhood Search.

Gschwind, T. and Drexl, M. (2016). Adaptive large neighborhood search with a constant-time feasibility test for the dial-a-ride problem. Technical Report LM-2016-08, Chair of Logistics Management, Gutenberg School of Management and Economics, Johannes Gutenberg University Mainz, Mainz, Germany.

Gutin, G. and Yeo, A. (2003). Upper bounds on ATSP neighborhood size. *Discrete Applied Mathematics*, 129(2-3):533–538.

Hansen, P. and Jaumard, B. (1997). Cluster analysis and mathematical programming. *Mathematical Programming*, 79(1-3):191–215.

Hansen, P., Mladenović, N., and Perez-Britos, D. (2001). Variable neighborhood decomposition search. *Journal of Heuristics*, 7(4):335–350.

Hansen, P., Mladenović, N., and Urosević, D. (2006). Variable neighborhood search and local branching. *Comput. Oper. Res.*, 33(10):3034–3045.

Hartmann, A. and Rieger, H. (2002). *Optimization Algorithms in Physics*. Wiley-VCH Verlag GmbH & Co. KGaA.

Haspeslagh, S., De Causmaecker, P., Schaerf, A., and Stø levik, M. (2012). The first international nurse rostering competition 2010. *Annals of Operations Research*, pages 1–31.

He, K. and Huang, W. (2010). A caving degree based flake arrangement approach for the container loading problem. *Computers & Industrial Engineering*, 59(2):344–351.

Heid, W., Hasle, G., and Vigo, D. (2014). VeRoLog solver challenge 2014 – VSC2014 problem description. http://verolog.deis.unibo.it/news-events/general-news/verolog-solver-challenge-2014.

Hintsch, T. and Irnich, S. (2017). Large multiple neighborhood search for the clustered vehicle-routing problem. Discussion paper number 1701, Johannes Gutenberg University Mainz, Germany. Available at http://wiwi.uni-mainz.de/Papers/Discussion_Paper_1701.pdf.

Huber, S. and Geiger, M. (2014). Swap body vehicle routing problem: A heuristic solution approach. In González-Ramírez, R., Schulte, F., Voß, S., and Ceroni Díaz, J., editors, *Computational Logistics*, volume 8760 of *Lecture Notes in Computer Science*, pages 16–30. Springer International Publishing.

Huber, S. and Geiger, M. J. (2017). Order matters - A variable neighborhood search for the swap-body vehicle routing problem. *European Journal of Operational Research*, 263(2):419–445.

Imahori, S. and Yagiura, M. (2010). The best-fit heuristic for the rectangular strip packing problem: An efficient implementation and the worst-case approximation ratio. *Computers and Operations Research*, 37(2):325–333.

Irnich, S. (2008). Solution of real-world postman problems. *European Journal of Operational Research*, 190(1):52–67.

Irnich, S. (2013). Efficient local search for the CARP with combined exponential and classical neighborhood. In *VeRoLog Conference 2013*, Southampton, U.K.

Ivancic, N., Mathur, K., and Mohanty, B. B. (1989). An Integer Programming Based Heuristic Approach to the Three-dimensional Packing Problem. *Journal of Operations Management*, 2:268–298.

Johnson, D. and McGeoch, L. (1997). The traveling salesman problem: A case study in local optimization. In Aarts, E. and Lenstra, J., editors, *Local search in Combinatorial Optimization*, pages 215–310. University Press, Princeton, NJ.

Józefowska, J. and Weglarz, J. (2006). *Perspectives in modern project scheduling*. International series in operations research & management science. Springer.

Kelley, Jr, J. E. and Walker, M. R. (1959). Critical-path planning and scheduling. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 160–173, New York, NY, USA. ACM.

Kendall, G., Bai, R., Błażewicz, J., De Causmaecker, P., Gendreau, M., John, R., Li, J., McCollum, B., Pesch, E., Qu, R., Sabar, N., Vanden Berghe, G., and Yee, A. (2016). Good Laboratory Practice for optimization research. *Journal of the Operational Research Society*, 67(4):676–689.

Kernighan, B. and Ritchie, D. (1988). *The C programming language*, volume 78. Prentice-Hall, Inc., 2nd edition.

Kerzner, H. (2013). *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*. Wiley, 10th edition.

Klein, R. (2000). *Scheduling of Resource-Constrained Projects*. Operations research/computer science interfaces series. Kluwer Academic.

Kolisch, R. and Hartmann, S. (1999). Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis. In Węglarz, J., editor, *Project Scheduling*, volume 14 of *International Series in Operations Research & Management Science*, pages 147–178. Springer US.

Kolisch, R. and Hartmann, S. (2006). Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, 174(1):23–37.

Kolisch, R. and Sprecher, A. (1997). PSPLIB - A project scheduling problem library. *European Journal of Operational Research*, 96(1):205–216.

Koné, O., Artigues, C., Lopez, P., and Mongeau, M. (2011). Event-based MILP models for resource-constrained project scheduling problems. *Computers & Operations Research*, 38(1):3–13.

Koné, O., Artigues, C., Lopez, P., and Mongeau, M. (2013). Comparison of mixed integer linear programming models for the resource-constrained project scheduling problem with consumption and production of resources. *Flexible Services and Manufacturing Journal*, 25(1-2):25–47.

Krishnamoorthy, M., Ernst, A., and Baatar, D. (2012). Algorithms for large scale shift minimisation personnel task scheduling problems. *European Journal of Operational Research*, 219(1):34–48.

Krüger, K., Shakhlevich, N. V., Sotskov, Y. N., Werner, F., Kruger, K., Shakhlevich, N. V., and Sotskov, Y. N. (2016). A Heuristic Decomposition Algorithm for Scheduling Problems on Mixed Graphs. *Journal of the Operational Research Society*, 46(12):1481–1497.

Land, A. H. and Doig, A. G. (1960). An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520.

Li, K. and Willis, R. (1992). An iterative scheduling technique for resource-constrained project scheduling. *European Journal of Operational Research*, 56(3):370–379.

Lim, S. and Rus, D. (2012). Stochastic motion planning with path constraints and application to optimal agent, resource, and route planning. *2012 IEEE International Conference on Robotics and Automation*, pages 4814–4821.

Lin, S.-W., Yu, V. F., and Chou, S.-Y. (2009). Solving the truck and trailer routing problem based on a simulated annealing heuristic. *Computers & Operations Research*, 36(5):1683–1692.

Lin, S.-W., Yu, V. F., and Chou, S.-Y. (2010). A note on the truck and trailer routing problem. *Expert Systems with Applications*, 37(1):899–903.

Lin, S.-W., Yu, V. F., and Lu, C.-C. (2011). A simulated annealing heuristic for the truck and trailer routing problem with time windows. *Expert Systems with Applications*, 38(12):15244–15252.

Lodi, A., Martello, S., and Monaci, M. (2002a). Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141:241–252.

Lodi, A., Martello, S., and Vigo, D. (2002b). Heuristic algorithms for the three-dimensional bin packing problem. *European Journal of Operational Research*, 141:410–420.

Loh, T. H. and Nee, A. Y. C. (1992). A packing algorithm for hexahedral boxes. In *Proceedings of the Conference of Industrial Automation*, pages 115–126.

López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., and Birattari, M. (2011). The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium.

Lourenço, H., Martin, O., and T., S. (2010). Iterated local search: Framework and applications. In *Handbook of Metaheuristics, 2nd. Edition*, volume 146, pages 363—397. Kluwer Academic Publishers, International Series in Operations Research & Management Science.

Lourenço, H. R., Martin, O. C., and Stützle, T. (2003). Iterated local search. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 320–353. Springer US.

Lü, Z. and Hao, J. K. (2012). Adaptive neighborhood search for nurse rostering. *European Journal of Operational Research*, 218(3):865–876.

Lübbecke, M. E. and Desrosiers, J. (2005). Selected Topics in Column Generation. *Operations Research*, 53(6):1007–1023.

Lum, O., Chen, P., Wang, X., Golden, B., and Wasil, E. (2015). A Heuristic Approach for the Swap-Body Vehicle Routing Problem. In *14th INFORMS Computing Society Conference*, pages 172–187.

Maniezzo, V., Stützle, T., and Voß, S. (2010). *Matheuristics: Hybridizing Metaheuristics and Mathematical Programming*. Annals of Information Systems.

Martello, S., Pisinger, D., and Vigo, D. (2000). The Three-Dimensional Bin Packing Problem. *Operations Research*, 48(2):256–267.

Martins, A. X., De Souza, M. C., Souza, M. J. F., and Toffolo, T. a. M. (2009). GRASP with hybrid heuristic-subproblem optimization for the multi-level capacitated minimum spanning tree problem. *Journal of Heuristics*, 15(2):133–151.

Michie, D. (1968). "Memo" functions and machine learning. *Nature*, 218(5136):19–22.

Miranda-Bront, J. J., Curcio, B., Méndez-Díaz, I., Montero, A., Pousa, F., and Zabala, P. (2017). A cluster-first route-second approach for the swap body vehicle routing problem. *Annals of Operations Research*, 253(2):935–956.

Mittelmann, H. (2017). Mixed integer linear programming benchmark. http://plato.asu.edu/ftp/milpc.html.

Möhring, R. H., Schulz, A. S., Stork, F., and Uetz, M. (2003). Solving project scheduling problems by minimum cut computations. *Management Science*, 49(3):330–350.

Moura, A. and Oliveira, J. F. (2005). A grasp approach to the container-loading problem. *IEEE Intelligent Systems*, 20(4):50–57.

Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):pp. 32–38.

Muter, I., Birbil, S., and Sahin, G. (2010). Combination of metaheuristic and exact algorithms for solving set covering-type optimization problems. *INFORMS Journal on Computing*, 22(4):603–619.

Narendra, K. S. and Thathachar, M. A. L. (1989). *Learning Automata: An Introduction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Ngoi, B. K. A., Tay, M. L., and Chua, E. S. (1994). Applying spatial representation techniques to the container packing problem. *International Journal of Production Research*, 32(1):111–123.

Nonobe, K. (2010). INRC2010: An approach using a general constraint optimization solver.

Özcan, E., Bykov, Y., Birben, M., and Burke, E. K. (2009). Examination timetabling using late acceptance hyper-heuristics. In *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, CEC'09, pages 997–1004, Piscataway, NJ, USA. IEEE Press.

Parragh, S. N. and Cordeau, J.-F. (2017). Branch-and-price and adaptive large neighborhood search for the truck and trailer routing problem with time windows. *Computers & Operations Research*, 83:28–44.

Parreño, F., Alvarez-Valdes, R., Tamarit, J. M., and Oliveira, J. F. (2008). A maximal-space algorithm for the container loading problem. *INFORMS Journal on Computing*, 20(3):412–422.

Pollaris, H., Braekers, K., Caris, A., Janssens, G. K., and Limbourg, S. (2015). Vehicle routing problems with loading constraints: state-of-the-art and future directions. *OR Spectrum*, 37(2):297–330.

Pritsker, A. A. B., Watters, L. J., and Wolfe, P. M. (1969). Multi project scheduling with limited resources: A zero-one programming approach. *Management Science*, 3416:93–108.

Rahimian, E., Akartunalı, K., and Levine, J. (2017). A hybrid integer and constraint programming approach to solve nurse rostering problems. *Computers and Operations Research*, 82:83–94.

Ren, J., Tian, Y., and Sawaragi, T. (2011a). A priority-considering approach for the multiple container loading problem. *International Journal of Metaheuristics*, 1(4):298–316.

Ren, J., Tian, Y., and Sawaragi, T. (2011b). A tree search method for the container loading problem with shipment priority. *European Journal of Operational Research*, 214(3):526–535.

Renaud, J., Boctor, F., and Laporte, G. (1996). An improved petal heuristic for the vehicle routeing problem. *Journal of the Operational Research Society*, 47(2):329–336.

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

Santos, H. G., Toffolo, T. A. M., Ribas, S., and Gomes, R. A. M. (2012). Integer programming techniques for the Nurse Rostering Problem. *9th International Conference on Practice and Theory of Automated Timetabling (PATAT) 2012 Proceedings*, pages 257–282.

Santos, H. G., Toffolo, T. A. M., Silva, C. L. T. F., and Vanden Berghe, G. (2017). Analysis of stochastic local search methods for the unrelated parallel machine scheduling problem. *International Transactions in Operational Research*, page (In press).

Scheuerer, S. (2006). A tabu search heuristic for the truck and trailer routing problem. *Computers & Operations Research*, 33(4):894–909.

Smet, P., Wauters, T., Mihaylov, M., and Vanden Berghe, G. (2014). The shift minimisation personnel task scheduling problem: A new hybrid approach and computational insights. *Omega*, 46:64–73.

Subramanian, A., Uchoa, E., and Ochi, L. (2013a). A hybrid algorithm for a class of vehicle routing problems. *Computers & Operations Research*, 40(10):2519–2531.

Subramanian, A., Uchoa, E., and Ochi, L. S. (2013b). A hybrid algorithm for a class of vehicle routing problems. *Computers & Operations Research*, 40(10):2519–2531.

Tassopoulos, I. X., Solos, I. P., and Beligiannis, G. N. (2015). A two-phase adaptive variable neighborhood approach for nurse rostering. *Computers & Operations Research*, 60:150–169.

Todosijević, R., Hanafi, S., Urošević, D., Jarboui, B., and Gendron, B. (2017). A general variable neighborhood search for the swap-body vehicle routing problem. *Computers & Operations Research*, 78:468–479.

Toffolo, T. A. M., Christiaens, J., Spieksma, F., and Vanden Berghe, G. (2016a). Grouping sport teams into round robin competitions. In Burke, E. K., Gaspero, L. D., Özcan, E., McCollum, B., and Schaerf, A., editors, *Proceedings of the 11th International Conference of the Practice and Theory of Automated Timetabling (PATAT 2016)*, pages 353–369.

Toffolo, T. A. M., Christiaens, J., Spieksma, F. C. R., and Vanden Berghe, G. (2017a). The sport teams grouping problem. *Annals of Operations Research*, (In press).

Toffolo, T. A. M., Christiaens, J., Van Malderen, S., Wauters, T., and Vanden Berghe, G. (2018). Stochastic local search with learning automaton for the swap-body vehicle routing problem. *Computers & Operations Research*, 89:68–81 (In press).

Toffolo, T. A. M., Esprit, E., Wauters, T., and Vanden Berghe, G. (2017b). A two-dimensional heuristic decomposition approach to a three-dimensional multiple container loading problem. *European Journal of Operational Research*, 257(2):526–538.

Toffolo, T. A. M., Santos, H. G., Carvalho, M. A. M., and Soares, J. (2013). An integer programming approach for the multi-mode resource-constrained multi-project scheduling problem. In G. Kendall, G. Vanden Berghe, . B. M., editor, *Proceedings of the 6th Multidisciplinary International Scheduling Conference (MISTA)*, pages 840–847.

Toffolo, T. A. M., Santos, H. G., Carvalho, M. A. M., and Soares, J. A. (2016b). An integer programming approach to the multimode resource-constrained multiproject scheduling problem. *Journal of Scheduling*, 19(3):295–307.

Toffolo, T. A. M., Van Malderen, S., Wauters, T., and Vanden Berghe, G. (2014). Branch-and-price and improved bounds to the traveling umpire problem. In Özcan, E., Burke, E., and McCollum, B., editors, *Proceedings of the 10th International Conference on Practice and Theory of Automated Timetabling (PATAT 2014)*, pages 420–432, York, UK.

Toffolo, T. A. M., Wauters, T., Van Malderen, S., and Vanden Berghe, G. (2016c). Branch-and-bound with decomposition-based lower bounds for the traveling umpire problem. *European Journal of Operational Research*, 250(3):737–744.

Toth, P. and Vigo, D. (2003). The granular tabu search and its application to the vehicle-routing problem. *INFORMS Journal on Computing*, 15(4):333–346.

Toth, P. and Vigo, D., editors (2014). *Vehicle Routing: Problems, Methods, and Applications*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2nd edition.

Trick, M. A. and Yildiz, H. (2007). Bender's cuts guided large neighborhood search for the traveling umpire problem. In Van Hentenryck, P. and Wolsey, L., editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, number 4510 in Lecture Notes in Computer Science, pages 332–345. Springer Berlin Heidelberg.

Trick, M. A. and Yildiz, H. (2011). Benders' cuts guided large neighborhood search for the traveling umpire problem. *Naval Research Logistics (NRL)*, 58(8):771–781.

Trick, M. A. and Yildiz, H. (2012). Locally optimized crossover for the traveling umpire problem. *European Journal of Operational Research*, 216(2):286–292.

Trick, M. A. and Yildiz, H. (2013). Traveling umpire problem, benchmark instances. Available at http://mat.gsia.cmu.edu/TUP/.

Trick, M. A., Yildiz, H., and Yunes, T. (2012). Scheduling major league baseball umpires and the traveling umpire problem. *Interfaces*, 42:232–244.

Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., and Subramanian, A. (2017). New benchmark instances for the Capacitated Vehicle Routing Problem. *European Journal of Operational Research*, 257(3):845–858.

Uchoa, E., Toffolo, T. A. M., de Souza, M. C., Martins, A. X., and Fukasawa, R. (2012). Branch-and-cut and hybrid local search for the multi-level capacitated minimum spanning tree problem. *Networks*, 59(1):148–160.

Valouxis, C., Gogos, C., Goulas, G., Alefragis, P., and Housos, E. (2012). A systematic two phase approach for the nurse rostering problem. *European Journal of Operational Research*, 219(2):425–433.

Van Den Dooren, D., Sys, T., Toffolo, T. A. M., Wauters, T., and Vanden Berghe, G. (2017). Multi-machine energy-aware scheduling. *EURO Journal on Computational Optimization*, 5(1):285–307.

Vanderbeck, F. and Wolsey, L. (2010). Reformulation and decomposition of integer programs. In Jünger, M., Liebling, T. M., Naddef, D., Nemhauser,

G. L., Pulleyblank, W. R., Reinelt, G., Rinaldi, G., and Wolsey, L. A., editors, *50 Years of Integer Programming 1958-2008*, pages 431–502. Springer Berlin Heidelberg.

Verstichel, J. and Vanden Berghe, G. (2009). A late acceptance algorithm for the lock scheduling problem. In Voss, S., Pahl, J., and Schwarze, S., editors, *Logistik Management, Hamburg, 2-4 September 2009*, pages 457–478.

Vidal, T., Battarra, M., Subramanian, A., and Erdogan, G. (2015). Hybrid metaheuristics for the clustered vehicle routing problem. *Computers & Operations Research*, 58(1):87–99.

Vidal, T., Crainic, T., Gendreau, M., and Prins, C. (2013). Heuristics for multi-attribute vehicle routing problems: A survey and synthesis. *European Journal of Operational Research*, 231(1):1–21.

Vidal, T., Crainic, T., Gendreau, M., and Prins, C. (2014). A unified solution framework for multi-attribute vehicle routing problems. *European Journal of Operational Research*, 234(3):658–673.

Vidal, T., Maculan, N., Ochi, L., and Penna, P. (2016). Large neighborhoods with implicit customer selection for vehicle routing problems with profits. *Transportation Science*, 50(2):720–734.

Villegas, J. G., Prins, C., Prodhon, C., Medaglia, A. L., and Velasco, N. (2013). A matheuristic for the truck and trailer routing problem. *European Journal of Operational Research*, 230(2):231–244.

Wäscher, G., Haußner, H., and Schumann, H. (2007). An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130.

Wauters, T. (2012). *Reinforcement learning enhanced heuristic search for combinatorial optimization*. PhD thesis, KU Leuven.

Wauters, T., Kinable, J., Smet, P., Vancroonenburg, W., Vanden Berghe, G., and Verstichel, J. (2016). The multi-mode resource-constrained multi-project scheduling problem. *Journal of Scheduling*, 19(3):271–283.

Wauters, T., Van Malderen, S., and Vanden Berghe, G. (2014). Decomposition and local search based methods for the traveling umpire problem. *European Journal of Operational Research*, 238(3):886–898.

Weglarz, J. (1999). *Project Scheduling: Recent Models, Algorithms, and Applications*. International series in operations research & management science. Kluwer.

Xue, L., Luo, Z., and Lim, A. (2015). Two exact algorithms for the traveling umpire problem. *European Journal of Operational Research*, 243(3):932–943.

Yildiz, H. (2008). *Methodologies and Applications for Scheduling, Routing & Related Problems*. PhD thesis, Carnegie Mellon University.

Yuan, B., Zhang, C., and Shao, X. (2015). A late acceptance hill-climbing algorithm for balancing two-sided assembly lines with multiple constraints. *Journal of Intelligent Manufacturing*, 26(1):159–168.

Zhao, X., Bennell, J. A., Bektaş, T., and Dowsland, K. (2016). A comparative review of 3D container loading algorithms. *International Transactions in Operational Research*, 23(1-2):287–320.

Awards and publications

Awards

2016 **Winner of Superminds 2016**, Brussels, Belgium.
The imec SuperMinds is a yearly single-track event where state-of-the-art research and innovation projects are presented in TED-like exhibitions by the researchers. The 2016 edition counted with approximately 500 researchers in the audience. Nine pre-selected participants presented their research and competed to the Superminds 2016 award. The prize was given to the best presentation, selected in a voting system by the approximately 500 researchers in the audience. The winning presentation is available online in imec's website, Youtube and Vimeo.

2015 **Best Presentation Award**, Guimarães, Portugal.
Best Presentation Award received with the presentation entitled "Decomposition approaches to 3D container loading problems and strip packing problems", given during the XI International Workshop on Cutting, Packing and Related Topics, July 2015.

2015 **ESICUP Cutting and Packing International Competition**, Portsmouth, UK.
Second place on the ESICUP Cutting and Packing International Competition 2014/2015 with the team CODeS.

2014 **International Vehicle Routing Competition**, Oslo, Norway.
Winner of the VeRoLog Vehicle Routing Competition 2013/2014 with the team CODeS.

2013 **MISTA Challenge** (Project Scheduling Competition), Gent, Belgium.
Third place on the MISTA Project Scheduling Competition 2012/2013 with the team GOAL.

2012 **International Timetabling Competition**, Son, Norway.
Winner of the Internation Timetabling Competition 2011/2012 with the team GOAL.

2006 **Brazilian Computer Society Student Award**, www.sbc.org.br.
Winner of the prize, given by the organization to the featured students of different regions of Brazil.

2006 **Featured Student Award**, Federal University of Ouro Preto, Brazil.
Winner of the prize, given by the University to the graduating student with highest grades.

Publications

Articles in internationally reviewed academic journals

2017 **Toffolo, T. A. M.**, Christiaens, J., Van Malderen, S., Wauters, T., and Vanden Berghe, G. (To appear). Stochastic local search with learning automaton for the swap-body vehicle routing problem. *Computers & Operations Research*. 89:68–81. doi:10.1016/j.cor.2017.08.002 (In press)

2017 **Toffolo, T. A. M.**, Christiaens, J., Spieksma, F., Vanden Berghe, G. (To appear). The sport teams grouping problem. *Annals of Operations Research*. doi:10.1007/s10479-017-2595-z (In press)

2017 **Toffolo, T. A. M.**, Esprit, E., Wauters, T., and Vanden Berghe, G. (2017). A two-dimensional heuristic decomposition approach to a three-dimensional multiple container loading problem. *European Journal of Operational Research*, 257(2):526–538. doi:10.1016/j.ejor.2016.07.033

2017 Van Den Dooren, D., Sys, T., **Toffolo, T. A. M.**, Wauters, T., and Vanden Berghe, G. (2017). Multi-machine energy-aware scheduling. *EURO Journal on Computational Optimization*, 5(1):285–307. doi:10.1007/s13675-016-0072-0

2016 **Toffolo, T. A. M.**, Wauters, T., Van Malderen, S., and Vanden Berghe, G. (2016). Branch-and-bound with decomposition-based lower bounds for the traveling umpire problem. *European Journal of Operational Research*, 250(3):737–744. doi:10.1016/j.ejor.2015.10.004

2016 **Toffolo, T. A. M.**, Santos, H. G., Carvalho, M. A. M., and Soares, J. A. (2016). An integer programming approach to the multimode resource-constrained multiproject scheduling problem. *Journal of Scheduling*, 19(3):295–307. doi:10.1007/s10951-015-0422-4

2016 Santos, H. G., **Toffolo, T. A. M.**, Gomes, R. A. M., and Ribas, S. (2016). Integer programming techniques for the nurse rostering problem. *Annals of Operations Research*, 239(1):225–251. doi:10.1007/s10479-014-1594-6

2016 Santos, H. G., **Toffolo, T. A. M.**, Silva, C. L. T. F., and Vanden Berghe, G. (To appear). Analysis of stochastic local search methods for the unrelated parallel machine scheduling problem. *International Transactions in Operational Research*. (In press). doi:10.1111/itor.12316

2016 Fonseca, G. H. G., Santos, H. G., **Toffolo, T. A. M.**, Brito, S. S., and Souza, M. J. F. (2016). Goal solver: a hybrid local search based solver for high school timetabling. *Annals of Operations Research*, 239(1):77–97. doi:10.1007/s10479-014-1685-4

2012 Uchoa, E., **Toffolo, T. A. M.**, de Souza, M. C., Martins, A. X., and Fukasawa, R. (2012). Branch-and-cut and hybrid local search for the multi-level capacitated minimum spanning tree problem. *Networks*, 59(1):148–160. doi:10.1002/net.20485

2009 Martins, A. X., Souza, M. C., Souza, M. J., and **Toffolo, T. A. M.** (2009). GRASP with hybrid heuristic-subproblem optimization for the multi-level capacitated minimum spanning tree problem. *Journal of Heuristics*, 15:133–151. doi:10.1007/s10732-008-9079-x

Papers at international scientific conferences and symposia, published in full in proceedings (during the PhD)

2017 Gomes, R. A. M., **Toffolo, T. A. M.**, and Santos, H. G. (2017). Variable neighborhood search accelerated column generation for the nurse rostering problem. *Electronic Notes in Discrete Mathematics*, 58:31–38. doi:10.1016/j.endm.2017.03.005

2016 **Toffolo, T. A. M.,** Christiaens, J., Spieksma, F., and Vanden Berghe, G. (2016). Grouping sport teams into round robin competitions. In Burke, E. K., Gaspero, L. D., Özcan, E., McCollum, B., and Schaerf, A., editors, *Proceedings of the 11th International Conference of the Practice and Theory of Automated Timetabling (PATAT 2016)*, pages 353–369.

2016 Araujo, J. A. S., Santos, H. G., Baltar, D. D., **Toffolo, T. A. M.**, and Wauters, T. (2016). Neighborhood composition strategies in stochastic local search. In Blesa, M. J., Blum, C., Cangelosi, A., Cutello, V., Di Nuovo, A., Pavone, M., and Talbi, E.-G., editors, *Hybrid Metaheuristics: 10th International Workshop, HM 2016, Plymouth, UK, June 8-10, 2016, Proceedings*, pages 118–130, Cham. Springer International Publishing.

2014 **Toffolo, T. A. M.**, Van Malderen, S., Wauters, T., and Vanden Berghe, G. (2014). Branch-and-price and improved bounds to the traveling umpire problem. In *Proceedings of the 10th International Conference on Practice and Theory of Automated Timetabling (PATAT 2014)*, pages 420–432, York, UK.

Conferences and symposia (during the PhD)

Presentations at international conferences and symposia

2017 **Toffolo, T. A. M.**, Wauters, T., Martinez-Sykora, A. A decomposition-based algorithm for a leather industry cutting problem. *XII International Workshop on Cutting, Packing and Related Topics.* IWCPRT 2017, Gent, Belgium – September 13, 2017.

2017 **Toffolo, T. A. M.**, Wauters, T., Martinez-Sykora, A. Matheuristics for a real-world leather industry cutting problem. *14th EURO Special Interest Group on Cutting and Packing Meeting.* ESICUP Meeting 2017, Liège, Belgium – May 05, 2017.

2017 **Toffolo, T. A. M.** Decomposition-based Branch-and-Bound for the Traveling Umpire Problem. *Escuela Latinoamericana de Verano en Investiación Operativa.* ELAVIO 2017, Buenos Aires and Miramar, Argentina – February 24, 2017.

2016 **Toffolo, T. A. M.**, Christiaens, J., Spieksma, F., and Vanden Berghe, G. Assigning youth football teams to leagues. *28th European Conference on Operational Research.* EURO 2016, Poznan, Poland – July 04, 2016.

2016 **Toffolo, T. A. M.**, Wauters, T., and Vanden Berghe, G. Exact and heuristic approaches to the Traveling Umpire Problem. *1st Workshop on Applied Combinatorial Optimization Methods.* WACOM 2016, Ouro Preto, Brazil – March 21, 2016.

2016 **Toffolo, T. A. M.**, Christiaens, J., Spieksma, F., and Vanden Berghe, G. Grouping sport teams into round robin competitions. *11th International Conference on Practice and Theory of Automated Timetabling.* PATAT 2016, Udine, Italy – August 25, 2016.

2015 **Toffolo, T. A. M.**, Wauters, T., and Vanden Berghe, G. Time-based Decomposition Strategies for the Traveling Umpire Problem. *7th Multidisciplinary International Scheduling Conference.* MISTA 2015, Prage, Czech Republic – August 25, 2015.

2015 **Toffolo, T. A. M.**, Wauters, T., Esprit, E., and Vanden Berghe G. Decomposition approaches to 3D container loading problems and 2D strip packing problems *XI International Workshop on Cutting, Packing and Related Topics.* IWCPRT 2015, Guimarães, Portugal – July 31, 2015.

2015 **Toffolo, T. A. M.**, Wauters, and Vanden Berghe G. Decomposition-based Branch-and-bound for the Traveling Umpire Problem. *Joint International Meeting Canadian Operational Research Society Institute for Operations Research and the Management Sciences.* CORS/INFORMS Meeting 2015, Montreal, Canada – June 14, 2015.

2015 **Toffolo, T. A. M.**, Wauters, T., Esprit, E., and Vanden Berghe G. A heuristic decomposition approach to the ESICUP 2015 Challenge Container Loading Problem. *12th EURO Special Interest Group on Cutting and Packing Meeting.* ESICUP Meeting 2015, Portsmouth, UK – March 31, 2015.

2014 **Toffolo, T. A. M.**, Santos, H. G., Carvalho, M. A. M., Soares, J. A., and Vanden Berghe, G. Hybrid Integer Programming Heuristic to a Generalized Project Scheduling Problem. *VIII ALIO/EURO Workshop on Applied Combinatorial Optimization.* ALIO/EURO 2014, Montevideo, Uruguay – December 08, 2014.

2014 **Toffolo, T. A. M.**, Santos, H. G., Carvalho, M. A. M., Soares, J. A., Vanden Berghe, G. and Wauters, T. An Integer Programming Approach for a Generalized Project Scheduling Problem. *5th International Workshop on Model-Based Metaheuristics.* Matheuristics 2014, Hamburg, Germany – June 12, 2014.

2014 **Toffolo, T. A. M.**, Santos, H. G., Carvalho, M. A. M., Soares, J. A., Vanden Berghe, G. and Wauters, T. An Integer Programming Approach for a Generalized Project Scheduling Problem. *New Challenges in Scheduling Theory.* Aussois, France – April 03, 2014.

2014 **Toffolo, T. A. M.**, Van Malderen, S., Wauters, T., and Vanden Berghe G. Branch-and-Price and Improved Bounds to the Traveling Umpire Problem. *10th International Conference on Practice and Theory of Automated Timetabling.* PATAT 2014, York, UK – August 27, 2014.

Presentations at other conferences and symposia

2017 **Toffolo, T. A. M.**, Wauters, T., and Martinez-Sykora, A. Integer programming based heuristics for a leather indystry nesting problem. *31st Annual Conference of the Belgian Operational Research Society.* ORBEL 2017, Brussels, Belgium – February 02, 2017.

2017 Heshmati S., **Toffolo T. A. M.**, Vancroonenburg W., and Vanden Berghe G. Heuristics for crane scheduling and location assignment problems in automated warehouses. *31st Annual Conference of the Belgian Operational Research Society.* ORBEL 2017, Brussels, Belgium – February 02, 2017.

2016 **Toffolo, T. A. M.**. What's your problem? *SuperMinds 2016.* Brussels, Belgium – October 27, 2016.

2016 **Toffolo, T. A. M.**, Christiaens, J., Spieksma, F., and Vanden Berghe, G. Geographically grouping youth teams into football leagues. *30th Annual Conference of the Belgian Operational Research Society.* ORBEL 2016, Louvain-la-Neuve, Belgium – January 28, 2016.

2015 **Toffolo, T. A. M.**, Wauters, T., Van Malderen, S., and Vanden Berghe, G. Decomposition-based Branch-and-bound for the Traveling Umpire Problem. *29th Annual Conference of the Belgian Operational Research Society.* ORBEL 2015, Antwerp, Belgium – Febrary 05, 2015.

2014 **Toffolo, T. A. M.**, Santos, H. G., Martinelli, R., Gomes, R. A. M., and Vanden Berghe, G. Integer Programming Techniques for the Nurse Rostering Problem. *28th Annual Conference of the Belgian Operational Research Society.* ORBEL 2014, Mons, Belgium – January 30, 2014.

Code and data published online (during the PhD)

* **Traveling Umpire Problem**
  Source code of the state-of-the-art solvers proposed for the TUP:
  – http://github.com/tuliotoffolo/tup

* **Nurse Rostering Problem**
  Source code of the state-of-the-art solver proposed for the NRP:
  – http://github.com/tuliotoffolo/nrp

* **General MIP decomposition-based heuristic framework**
  Source code of the general decomposition-based heuristic framework:
  – http://github.com/tuliotoffolo/jads

* **Robust no-fit polygon library**
  Source code of a robust no-fit polygon library including a visualization tool:
  – http://github.com/tuliotoffolo/jnfp

* **Unrelated Parallel Machine Scheduling Problem**
  Source code of a state-of-the-art solver for the Unrelated Parallel Machine Scheduling Problem (Santos et al., 2017):
  – http://github.com/tuliotoffolo/upmsp

* **Additional data, solver binaries and tools**
  Additional data (formulation, instance and solution files), solver binaries and visualization tools published online throughout the PhD:
  – https://benchmark.gent.cs.kuleuven.be/mclp
  – https://benchmark.gent.cs.kuleuven.be/nrp
  – https://benchmark.gent.cs.kuleuven.be/psp
  – https://benchmark.gent.cs.kuleuven.be/sbvrp
  – https://benchmark.gent.cs.kuleuven.be/stgp
  – https://benchmark.gent.cs.kuleuven.be/tup

FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE
COMBINATORIAL OPTIMISATION DECISION SUPPORT (CODES)
Celestijnenlaan 200A box 2402
B-3001 Heverlee