THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

On scheduling using optimizing SMT-solvers

Sabino Francesco Roselli

CHALMERS
UNIVERSITY OF TECHNOLOGY
Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2020

On scheduling using optimizing SMT-solvers

Sabino Francesco Roselli

Department of Electrical Engineering

Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
Phone: +46 (0)31 772 1000
www.chalmers.se

Printed by Chalmers Reproservice
Gothenburg, Sweden, September 2020

To those who care about me, and those who care about optimization.

Abstract

Modern production systems are becoming more complex by the year and flexibility of production is one of the key factors to success. Companies want to be able to provide a customized product that fits exactly the customer requirements and, therefore production systems have to be able to produce a wide range of product variants. This introduces additional complexity in the production system,

Among the problems that companies have to deal with is the vehicle routing problem (VRP), which is the problem of scheduling routes for vehicles to serve customers according to predetermined specifications, such as arrival time at a customer, amount of goods to deliver, etc. The problem is industrially relevant since material needs to be delivered from warehouses to the production lines and as the plants grow in size, managing an ever growing fleet of vehicles is becoming a challenge.

Due to the complexity of the requirements and the increasing size of the transportation systems, it is no longer feasible to solve these problems manually, since the variables and constraints to keep into account are simply to many. Modern computers can provide feasible schedules much faster than human beings and for this reason companies are willing to pay a high fee to use the cutting edge scheduling solvers on the market.

Among the class of general purpose solvers, we find mixed integer linear programming (MILP) solvers and satisfiability modulo theory (SMT) solvers. They are not designed to solve one specific problem, but entire classes of problems. In particular, both MILP and SMT solvers can handle mixed integer linear models and, since the VRP can be described by a mixed integer linear model, both MILP and SMT qualify as suitable tools to deal with it.

In this work, we run extensive computational analysis over well-known combinatorial optimization problems to compare the performance of state-of-the-art MILP and SMT solvers and to better understand their strengths and weaknesses. Based on the acquired knowledge we design a monolithic model for a real-world VRP. While the model can be used to find the *true optimum*, it does not scale well with the problem size. We therefore develop a compositional algorithm that can handle much larger problems at the cost of having sub-optimal yet good solutions.

Keywords: Job Shop, Vehicle Routing, Bin Sorting, SMT, MILP.

List of Publications

This thesis is based on the following publications:

[A] **Sabino Francesco Roselli**, Kristofer Bengtsson, Knut Åkesson, "SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation". Proceedings of 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE), Munich, Germany.

[B] **Sabino Francesco Roselli**, Kristofer Bengtsson, Knut Åkesson, "SMT Solvers for Flexible Job-Shop Scheduling Problems: A Computational Analysis". Proceedings of 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE), Vancouver, Canada..

[C] **Sabino Francesco Roselli**, Kristofer Bengtsson, Knut Åkesson, "Compact Representation of Time-Index Job Shop Problems Using a Bit-Vector Formulation". Published in 2020 IEEE 16th International Conference on Automation Science and Engineering (CASE), Hong Kong.

[D] **Sabino Francesco Roselli**, Fredrik Hagebring, Sarmad Riazi, Martin Fabian, Knut Åkesson, "On the Use of Equivalence Classes for Optimal and Sub-Optimal Bin Packing and Bin Covering". Conditionally accepted to 2020 IEEE Transactions on Automation Science and Engineering (TASE).

Other publications by the author, not included in this thesis, are:

[E] **Sabino Francesco Roselli**, Fredrik Hagebring, Sarmad Riazi, Martin Fabian, Knut Åkesson, "On the Use of Equivalence Classes for Optimal and Sub-Optimal Bin Covering". *Proceedings 15th IEEE Conference on Automation Science and Engineering (CASE)* Vancouver, Canada, Aug. 2019.

[F] E. Jernheden, C. Lindström, R. Persson, M. Wedenmark, E. Eros, **S.F. Roselli**, K, Åkesson, "Comparison of Exact and Approximate methods for the Vehicle Routing Problem with Time Windows". *Proceedings 16th IEEE Conference on Automation Science and Engineering (CASE)* Hong Kong, Aug. 2020.

Acknowledgments

This work has been supported by EUREKA ITEA3-projektet ENTOC (Label nr. 15015, Engineering Tool Chain for Efficient and Itertive Development of Smart Factories) and Vinnova, Chalmers AI Research Centre (CHAIR) and AB Volvo (Project ViMCoR)

Acronyms

EO:	Exactly One
EN:	Exactly n
AMO:	At Most One
SAT:	Boolean Satisfability
SMT:	Satisfiability Modulo Theory
OMT:	Optimization Modulo Theory
MILP:	Mixed Integer Linear Programming
JSP:	(Standard) Job Shop Problem
FJSP:	Flexible Job Shop Problem
VRP:	Vehicle Routing Problem
VRPTW:	Vehicle Routing Problem with Time Windows
ATR:	Automated Transportation Robot
BSP:	Bin Sorting Problem
BCP:	Bin Covering Problem
BPP:	Bin Packing Problem

Contents

Part I

Overview

CHAPTER 1

Introduction

The past few decades have witnessed a change of paradigm: the main production resources are no longer human beings, but robots and computers. This is happening not only on the shop floor, but at every echelon of the production, including the planning area. Machines can execute calculations much faster than humans and (within certain limits) they are more reliable.

This may sound threatening as machines are taking over more and more jobs that were exclusive to humans until only a few decades ago, but it is actually beneficial for people. Robots are nowadays advanced enough to execute long, repetitive (and boring) tasks with great accuracy, relieving the worker from the alienation of a never-changing environment; computers can schedule the production months ahead, taking just a few minutes and saving hours (or most likely days) of pen-and-paper work. People will be free therefore to dedicate their time to more rewarding and creative tasks.

One field that has drawn the attention of both academia and industry in the last decades is the vehicle routing problem (VRP). It is the problem of designing routes for a fleet of vehicles to visit and serve customers according to pre-specified constraints.

There are many different sectors that could take advantage of efficient

scheduling of delivery vehicles: from long haul trucks shipping goods across continents, to transport carts managing the supply of components to the assembly line, to fleets of taxis serving customers all over the city.

Nowadays transportation systems are growing bigger and bigger; the warehouse of a plant can host hundreds of vehicles; the transport service of a city can have just as many buses. Moreover, there can be a number of additional constraints that complicate the system: not all vehicles are feasible to serve a certain customer; some customers must be served not later than a certain time; some customers must be served before others, etc. It is unthinkable for humans to be able to schedule the routes manually.

However, even for the most advanced computers, solving a VRP can be challenging. The large size and the additional requirements of a modern scenario have added so many degrees of freedom to the system, that the number of possible combinations can be in the same order as the number of atoms in the universe.

Fortunately (and this not only applies to the VRP, but to all complex systems), one does not have to check all possible combinations in order to find the ones that meet the desired requirements. There exist very efficient approaches to cut off entire portions of this large search-space and only focus on the regions that contain interesting solutions [1]. For particular problems, there are specific purpose algorithms that can solve large (and very large) problem instances in reasonable time. The drawback of these algorithms is their specificity: they are designed for one particular problem. In the real world, for every problem, there may exist a thousand variations and it is often the case that, no matter how close these variations are to the original problem, each needs a specific algorithm. For instance, there are different approaches to solve the job shop problem (JSP) (further details about the JSP are given later in this work), but each approach has to be tailored to the specific version of the JSP. For instance, [2] and [3] present a genetic algorithm for different versions of the JSP, while in [4], a bee colony algorithm is used for the standard JSP.

This is not always the case though. In fact, there exist general purpose solvers that can deal with entire classes of problems using the same set of algorithms. An example is MILP solvers [5], that can be used to solve models of any real-world system, as long as the system behaviour can be described with linear equations. It is not a coincidence that MILP solvers are nowadays

a well established and widespread approach to solve industrial problems [6], [7] and [8], to name a few. As a matter of fact, a VRP can be described by a linear model and, therefore, solved by a MILP solver.

Another important truth about modern production can be effectively expressed by the following quote:

"Chances are, if something can be expressed as a mathematical equation, then at some point somebody will want to minimise it." [9]

In fact, almost every process, schedule, action, etc., can be designed in more than one way, and just as often it is possible to identify an objective function related to some process parameter; it could be some direct cost, or manufacturing time (very often it is somehow related to the profit the company wants to make, either by maximizing some positive factor, or by minimizing costs); this could also be environment related, such as overall CO_2 emissions. No matter what the objective function is, the goal is to find, among all possible solutions, the one that optimizes (minimizes or maximizes) it. In the VRP, a factor to minimize could be the overall travelling distance, or the number of vehicles, while a factor to maximize could be the number of goods delivered per time unit.

The good news is that MILP solvers are perfectly able to do that. The bad news is that finding the optimum may be very hard work [10] and sometimes take days, weeks, or even months. This is especially true when dealing with non-real variables that, in many industrial problems, are just as common as the real ones. For instance, in any VRP problem, binary and/or integer variables are required to model choices over different routes.

The primary algorithm running under the hood of a MILP solver is Simplex [11]. It is about 60 years old, its theoretical complexity is non-polynomial in the worst case and yet it works surprisingly well in practice. The main problem is that it is designed to work with real variables. The only way to use it when integrality constraints are involved, is to solve multiple *relaxed* problems where the integrality is neglected. Having n integer variables with domain size k means that, in the worst case we need to solve k^n relaxed problems. In real problems, n could easily be in the order of tens of thousands, and k could be just as large. Of course modern solvers are smart and they can cut off entire regions of this large search-space as they run, so they do not need to check all these combinations, but this is unfortunately not enough in many cases.

On the other hand, MILP is not the only approach that provides a general

framework to model linear systems [12]; Among others, satisfiability modulo theory (SMT) solvers are nowadays a viable alternative and have in quite a few cases showed interesting performance when dealing with industrial problems, as in [13] and [14].

SAT solvers were originally employed for hardware and software verification [15] but the research in other areas proved them suitable to solve industrial problems as well. Unlike MILP solvers, they are naturally built to handle systems described by discrete variables [16], so we can expect interesting results in the comparison of SMT solvers against MILP solvers on problems where the integrality constraint is the bottleneck of the search.

SMT solvers owe their efficiency to the SAT technology; SAT solvers are specifically designed to handle Boolean satisfiability problems, and nowadays they can solve problems counting up to millions of variables [17]; Unfortunately modelling a problem using only Boolean variables can be a real challenge and can also lead to a model that is just too big (even for a SAT solver).

This is why, since the early 2000, research started focusing on how to take advantage of the SAT solvers speed to solve problems formulated using more than Boolean variables. Researchers came up with the idea of combining a SAT engine and a *theorem prover* (further details about the relation between SAT and SMT are prvided in Chapter 3).

In this way SMT solvers can exploit the speed of SAT solvers in dealing with large models, while still allowing for constraints that involve different theories, from Linear algebra, to graphs, from propositional logic to bit-vectors.

Another important aspect of modern SMT solvers is their ability to handle optimization problems (which also makes SMT and MILP direct competitors): as the name points out, Boolean Satisfability is mainly concerned about finding satisfiable solutions, and so were the early SMT solvers; it did not take long though, before researchers started thinking about how to drive the search towards optimal solutions. Soon after the first *optimizing SMT solvers* were born (together with the neologism *optimization modulo theory* (OMT)).

So, when it comes to the type of problems that SMT and MILP solvers can handle, they are equivalent. On the other hand, many industrial problems require discrete modelling and logical reasoning (implemented using binary or Boolean variables) and some of these problems are not easy to solve using MILP solvers, while others have the "right structure" to exploit MILP strengths. It is therefore interesting to test these different solvers and find

out more about their performance depending on the problem. A better understanding on their strengths and weaknesses could be exploited to design an efficient algorithm for the VRP.

1.1 Research Questions

This thesis explores the following research questions:

RQ1 What are the strengths and weaknesses of SMT solvers and how do they compare to MILP solvers when used to solve industrial problems ?

SMT solvers were born within the computer science community and they were originally employed for software and hardware verification. Only recently they have been used as solving tools in other fields. Therefore testing them on different industrial problems could reveal their strengths and weaknesses.

Also, there is experimental evidence that some problems have a structure that allows MILP solvers to find the optimum very quickly, while other problems are intrinsically harder to solve. The same can be said about SMT solvers. It is interesting to compare SMT solvers against MILP solvers on different classes of problems and find out whether one technology outperforms the other, when and, possibly, why.

RQ2 How can the strengths of SMT and/or MILP solvers be exploited and combined to design an efficient algorithm for a real-world VRP?

A real-world VRP can involve constraints related to specific plant's features; a scenario involving automated vehicles that have to travel through a plant where workers and other equipment are present, charge their batteries if they run low on energy and deliver goods in time is a challenging problem. A good strategy to solve it in reasonable time is to break it down into sub-problems. Since each sub-problem is solved on its own, different strategies can be applied depending on the problem features. MILP and/or SMT could be used as back-end solvers in a compositional algorithm designed to find a solution to the overall problem by iteratively solving the sub-problems. The method does not guarantee to find the optimum but it can still provide very accurate solutions while significantly shortening the running time.

1.2 Method

In order to gather information about MILP and SMT different problem classes have been studied: job shop [18], vehicle routing [19], and bin sorting (BSP) [20]. For each of these problems, benchmark instances are available. In this work, mathematical models have been formulated, based on existing literature about the problems and then solved using both MILP and SMT solvers. Comparison is based on running time necessary to find the optimum, or accuracy (gap between optimum and current best estimate) when timeout is reached.

For a VRP with specific requirements, we formulated a monolithic model and tested it on some generated problem instances using SMT; We then broke down the problem into sub-problems and for each of them, we formulated mathematical models and then solved them using SMT; models and results are presented in Chapter 5. Such models are used in a compositional algorithm to find a solution to the overall problem.

1.3 Outline

This thesis consists of two parts. Part I is a general introduction to the field and puts the appended papers into context. Part II contains the appended papers. Part I is organized as follows: Chapter 2 gives an overview on VRPs in general and on the specific industrial setup for which the compositional algorithm is designed. Chapter 3 gives a background on MILP and SMT. In Chapter 4 the results achieved in the appended papers are presented and a bigger picture is drawn about usefulness of MILP and SMT when dealing with different problems. Chapter 5 introduces the monolithic model and compositional algorithm developed to solve the VRP presented in Chapter 2. Chapter 6 summarizes contributions of the included papers. The thesis ends with some closing remarks and directions for future work in Chapter 7.

CHAPTER 2

The Vehicle Routing Problem

In this chapter, a more detailed overview on VRPs in general is given, with a particular focus on the industrial setup that is used as a real-world scenario for which the compositional algorithm is designed.

2.1 The ViMCoR Project

The assumptions we made when designing the industrial setup were motivated by the need of meeting the requirements of the ViMCoR project. The project has the goal of implementing a fleet of automated transportation robots (ATRs) whose purpose is to feed an assembly line with components from a detached storage house as showed in Figure 2.1. This way the components do not have to be kept close to the assembly line, making the environment more worker-friendly; especially in a plant where products have a high degree of variability and many variants can be produced, it is beneficial to not have all the components stored closed to the assembly line. In the ViMCoR context we can talk about pickup-delivery tasks, since the ATRs always have to visit one or multiple storage locations before delivering the material to the due workstation. Also, there are time windows for the delivery: vehicles

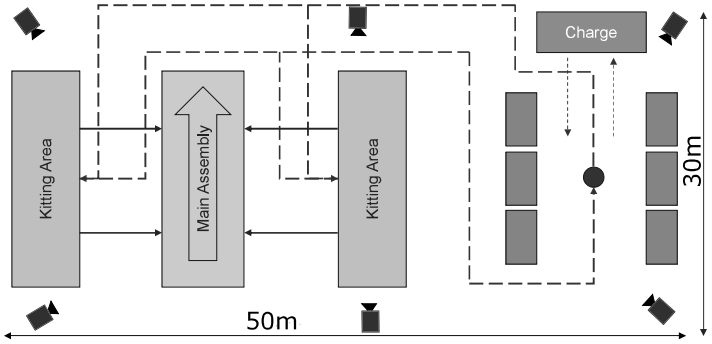

Figure 2.1: ViMCoR industrial setup.

should never be later than the latest arrival time, otherwise they will cause disruptions in the production, but at the same time they should not be too early, in order to avoid long queues at the workstations.

In the ViMCoR scenario, ATRs run in the plant where human workers and human-driven vehicles are also executing their tasks; hence, changes in the plant layout are expected to happen and the vehicles have to be able to react to these changes and deliver the goods in time anyway.

The idea to build a robust system is to install cameras on the plant ceiling that can detect changes in the environment and update a map were the information about the drivable road segments is available. A real-time scheduler has to instruct the vehicles about the routes to follow in order to pick up and delivery goods and every time something changes in the plant layout, the scheduler has to "adjust" the routes to handle the changes. Vehicles will also be equipped with a low-level controller to ensure collision avoidance.

The ATRs are henceforth simply called *vehicles*, since this is the general term used when discussing the VRP.

2.2 Overview on the VRP and industrial setup specifications

The Vehicle Routing Problem (VRP) is the combinatorial optimization problem of computing routes to serve customers while minimizing a cost function, typically the travelled distance, or the number of vehicles required (or a com-

bination of both). Customers are also characterized by a *service time*, which is the time the vehicle serving the customer has to spend at the customer's location in order to serve it. There exist many different extensions of the basic problem, involving additional constraints such as limited capacity of goods that a vehicle can deliver (this corresponds to specific demands of goods for each customer), limited operating range of the vehicles in terms of travelled distance, time windows to deliver goods (i.e. earliest and latest arrival time at a customer), multiple depot stations, etc. A common trait when dealing with the VRP is to treat the real world map as a graph, where each point of interest (i.e. depots, customers) is a node and two nodes are connected with each other by a weighted edge representing their distance. In the VRP, for a route to be valid, each node has to be visited exactly once (in order to make sure that routes start and end at the depot). Further details about VRPs and previous studies on the subject are presented in Paper F.

In order to meet the ViMCoR requirements, we can make some assumptions about the system:

- the goal of the scheduler is to make sure all jobs are completed; for a job to be completed a vehicle has to be assigned to it. A job is composed by one or more pickup tasks and one delivery task. Pickup tasks do not usually have precedence constraints among them (unless specified otherwise), while the delivery for one job always happens after all pickups for that job. Therefore pickup tasks of the same job can be executed in different order, and the delivery task of the same job is executed after them;
- when a vehicle can be assigned to more than one job, it has to execute the job's tasks sequentially; all tasks of a job must be completed before it can execute any task of another job;
- customers of the VRP are either pickup or delivery stations. Executing a task means either pick material from or deliver material to the task location;
- there is only one depot station. All routes have to start and end at the depot;
- vehicles are powered by batteries with limited capacity but with the ability to recharge at a charging station. It is assumed that charging

and discharging of the batteries is linear and the only charging station is the depot;

- not all vehicles are eligible to execute all jobs. On the other hand, transportation capacity of the vehicles is not taken into account since it is assumed that if a vehicle is eligible for a task, then it can carry the corresponding material. Also, it does not matter how many jobs a vehicle is assigned to, since each of the jobs ends with a delivery, which means delivering all the material and going back to full transportation capacity;
- different road segments in the plant have different capacities in terms of number of vehicles they can accommodate.
- the service time is assumed to be zero since the pickup and delivery time can be considered negligible compared to the travelling time.

Moreover, the model is meant to provide a high level schedule for the production taking place during one or more shifts, and vehicle have a limited operating range, hence it is reasonable to assume that vehicles may run out of charge and it is therefore necessary to plan for charging time as well. The subject is discussed further in the next section.

Another important aspect of the project is the conflict-free planning:

- *capacity constraint*: two or more vehicles can never occupy the same physical spot, therefore n vehicles cannot start traversing, at the same time, a road segment that allows for $n-1$ vehicles travelling in the same direction;
- *deadlock avoidance*: the plant layout may have sections that are geometrically unfit for the transit of two vehicles in opposite directions (i.e. narrow aisles or sharp turns). While the idea is to have a lower level controller guaranteeing collision avoidance, deadlocks may still occur and therefore a centralized scheduler is required to avoid them.

2.3 Operating Range and Conflict-free Routing

In order to model the above mentioned type of system, the focus is on two main aspects: vehicles' limited operating range, and conflict-free routing.

Operating Range and Charging Stations

When it comes to vehicles' operating range, there can be two possible scenarios of interest:

- The round-trip between the depot and each customer is shorter than the vehicles operating range, therefore vehicles can always serve at least one customer and go back to the depot before running out of charge. If this is the case there are two possible alternatives: either the vehicle can be recharged at the depot or there have to be enough vehicles to satisfy the customer's demand.

- The round-trip between the depot and each customer is longer than the vehicles' operating range, therefore vehicles have to recharge along the way. The first attempts to model rechargeable vehicles date back to the late 90s, [21], and models were subsequently extended in [22] and [23]. When representing the problem as a graph, charging stations are special nodes where the vehicle can charge its battery on its way to some customer. Since vehicles have to to visit customers exactly once, the solver may misjudge an instance (i.e. declare it unfeasible while it is actually feasible) because a feasible solution could require the same vehicle to visit the same charging station twice or more or not at all. In order to avoid this mistake, it is necessary to define *dummy* charging stations: copies of the real ones, that are located in the same spot, so that the same vehicle can go back to the same location to recharge without having to visit the same node more than once. [23] developed a model formulation based on the concept of dummy charging stations to implement a hybrid local and tabu search. The model has been extended further by [24], including additional requirements on the cost function to reduce the energy consumption. Ever since, due to the faster and faster spread of electric vehicles as well as car sharing services, there has been a great effort in finding scalable algorithms for the VRP. In [25], a hybrid Adaptive Large Neighbourhood Search is proposed; in [26] the focus is on optimizing the vehicle distributions and fleet sizes for Mobility-on-Demand systems; in [27] models and coordination policies for fleets of self-driving vehicles combined with public transit are discussed. Further details about how models for the VRP are formulated are given in Chapter 5

Path Planning and Conflict-Free Routing

Designing a model for the VRP that will yield a conflict-free solution involves some challenges because it is necessary to know where each vehicle is at any time; one way to handle this is time and space discretization. It is well known within the community of scheduling and optimization that discretization often leads to state-space explosion even for relatively small problem instances. Thus, discretization is not suitable for large systems involving a high number of vehicles, customers or a long time horizon. Nonetheless, it is possible to build efficient algorithms based on discretized time-and-space models: one of the most important contributions to the field is [28], where the authors used column generation [29] to deal with the VRP. Another remarkable contribution is [30], where the master problem is split into two sub-problems, a scheduling one and a routing one and then solved by a constraint programming approach [31] and a MILP solver respectively. More recent works on the topic are [32] and [33], where the authors employ local search and MILP. In [34] the conflict-free routing VRP is combined with the storage allocation problem [35].

To the best of our knowledge, there is no work focusing on both vehicle's operating range and conflict-free routing. With this work, we have the opportunity to fill up this gap in the literature and at the same time formulate a model that suits the ViMCoR requirements. We will use the *total travelled distance* as cost function to evaluate the quality of the results.

CHAPTER 3

On Satisfiability Modulo Theory and Mixed Integer Linear Programming

As MILP and SMT play an important role in this thesis, it is worth giving an overview on them. The purpose of the following sections is to present the main concepts regarding the two approaches; therefore no proofs are provided nor any in-depth explanation. Instead, we refer the reader for relevant literature.

3.1 Mixed Integer Linear Programming

In order to understand how MILP works, it is necessary to take a step back and look first at linear programming (LP). The difference between MILP and LP is that in LP all variables have to be real, while in MILP they can also be integer, or binary. A linear program is a conjunction of linear inequalities and a linear objective function over a set of real variables. As a convention, a linear program is a minimization, though it can easily be converted into maximization (or the other way around) by changing the sign of the objective function.

A general linear program counting n variables and m constraints takes the

form:

$$\begin{aligned} \min\ & c^T\vec{x} \\ \textit{such that}\ & A\vec{x} \leq b \\ & \vec{x} \geq 0 \end{aligned}$$

where $\vec{x} = [x_1, \cdots, x_n]$ is a vector containing the decision variables, $c^T = [c_1, \cdots, c_n]$ is an array of coefficients for the objective function, b is a column vector of the right side values, and $A_{m,n}$ is the coefficient matrix.

An example with only two variables can be represented on the plane where each constraint is a line and since the program is a conjunction of them, they all "work" together to define an area called the *feasible region*, a portion of the plane where all possible solutions to the problem lie.

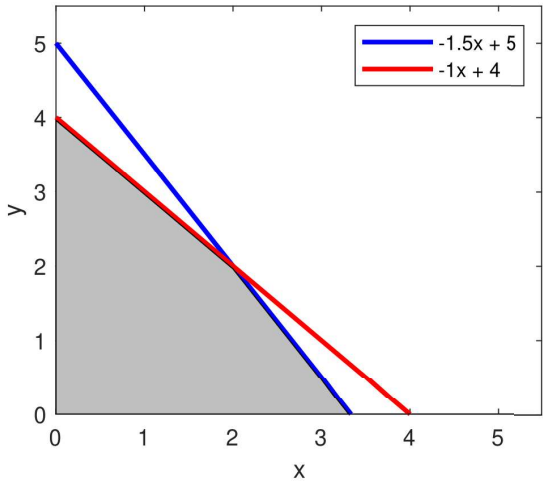

Figure 3.1: Illustration of feasible region for LP problem.

In Figure 3.1, the two constraints (blue and red line), together with the non-negativity of the variables, define the feasible region (gray area): any feasible solution to the program lies within that area (i.e. any point in the white area conflicts with at least one constraint). This is why it is important to have inequalities rather than equalities: they point out which side of the

line is feasible and which one is not.

Since all constraints are linear, the feasible region has the shape of a convex polygon and the optimal solution lies in one of its extreme points.

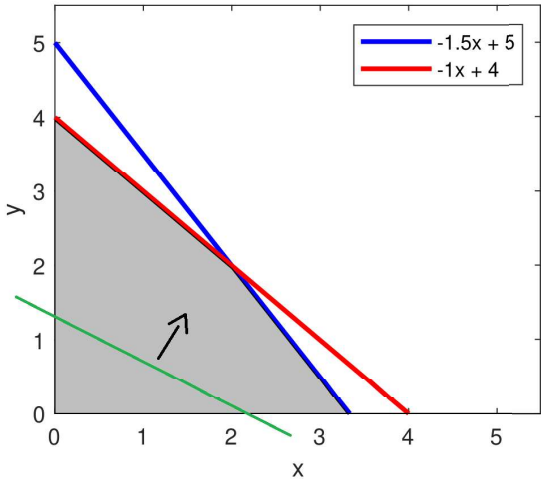

Figure 3.2: Illustration of objective function for LP problem.

The objective function defines which vertex represents the optimal solution (the green line in Figure 3.2) by shifting it along its perpendicular upwards if the problem is a maximization or downwards if it is a minimization. Eventually the objective function will exit the feasible region at a vertex, which is then the optimal solutions. In some cases, the objective function gradient line will be parallel to some constraint, and in that case the two vertices and all points in between them are optimal solutions. In the example above, vertices are in $(0,0), (10/3,0), (2,2), (0,4)$; since the problem is a maximization, we want to shift the green line upwards, so it will end up in the vertex $(2,2)$.

Things become more complicated if the variables are restricted to be integer, because there is no guarantee that the optimum lies on a vertex. Figure 3.3 shows again the feasible region of a program but this time the integrality constraint applies and only the points marked by a star are actually feasible. In this case, the extreme point corresponding to the optimal solution, happens to be integer, therefore solving the relaxed LP would yield a feasible solution

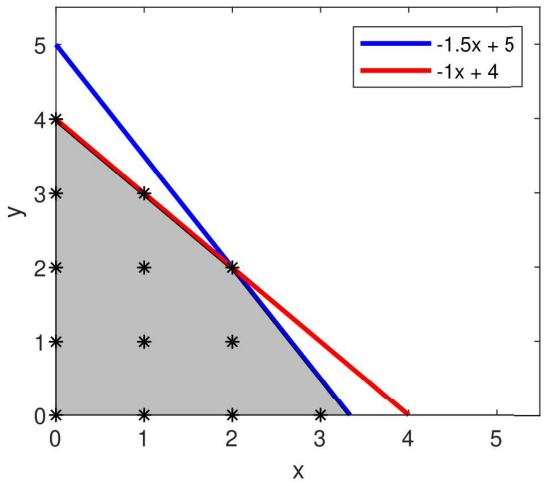

Figure 3.3: Illustration of objective function for LP problem.

to the MILP problem too.

One possibility to solve a mixed integer linear program is to apply the branch and bound algorithm; based on the assumption that solving a relaxed version of the program (i.e. the integrality constraint is removed) yields a solution at least as good or better than the solution of the original problem, the algorithm iteratively solves relaxed problems to find tighter and tighter bounds for the original one, until the bounds overlap.

Once again, an example can be used to clarify the procedure:

$$\begin{aligned} \max\ & 50x_1 + 20x_2 + 60x_3 \\ s.t.\ & 2x_1 + x_2 + 2x_3 \leq 120.5 \\ & 3x_1 + 2x_2 + 2x_3 \leq 150 \\ & x_1 \geq 20 \\ & x_1, x_2, x_3 \geq 0 \\ & x_1, x_2, x_3 \in \mathbb{N} \end{aligned}$$

In the program above the integrality constraint over the variables is included; this means that the objective function value may be fractional, de-

pending on the coefficients (this is not the case here since they are all integer as well), but the values of the decision variables x_1, x_2 and x_3 must be integer.

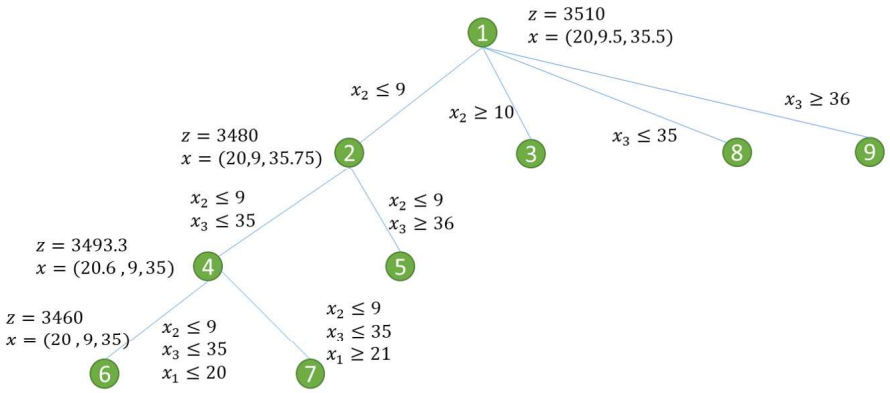

Figure 3.4: Illustration of branch and bound steps.

Figure 3.4 shows some of the steps that the branch and bound algorithm would take to find the optimal solution to the program. In step 1 the relaxed problem is solved and the optimal value is 3510 but the solution is not feasible for the original problem because x_2 and x_3 are not integer. This leads to four alternatives, represented by nodes 2, 3, 8, and 9 respectively. Basically an additional inequality is added that restricts the variable domains to be either smaller than or equal to the floor or larger than or equal to the ceiling of its current value. In node 2, x_2 is restricted to be smaller than or equal to 9 (its value from the previous iteration was 9.5). The relaxed problem with the additional constraint is solved again and this time the optimum is 3480 but the solution is still unfeasible for the original problem, since x_3 is 35.75. We can branch again by adding another inequality: either x_3 is larger than or equal to 36 (ceiling of 35.75) or it is smaller than or equal to 35 (floor of 35.75).

The process goes on until a feasible integer solution is found, as happens in node 6. Now we have a lower bound for the original problem $\underline{z} = 3460$. From now on, every time we explore a branch we can stop whenever we found a solution whose value is smaller than the current $\underline{z}$ and prune (stop exploring) that branch because we know that no better solution can be found there (when

going down a branch we can only find worse and worse solutions because we keep shrinking the feasible region).

Every time an integer solution is found, the search on that branch is over. If the optimal value for the relaxed problem is better than the current lower bound, this becomes the new lower bound and the search starts on another branch. There exist heuristics to decide which branch to pick for the search. The algorithm terminates when all branches are either searched or pruned. Note that this example concerns the maximization of the objective function. For minimization the lower bound is found by relaxing the problems and integer solutions provide upper bounds.

There exist techniques that build on top of the branch and bound algorithm to increase the performance but as of today, solving a mixed integer linear program is computationally demanding and many problems still remain intractable [36]. The reader is referred to [37] for further details.

3.2 Satisfiability Modulo Theories

In this section a background on SMT is given; the focus is on describing how SMT solvers find satisfiable assignments. The search for optimal solutions is not description in details because there are many different algorithms that can be built on top of a solver to drive its search for the optimum, while the decision process for feasibility is common among them all. However, a brief discussion on optimizing SMT solvers is provided at the end of the section.

In order to understand how SMT solvers work, it is necessary to talk briefly about Boolean satisfiability [38]. A SAT problem is formed by a set of clauses (formula) of Boolean variables in conjunctive normal form (see below). Solving a SAT problem means finding an assignment to each variable (either *true* or *false*) that makes the formula *true*, or providing a counterexample that proves that such assignment does not exist. Over the years SAT solvers have become very efficient and can nowadays solve large problems counting even millions of variables and clauses in relatively short time [39].

In the following, some terminology is given:

Literal: a literal is either a variable or its negation. We say that a literal is negative if it is a negated variable and positive otherwise. A positive literal is *satisfied* if its variable is assigned to *true*. Similarly, a negative literal is

satisfied if its variable is assigned to *false*.

Conjunctive Normal Form (CNF): a formula is in conjunctive normal form if it is a conjunction of disjunctions of literals, i.e. it has the form

$$\bigwedge_i \left(\bigvee_j l_{ij} \right),$$

where l_{ij} is the j-th literal of the i-th clause, where a *clause* is a disjunction of literals.

When modelling a real system, there is no guarantee that the best way to represent its behaviour is CNF; fortunately, there exist ways to transform any formula into a CNF formula, and it can be done quickly (Tseitin's linear encoding [40]). The reason why it is worth to spend computation time on transforming a formula into CNF is that since it is a conjunction of clauses, as soon as one clause is determined unsatisfiable, the whole formula is unsatisfiable. It is now briefly explained how a SAT solver can take advantage of having a formula in CNF to make an assignment:

state of a clause under an assignment: a clause is **satisfied** if one or more its its literals are satisfied (*true*), **conflicting** if all of its literals are assigned but the clause is not satisfied, **unit** if it is not satisfied and all but one of its literals are assigned, and **unresolved** otherwise.

If a clause is *unit*, this means that all but one of its literals are assigned but the clause is still not *satisfied* (i.e. so far all assigned literals are *false*); this means that the remaining literal has to be *true*, otherwise the clause would be *conflicting*. Therefore the remaining literal is **implied** by the clause.

The solver starts assigning literals based on some heuristic strategy, until one or more clauses become *unit* and *implies* one (or more) literals. This implication may turn more clauses into *units* (otherwise there is going to be more heuristic based assignments) and then the process goes on until all literals are assigned or a conflict arises because the implications made a clause *conflicting*.

SAT solvers exploit the above mentioned properties of formulas in CNF by applying the *conflict driven clause learning* (CDCL) framework to the problem. The search-space can be thought of as a binary tree, in which nodes

are partial assignments and leaves are full assignments. The solver traverses the tree and when a conflict is found, it "learns" from the clause by adding a new clause to the model containing the information about the assignment that led to the conflict, backtracks to the point where the assignment was made and tries a different assignment. The newly added clause will prevent making the same (conflicting) assignment again. The solver keeps traversing and backtracking until it finds a feasible assignment or no assignment exists that satisfies all the clauses, including the *learned* ones.

Example on the CDCL procedure

Let us clarify the concepts introduced so far with a small example; assume we want to find an assignment for the following formula, if such exist:

$$(\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_3 \vee x_1). \tag{3.1}$$

At the moment no literal is implied so the choice to satisfy one literal rather than another is merely dependent on the heuristic strategy. Remember that, since the formula is a conjunction of clauses, as soon as one of them is conflicting, the whole formula is not satisfied by the assignment.

Let us assign the value *false* to the variable x_1 in order to satisfy the literal $\neg x_1$ in the first clause; this implies the variable x_2 to be *true*, otherwise the second clause would be conflicting, since its first literal x_1 is not satisfied. Now x_3 is implied by the third clause to be *false*, since the first literal in it, $\neg x_2$ is not satisfied. Finally, the last implication makes the fourth clause conflicting, because it forces the variable x_1 to be *true*, since the first literal in it is not satisfied, but we previously assigned the value *false* to it.

So we need to backtrack to the point where we made a choice that eventually led to the conflict and make a different choice. In this case, that point was assigning the value *false* to x_1.

In order to avoid repeating the same mistake, we will add a clause to the formula: $\neg(\neg x_1)$, which is equivalent to say that x_1 must be *true*. In this case the new *learned* clause only contains one literal, which forces us to satisfy it; satisfying x_1 also satisfies clauses *two* and *four* and makes x_2 implied by clause *one*, which in turn makes $\neg x_3$ implied by clause *three*.

We have now a full assignment that satisfies (3.1).

From SAT to SMT

While SAT solvers can be extremely efficient in dealing with large models, SAT solvers are not expressive enough to easily model many real-world industrial problems. Boolean variables and propositional logic can be handy when it comes to model binary conditions (executing this operation or not), but integer and real variables and linear algebra are necessary to describe other important features (an operation can last at most this long). As it happens, SMT solvers are able to use integer and real variables over a range of different theories, including combinations of them (e.g. having logical conditions over linear inequalities). They can do so in two different ways: **eager approach** or **lazy approach** [41].

However, before going into details about eager and lazy approach, it is necessary to provide some more terminology:

- *logical symbols*: standard Boolean connectives (e.g. "$\wedge$","$\neg$"), quantifiers ("$\exists$" and "$\forall$"), and parenthesis;
- *non-logical symbols*: function, predicate, and constant symbols. a set of non-logical symbols is called a *signature* and is denoted by the symbol Σ;
- $\Sigma-$*formula*: formula that uses only the non-logical symbols from Σ (possibly in addition to logical symbols, this is why the distinction between logical and non-logical symbols is due);
- *free variable*: variable that is not bound by a quantifier;
- *sentence*: formula without free variables;
- *syntax*: rules for constructing formulas.

The distinction between *logical* and *non-logical* symbols is necessary because, while the clause of a theory may or may not be constructed using logical symbols, clauses are combined using logical symbols, regardless of the theory they belong to. For instance, a formula in linear algebra, is a conjunction of clauses, where each clause is a linear equality/inequality.

In a theory, the syntax is needed to *interpret* the non-logical symbols. For instance, the symbol "+" is usually associated to addition, but this may not be true for some theories. Hence, we need rules to use the non-logical symbols to construct the formulas.

theory T: a theory is a set of Σ-sentences. For a given Σ-theory, a Σ-formula φ is **T-satisfiable** if there exists an assignment that satisfies both the formula and the sentences of T.

In other words, we use a set of *sentences* to define the syntax of a theory, i.e. to define the interpretation of the non-logical symbols.

Eager Approach

One possibility is to convert the model into SAT; for each variable, depending on its domain, a number of Boolean variables are generated and for each constraint, a set of clauses. Then the problem is solved using the CDCL framework as described above.

This procedure is called *bit blasting* and can be computationally quite expensive. For each variable in the original problem there will be as many Boolean variables as the size of the original variable domain. When it comes to constraints declaration, the procedure can be even more expensive. Let us clarify this point with an example; the goal is to find an assignment to x_1 and x_2 such that x_1 is smaller than or equal to x_2, and their domains can be either 0, 1 or 2.

$$\begin{aligned} &x_1 \leq x_2 \\ &x_1, x_2 \in \{0, 1, 2\} \end{aligned}$$

To turn this problem into a SAT formulation, we define the following Boolean variables:

$$x_{ij} \quad i = \{0, 1\},\, j \in \{0, 1, 2\}$$

There is one Boolean variable for each variable in the original problem and for each element in the variable domain.

The linear inequality of can be converted as follows:

$$x_{ij} \implies \bigwedge_{\substack{j' \in \{0,1,2\} \\ j' \neq j}} \neg x_{ij'} \qquad \forall i = 0, 1, j \in \{0, 1, 2\} \tag{3.2}$$

$$x_{1j} \implies \bigwedge_{\substack{j' \in \{0,1,2\} \\ j' < j}} \neg x_{ij'} \qquad \forall j \in \{0, 1, 2\} \tag{3.3}$$

(3.2) states that if one of the Boolean variables representing each original variable is *true*, all the other Boolean variables representing the same original variable must be *false*. This is equivalent to say that only one of the Boolean variables representing each original variables can be *true* and this constraint is necessary because in the original problem, each variable can assume only one value from its domain; (3.3) states that if the Boolean variable representing x_1 having the value j is *true*, none of the variables representing x_2 having a value strictly smaller than j can be true.

While the eager approach can successfully be applied to finite domain theories, it is not clear how it would be possible to reduce linear real arithmetic literals to a Boolean satisfiability problem and since in this work we deal with discrete theories, the subject is not further investigated, though the reader is referred to [42] for further details.

Lazy Approach

The lazy approach combines the underlying SAT solver with a theorem prover to decide the assignment of variables. This means that the SMT solver must be able to recognize the theory it is dealing with and must be equipped with a prover appropriate for that theory.

A formula of a certain theory is a logical combination of clauses belonging to that theory. The solver will replace each clause in the formula with a Boolean variable and call the SAT solver to find a satisfiable assignment. If such assignment exists, the solver will call the theorem prover to check whether the current assignment is feasible within the theory. If not, a new clause is learned and the process starts again.

Once again, an example can aid understanding. Consider the formula:

$$(x_1 = x_3 \vee x_1 = x_2) \wedge (x_1 = x_2 \vee x_1 = x_4) \wedge x_1 = x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4 \quad (3.4)$$

The theory involved in this formula is called *equality theory* [43] where literals are either equalities or inequalities. Given a conjunction of literals (that would be the outcome of the SAT solver), there exist procedures to check whether the assignment is satisfiable or not.

In this case, the SMT solver would generate Boolean variables:

$$c_1 : x_1 = x_2$$
$$c_2 : x_1 = x_3$$
$$c_3 : x_1 = x_4$$

Negating that two variables are equal is equivalent to say that they are different, therefore $c_i \ \forall i = 1, \cdots, 3$ will evaluate to *true* if the corresponding equality literal is an equality, *false* otherwise.

It is now time to call the SAT solver to find an assignment for

$$(c_2 \vee c_1) \wedge (c_1 \vee c_3) \wedge c_1 \wedge \neg c_2 \wedge \neg c_3$$

Let us assume the solver finds the satisfiable assignment $c_1 \wedge \neg c_2 \wedge \neg c_3$; this means that the following is true: $x_1 = x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4$.

The solver will now call the theorem prover to check the assignment (which, in this case is satisfiable)

The example presented in this section are inspired by [43], which is the recommended text for further reading.

Optimization Modulo Theory

As mentioned in Chapter 1, it is practically always possible to express a problem in quantitative terms and define a cost function to optimize. SMT-solver developers soon realized this and started working on how to turn their solvers into SMT-based optimizing tools. In [44], the authors introduce an SMT variant where the theory handled by the solver becomes progressively stronger; this is done by implementing a branch-and-bound setting where the solver knows the cost function and the current best bound. Each time a better bound is found, the theory the solver is dealing with becomes stronger; therefore, models with a cost higher than this new bound become inconsistent with the theory. In [45] and [46], the authors describe some of the optimizing algorithms running under the hood of the SMT solver Z3, such as a collection of *MaxSAT* solvers (solvers that drive the search for an optimal solution by trying to satisfy as many *soft* constraints as possible) and a module for optimization of linear arithmetic objective functions; among others, the module contains is a Simplex-based algorithm for the CDCL framework [47]. In

[48], the authors provide an insight on how optimization is achieved with the SMT solver OptiMathSAT. They explain that unlike other OMT solvers, in which the SMT solver is used as a black-box and the optimization proceeds through a sequence of incremental SMT calls, with OptiMathSAT the whole optimization procedure is pushed inside the SMT solver.

In conclusion, optimization with SMT solvers can be achieved in different ways and it is hard to tell which one is the most efficient, since different approaches perform differently, depending on the problem.

3.3 SMT vs MILP

In the previous section we give a brief overview on what happens under the hood of a MILP solver and an SMT solver. There emerged, among other things, that SMT solvers allow for logical conditions over literals belonging to different theories; this feature can be extremely handy when modelling real system and saves time both in the design and the solving phase.

The following example is about non-overlapping of operations sharing the same resource in a job shop problem. One way to model the problem is by having an integer variable representing the start of each operation. Some operations are supposed to be executed by the same machine and therefore cannot overlap in time; either one has to be completed before the other one can start or the other way around.

Let s_a and s_b be integer variables representing the starting time of operations a and b, and let d_a and d_b be the duration of operations a and b respectively. For a MILP solver to handle the *non-overlap* constraint, it is necessary to declare an additional binary variable $z \in \{0, 1\}$ and find a large enough integer number M. If M is not large enough the solver may yield the wrong schedule; on the other hand, the larger it is, the longer the running time. The constraints to declare are then:

$$\begin{aligned} s_a \geq & s_b + d_b - M \cdot z, \\ s_b \geq & s_a + d_a - M \cdot (1 - z) \end{aligned}$$

If the variable z is equal to *one*, M being a large enough integer, s_a will be larger than $s_b + d_b - M$. This is equivalent to say that operation a starts after operation b. On the other hand, if z is equal to *zero*, s_b will be larger than

$s_a + d_a - M$.

On the other hand, with an SMT solver it is possible to combine literals of linear algebra, i.e. linear inequalities, with propositional logic:

$$s_a \geq s_b + d_b \vee s_b \geq s_a + d_a.$$

There can be much harder logical conditions to model and it becomes harder and harder to do it with a MILP solver since everything has to be in the form of a linear inequality. Also, the additional binary variables used to model logics increase the search-space size exponentially; in a model counting n binary variables, the branch and bound algorithm has to solve 2^n relaxed problems in the worst case, only to account for the binary variables (there may be other integer variables that increase the size even further). In paper A the disjunctive formulation is used to solve problem instances of the JSP using both a MILP and an SMT solver and the SMT solvers outperforms the MILP solver, sometimes by more than one order of magnitude.

After introducing MILP and SMT in this chapter, we are going to test them on different problems in the next chapter to compare their performance and evaluate how much the model formulation affects the solving process.

CHAPTER 4

Comparison of MILP vs SMT

As described in the previous chapter, both MILP and SMT solvers offer a general framework to model and solve integer linear problems. When it comes to modelling, SMT provides a more flexible language, since it is possible to combine different theories and declare logical conditions over their literals; nonetheless, modelling obstacles can be overcome in MILP using modelling tricks such as *big M* [49]. So, assuming that the constraints and the objetive functions are linear, both MILP and SMT allow to model the same type of problems.

When it comes to performance, though, depending on the particular problem one may outperform the other. It is therefore interesting to test them over well known benchmark problems, where plenty of literature is available for comparison. Note that benchmark problems are usually harder than randomly generated ones, since the benchmark have been selected specifically to stress-test a solution method.

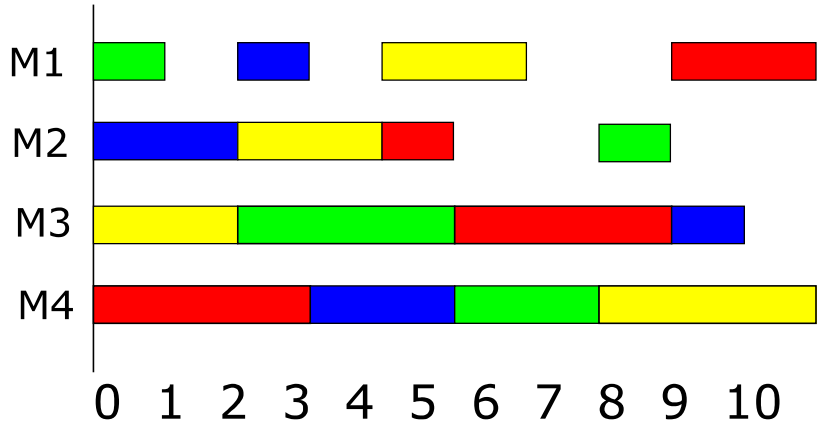

Figure 4.1: Illustration of the job shop problem.

4.1 The Job Shop Problem

An important classical optimization problem is the job shop problem (JSP) [50]. It is a well-known combinatorial optimization problem, that has been studied for decades and there exists a vast literature on exact and approximate methods to solve it, together with a large set of benchmark problems ranging from small to very large. Paper A and Paper B introduce the subject thoroughly and provide reference for further reading. Figure 4.1 shows a possible schedule for a problem concerning four jobs and four machines. Also, the JSP comes in different variants. The standard version has the goal of allocating single-capacity resources to jobs such that the overall execution time, the *makespan*, is minimized; each job has a predefined sequence of machines to visit (a job visiting a machine is called an *operation*) before it can be declared *finished*, and machines can only execute one job at a time. Finding an optimal schedule means to find the allocation order that minimizes the waiting time of jobs and, therefore, minimizes the make-span. The problem is notoriously hard to solve and, as of today, there exist problem instances that have not be solved to optimality [51].

Other common variants of the problem are: the flexible job shop problem (FJSP), discussed in Paper B, where operations can be executed by more than one machine; the *no-buffer* JSP [52] where jobs cannot leave a machine until

the next machine in the sequence is available for processing, since there is no buffer to hold the part. A sub-variant of this problem involves only limited-capacity buffers; there is the *no-wait* JSP, which involves constraints on the elapsing time between operations of the same job, due to the perishability of the goods, also described in [52].

The Standard JSP

As for all classes of problems, the model formulation plays an important role in the solving phase so it is important to know which are the main formulations existing to model the JSP and how they affect the solver's performance when. In [53], the authors present the *disjunctive*, *time-index*, and *rank-based* models as the most common formulations for the JSP (an in-depth description of these models is provided in Paper A) and present a computational analysis over a set of benchmark instances. Such formulations are tested using the state-of-the-art MILP solvers CPLEX [54] and Gurobi [55] and, for both solvers, the *disjunctive* model shows the best performance.

In Paper A, the above mentioned formulations are adapted for an SMT solver and then they are run over a set of benchmark problems using the state-of-the-art SMT solver Z3 [45]. The comparison shows that also for SMT solvers the *disjunctive* model outperforms the other two formulations, sometimes by an order of magnitude, allowing to solve much larger problem instances in reasonable time, as shown in the tables of Paper A. For a fair comparison, the *disjunctive* model is formulated for MILP as well and run over the same set of problems, using the same hardware as for the SMT solver. These tests show that the SMT solver outperforms the MILP solver on all problems and the gap increases as the problem size increases. In general, the combination SMT solver and *disjunctive* model is able to handle relatively large benchmark instances in a relatively short time (timeout set to 20 minutes). Further investigations [43] revealed that, when expressing the JSP using the disjunctive model, the problem's constraints fall into the domain of *difference logic*, a fragment of linear algebra; polynomial algorithms exist to check whether an assignment for this theory is satisfiable or not. Having the SAT engine efficiently finding feasible assignments and the theorem prover quickly checking them leads to an overall fast execution.

For the other problem formulations, *time-index* and *rank-based*, only a limited number of problems could be solved within reasonable time. Paper A

provides extensive data about the comparison. Even so, there are reasons to work on improving such formulations; the *time-index* formulation is widely used because its relaxations provide strong bounds useful in specific-purpose algorithms. In Paper C the *time-index* formulation is improved by exploiting the ability of using bit-vectors with SMT solvers. The *time-index* model is based on time discretization where one Boolean (or binary) variable is declared for each time step (and each operation of each job). This approach has the disadvantage of leading to a very large number of variables (and especially constraints), making the model size and the model generation time, not negligible (in some cases, they are actually the bottleneck). Using bit-vectors makes the model extremely more compact and brings the model size down to a small fraction of what it would be otherwise.

When it comes to performance (measuring only the solving time), the bit-vector-based *time-index* model outperforms its Boolean counterpart, but, unfortunately, it is still no match for the *disjunctive* model, as shown by comparing the data from Paper A and Paper C

The Flexible JSP

As mentioned above, there exist many different variants of the JSP that include additional constraints: setup times, non-infinite buffers, etc. One of them is the flexible JSP (or FJSP), where multi-purpose machines can perform different tasks; hence, for each operation, there is a *set of machines* to choose from. Also, processing times are dependent on the machine executing the operation.

Once again it is possible to formulate the problem in different ways. In Paper B extended versions of the *disjunctive*, *time-index*, and *rank-based* models are presented and compared against a set of benchmark problems for the FJSP (note that this is not the same set as for the standard JSP). The computational analysis presented in Paper B shows consistency with the previous study, i.e. the *disjunctive* model outperforms the *time-index* and *rank-based* formulations.

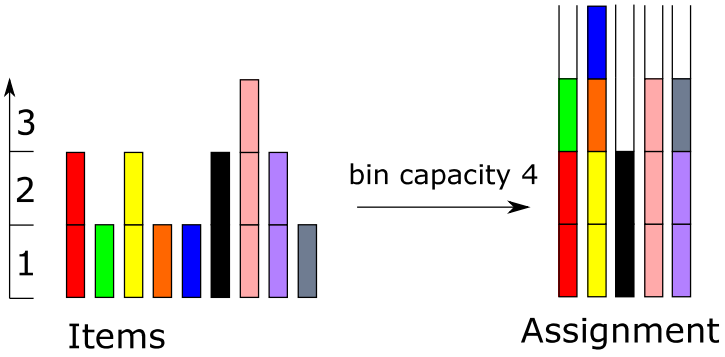

Figure 4.2: Illustration of the bin sorting problem.

4.2 The Bin Sorting

Another well-known combinatorial optimization problem is the bin sorting problem (BSP). This is the problem of fitting items into bins such that the number of bins is optimal. Items are characterized by a value indicating their size (in real world scenarios this could represent their weight or their volume) and there exist two versions of the BSP, one being the dual of the other. In the bin packing problem (BPP), there is a maximum capacity of the bins that cannot be exceeded and the goal is to minimize the number of bins; in the bin covering problem (BCP), there is a minimum capacity of the bins that cannot be under-reached and the goal is to maximize the number of bins. Figure 4.2 shows a BPP where nine items have to be packed in bins of maximum capacity *four*.

Especially for the BPP there is plenty of literature available for comparison, as well as five different sets of standard benchmark problems (references to the problems are given in Paper D). Paper D shows an enumerative model formulation to solve both BSPs based on the concept of *skinny/fit* bins. When enumerating possible combinations of items that could form a bin, it is possible to tell beforehand which combination that may never be included in the optimal solution (those would be the *non-skinny/non-fit*). This allows to cut off a (usually large) portion of the search-space and, subsequently, to reduce the running time to search it.

Paper D is an extension of our previous work on the subject presented in Paper E, where we implemented such an enumerative approach for the BCP

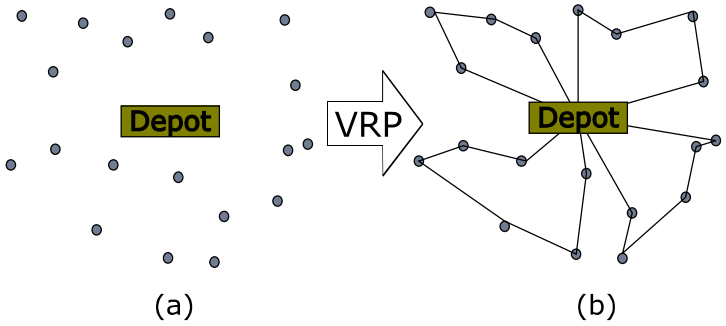

Figure 4.3: Illustration of the vehicle routing problem.

only, but for both MILP and SMT (in Paper D only MILP is used). This time the MILP solver proved to be much faster than the SMT solver (which is the reason why SMT is not used in Paper D).

4.3 The Vehicle Routing Problem

In order to get a better understanding of the VRP, we decided to work on one of its standard versions, namely the *vehicle routing problem with time windows* (VRPTW). This problem is a VRP where each customer must be served within a time window. Also, the problem belongs to the class of *capacitated* VRPs, where customers have requirements on the amount of goods to be delivered, namely *demands*, and vehicles have limits on the amount of goods they can deliver, namely *capacity*. Figure 4.3(a) shows a VRP where vehicles have to be dispatched from the *depot* to serve customers. In (b) the VRP is solved and the routes are showed.

In Paper E a MILP formulation and a set-based particle swarm optimization (S-PSO) algorithm for VRPTW are compared. Unfortunately we do not have data about the performance of any SMT solver for the benchmark problems, since the goal of the work presented in Paper E was the comparison of an exact and an approximate method and MILP had been chosen as exact method.

The work reveals that MILP does not scale very well, and is outperformed by the S-PSO algorithm for larger problem instances. This result does not come unexpected. Though there exist different formulations, and a most recent and efficient one was chosen for the comparison [19], they all have to represent the

choices of travelling from one point to another or not. This means that for a problem counting n customers there are n^2 binary variables so in the worst case the MILP solver will have to solve 2^{n^2} relaxed problems. Therefore, for a problem involving several logical constraints, MILP does not seem to be the best option.

Moreover, the specific industrial setup presented in Chapter 2 involves additional logical constraints that would make the number of binary variables grow even larger.

4.4 Conclusions on the SMT/MILP performance

So far we have introduced the performance evaluation of two solvers (one MILP solver and one SMT solver) over benchmark problems of two optimization problems: for the JSP the SMT solver outperformed the MILP solver while for the BSP it was the opposite.

The reason for this (or at least part of the reason), lies in the model structure. The BSP perfectly fits the standard requirements for MILP because it is a conjunction of linear inequalities. Therefore the MILP solver has no trouble dealing with very large models (in the evaluation presented in Paper D, we generated models counting up to twenty million variables). The SMT solver also comes with theories to handle linear algebra, but it is not as efficient as the MILP solver and, therefore, slower. One reason for this is that companies such as Gurobi [55] and CPLEX [54] have been spending decades refining the algorithms that run under the hood of a MILP solver.

On the other hand, the JSP has a disjunctive constraint (non-overlap of operations requiring the same job) that has to be modeled using additional binary variables together with the *big M* method when using the MILP solver. The SMT solver, instead, will set up a SAT problem where each linear inequality is treated as a Boolean variable and find an assignment (if such exists) for them; then the theory prover will check the feasibility of the assignment of the Boolean variables for the corresponding inequalities.

Based on the computational anlysis run for the JSP and the BSP and based on what we know about the algorithms running under the hood of the MILP and SMT solvers, we can draw the following conclusion: if the problem naturally involves only conjunctions of literals belonging to linear algebra, the MILP solver will likely be the most suitable option; on the other hand, if the

problem involves more logical constraints, such as disjunctions, implications, etc., the SMT solver will likely outperform the MILP solver, being naturally designed to handle constraints of propositional logic over literals of different theories.

In Paper F is presented a computational analysis over a benchmark set of problem instances of the VRPTW using a MILP solver showing that the solver is not able to handle large size instances. Though we do not know how an SMT solver would compare on the same benchmark set, given the structure of the VRP involving logical constraints, we would expect the SMT solver to be a better candidate for the task. We hence decided to use an SMT solver to deal with the VRP presented in Chapter 2.

In the following chapter, the specifications for the VRP presented in Chapter 2 are given. A monolithic model and a compositional algorithm to solve the VRP are presented. Finally, four test-cases are used to provide data about the performance of the two approaches.

CHAPTER 5

Monolithic Model and Compositional Algorithm for the VRP

In the following sections we present a monolithic model that takes care of charging stations and conflict-free routing, as well as the other specific constraints; we also present a compositional algorithm that is able to handle larger problems in a reasonable time. For both the monolithic model and the compositional algorithm, the total travelled distance is used to evaluate the quality of the solutions.

The monolithic model is developed using a time-index formulation where a set of Boolean variables is used to model locations of the vehicles and movements at each time-step; other sets of Boolean variables keep track of the assignment of vehicles to jobs; integer variables are also used to keep track of the vehicle remaining charge.

On the other hand, the compositional algorithm breaks the overall VRP problem down into three main sub-problems:

- The first sub-problem is the *routing problem*; using the SMT solver to find non-cyclic paths between any two customers, and between the depot and each customer (this phase is discussed in Section 5.3), we can

now treat the problem as a VRPTW. We can do this because knowing the paths between any two points of interest allows us to calculate the distance between them; in the VRPTW the actual trajectory to travel from a point to another does not matter, only the distance does. In this phase we also constraint the routes to be shorter than the vehicles operating range because we do not handle recharging during the routing.

- The second sub-problem is the *assignment problem*; in this phase the routes selected by the solver in the previous phase have to be assigned to the vehicles. The assignment is done according to the job's requirements on the vehicles and to the job's time windows. Recharging of vehicles is handled in this phase; if a vehicle is assigned to more than one route, every time it goes back to the depot after executing a job, it has to wait until its battery has enough charge to complete the next job and go back to the depot.

- The third sub-problem is the *scheduling problem*; in this phase the assignment of the routes to the vehicles is checked against the capacity constraint. In other terms, we know now which road segments each vehicle has to traverse in order to execute the jobs; we know what is the latest time at which the vehicles can visit the customers and still be able to meet the time windows; we want to know if there is a schedule compatible with the capacity constraints that allows the vehicles to meet their deadlines. Computing a schedule means to decide where each vehicle must be at each time, and whether it is moving or it is stationary.

While the *routing problem* is handled as a VRPTW, the *assignment* and the *scheduling* problems can be formulated as variations of the JSP. Hence, they can be modelled using the *disjunctive* formulation. Figure 5.1 shows a flowchart that qualitatively represents how the compositional algorithm is constructed. All the functions that the algorithm calls are described in details in Section 5.3. The variables *PF*, *RF*, *AF*, and *SF* represent the status of the *Path Finder*, *Routing*, *Assignment*, and *Scheduling* problems respectively, i.e. they say whether the problem is satisfiable (*sat*) or unsatisfiable (*unsat*).

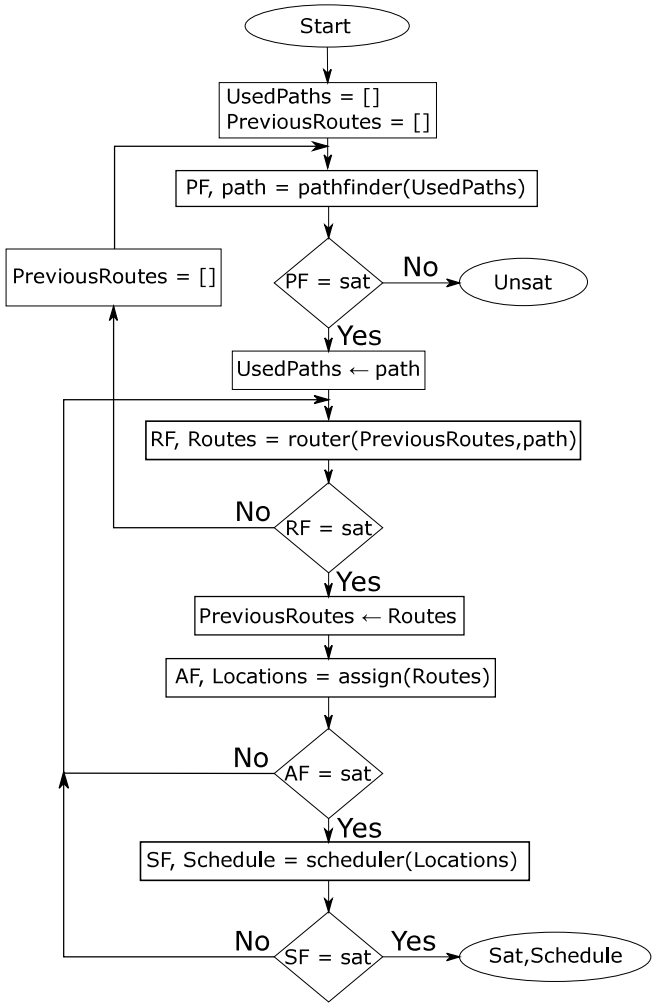

Figure 5.1: Flowchart of the Compositional algorithm.

5.1 The data structure

As discussed in Paper F and Chapter 2, the VRP problem can be modelled using a graph, where nodes represent points of interest, such as depot stations, charging stations, and customers (pickup and delivery points). When dealing with the standard VRP, without considering conflict-free routing, a complete undirected graph can be used, where nodes represent the depot and the customers, and there exist edges connecting any two nodes and where the edge weights represent the distance between such nodes. On the other hand, when dealing with conflict-free routing, the graph represents the actual layout of the plant, where edges are road segments and nodes are intersections. Therefore, not all pairs of nodes are connected and the graph is directed, in order to model sections where the flow is allowed only in one direction. Customers and depot are located in some specific nodes.

It is assumed that nodes can accommodate an unlimited number of vehicles at the same time. Edges can as well accommodate multiple vehicles at the same time, as long as they do not occupy the same location (this constraint is further addressed in the subsections about conflict-free routing of both the monolithic model and the compositional algorithm).

Another important piece of information regarding a problem instance is the list of jobs; for each of them, the list of tasks (pickup or delivery) is given, together with a list of vehicle types eligible to execute the job; moreover, for each task is provided a location (the node the vehicle has to reach in order to pick up or deliver goods), a precedence list enumerating the tasks that have to be executed before the task the list refers to (typically, only delivery points have a non-empty precedence list which corresponds to the pickup points belonging to the same job) and a time window (again, this is typically true only for the delivery points). A global list of vehicle types is also given and for each type it is specified how many vehicles are available. Finally, other parameters are the operating range of the vehicles OR, the charging coefficient C, the discharge coefficient D, and a large number B representing the time when all jobs should be finished. Finding good bounds for B leads to smaller models though being too strict can lead to mistakenly declare a problem unfeasible. A trivial upper bound for the value B can be calculated by summing the time it takes to execute all jobs in sequence.

5.2 The Monolithic Model

In order to model the VRP described above, we define the following notation that will be used to define constraints:

N: the set of nodes

$O \in N$: the origin node

$E \subseteq N \times N$: the set of edges

$A_n \subset N \; \forall n \in N$: the set of nodes that have an incoming edge from node n

h: the value of the heaviest edge (longest transition time)

B: an upper bound representing the time when all jobs should be finished

$T = B + h$: the time horizon [1]

J : the set of jobs

$K \; \forall j \in J$: the tasks of job j

$L_{jk} \in N \; \forall j \in J, k \in K$: the location of task k of job j

V: the set of all vehicles

OR: the maximum operating range of the vehicles

C: the charging coefficient

D: the discharging coefficient

$V_j \subseteq V \; \forall j \in J$: the set vehicles eligible for job j

F: the set of charging stations

$P_{jk} \subset K_j \; \forall j \in J, k \in K$: the set of tasks to execute before task k of job j

$l_{jk} \; \forall j \in J, k \in K$: the time window's lower bound for task k of job j

$u_{jk} \; \forall j \in J, k \in K$: the time window's upper bound for task k of job j

$d_{nn'} \; \forall n, n' \in N$: the distance between nodes n and n' [2]

[1] The need to increase the value B is discussed on page 45.

[2] it is assumed that one unit of distance is covered in one time step, hence distance and duration are interchangeable

The set of variables used to build the model are:

x_{ij}: Boolean variables that evaluate to true if the i-th vehicle is assigned to the j-th job

y_{ijk}: Boolean variables that evaluate to true if the i-th vehicle is assigned to the k-th task of the j-th job

z_{ijkt}: Boolean variables that evaluate to true if the i-th vehicle serves the k-th task of the j-th job at time t

rc_{it}: Integer positive variables that keep track of the remaining operating range of vehicle i at time t

at_{int}: Boolean variables that evaluate to true if the i-th vehicle is at the n-th node at time t

$move_{int}$: Boolean variables that evaluate to true if the i-th vehicle decides to move to the n-th node at time t

In order to build the models, the following logical operators will be used:

- AMO(a) : at most one variable of the set a evaluates to true;
- EO(a) : exactly one variable of the set a evaluates to true;
- EN(a, n) : exactly n variables of the set a evaluate to true;
- If(c, o_1, o_2) : if condition c is *true* returns o_1, else returns o_2.

For further clarity, the constraint sets have been divided into sections, depending on the features of the system they take care of.

Jobs Assignment

The following constraints are about assignment of jobs to vehicles:

$$z_{ijkt} \implies at_{iL_{jk}t} \qquad \forall i \in V, j \in J, k \in K_j, t = 0, \ldots, T \tag{5.1}$$

$$x_{ij} \implies \bigwedge_{\substack{i' \in V \\ i \neq i'}} \neg x_{i'j} \qquad \forall i \in V, j \in J \tag{5.2}$$

$$y_{ijk} \implies \bigwedge_{\substack{i' \in V \\ i \neq i'}} \neg y_{i'jk} \qquad \forall i \in V, j \in J, k \in K_j \tag{5.3}$$

$$y_{ijk} \implies \bigvee_{t=0}^{T} z_{ijkt} \qquad \forall i \in V, j \in J, k \in K_j \tag{5.4}$$

$$z_{ijk_1t} \implies \bigwedge_{\substack{k_2 \in P_{jk_1} \\ t' = t, \ldots, T}} \neg z_{ijk_2t'} \qquad \forall i \in V, j \in J, k \in K_j, t = 0, \ldots, T \tag{5.5}$$

$$x_{ij} \implies \bigwedge_{k \in K_j} y_{ijk} \qquad \forall i \in V, j \in J \tag{5.6}$$

$$y_{ijk} \implies \bigvee_{t = l_{jk}, \ldots, u_{jk}} z_{ijkt} \qquad \forall i \in V, j \in J, k \in K_j \tag{5.7}$$

$$\neg x_{ij} \qquad \forall i \notin V_j, j \in J \tag{5.8}$$

(5.1) states that in order to serve a task, a vehicle has to be at the task location; (5.2) states that only one vehicle can be assigned to a job; similarly, (5.3) states that only one vehicle can be assigned to a task; (5.4) states that, in order to execute a task, a vehicle must be at the task location at some point in time; (5.5) enforces the precedence constraint among tasks of the same job; (5.6) states that, when assigned to a job, a vehicle must execute all its tasks; (5.7) states that a vehicle must execute tasks within their time windows (when the task has got one); (5.8) states that only eligible vehicles can execute the corresponding job.

Vehicles' Movements

The following constraints are related to the vehicles' movements:

$$at_{iO0} \qquad \forall i \in V \tag{5.9}$$

$$at_{iOB} \qquad \forall i \in V \tag{5.10}$$

$$\text{AMO}_{n\in N}(at_{int}) \qquad \forall i \in V, t = 0, \ldots, T \tag{5.11}$$

$$\text{AMO}_{n\in N}(move_{int}) \qquad \forall i \in V, t = 0, \ldots, T \tag{5.12}$$

$$at_{int} \implies \neg move_{int} \qquad \forall i \in V, n \in N, t = 0, \ldots, T \tag{5.13}$$

$$at_{int} \implies \bigwedge_{n' \notin A_n} \neg move_{in't} \qquad \forall i \in V, n \in N, t = 0, \ldots, T \tag{5.14}$$

$$(at_{int} \wedge \bigwedge_{n' \in N} \neg move_{in't}) \implies at_{int+1}$$

$$\forall i \in V, n \in N, t = 0, \ldots, T-1 \tag{5.15}$$

$$(at_{int} \wedge move_{in't}) \implies (\bigwedge_{\substack{n'' \in N \\ t'=t+1,\ldots,t+d_{nn'}-1}} \neg at_{in''t'} \wedge at_{in't+d_{nn'}})$$

$$\forall i \in V, (n, n') \in E, t = 0, \ldots, T - d_{nn'} \tag{5.16}$$

$$(z_{ij_1k_1t_1} \wedge z_{ij_1k_2t_2}) \implies \bigwedge_{\substack{j_2 \in J, j_1 \neq j_2 \\ k' \in K_{j_2} \\ t'=t_1+1,\ldots,t_2}} \neg z_{ij_2k't'}$$

$$\forall i \in V, j_2 \in J, k_1, k_2 \in K_{j_1}, k_1 \neq k_2, t_1 = 0, \ldots, T, t_2 = t_1, \ldots, T \tag{5.17}$$

(5.9) states that all vehicles are at the depot at time zero; (5.10) states that all vehicles are at the depot at time B. this constraints forces the vehicles to return to the depot after they execute the jobs they are assigned to; (5.11) states that a vehicle can be at most in one location at a time. Normally, it would make sense to state that a vehicle should be *exactly* in one place at a time. However, given the way the vehicle movement is modeled, this constraint needs to be relaxed because moving can take more than one time-step; (5.12) states that a vehicle can move to at most one location at a time;(5.13) forbids a vehicle to move the node where it already is, and (5.14) states that if vehicle i at time-step t is at node n, it cannot move to any non-adjacent node; (5.15) states that, if a vehicle is not moving at a certain time step, it will be at the same location at the next time step; (5.16) states that, if a vehicle is to move

to a new location, it will occupy no place while it is transiting on the edge connecting the start and arrival node and it will be at the arrival node after as many steps as it takes to traverse the edge, defined by the edge's weight. It is for this reason that constraint (5.11) needs to be relaxed. Also, in (5.16) $t' = t+1, \ldots, t+d_{nn'}-1$, therefore $t = 0, \ldots, T-d_{nn'}$. This means that there are no constraints applying to events taking place in the interval $[T-d_{nn'}, T]$. For this reason it is necessary to extend the time horizon to $T = B + h$: the steps $B, \ldots, h$ are not actually used in the model but this way it is guaranteed that all constraints apply for $T = 0, \ldots, B$; (5.17) states that if a vehicle is executing two tasks k_1 and k_2 of job j, then the vehicle cannot execute a task of any other job different from j in between. This constraints is used to make sure that, if a vehicle is selected for multiple jobs, it has to execute them in sequence (i.e. finish all tasks of a job before executing any other task).

Conflict-Free Routing

The following constraints are related to the conflict-free routing:

$$(at_{i_1nt} \wedge at_{i_2nt}) \implies \bigwedge_{n' \in A_n} (\neg move_{i_1n't} \vee \neg move_{i_1n't})$$
$$\forall i_1, i_2 \in V, i_1 \neq i_2, n \in N, t = 0, \ldots, T \quad (5.18)$$

$$(at_{i_1,n,t} \wedge at_{i_2,n',t} \wedge move_{i_1,n't}) \implies \bigwedge_{t'=t,\ldots,t+d_{nn'}} \neg move_{i_2nt'}$$
$$\forall i_1, i_2 \in V, i_1 \neq i_2, (n, n') \in E, t = 0, \ldots, T - d_{nn'} \quad (5.19)$$

(5.18) states that if two vehicles are on the same node, they cannot transit on the same edge at the same time step; (5.19) states that if two vehicles are on two adjacent nodes and one decides to traverse the edge connecting them, the other vehicle has to wait for the first one to finish the transit before it can traverse the edge.

Battery Management

The following constraints are related to the battery management:

$$rc_{it} \geq 0 \wedge rc_{it} \leq OR \qquad \forall i \in V, t = 0, \ldots, T \tag{5.20}$$

$$(at_{int} \wedge move_{in't}) \implies \bigwedge_{t'=t+1,\ldots,t+d_{nn'}+1} rc_{it'} = rc_{it'-1} - D$$

$$\forall i \in V, (n, n') \in E, t = 0, \ldots, T - d_{nn'} \tag{5.21}$$

$$(at_{int} \wedge \bigwedge_{n' \in N} \neg move_{in't}) \implies rc_{it+1} = rc_{it}$$

$$\forall i \in V, n \in N \setminus F, t = 0, \ldots, T-1 \tag{5.22}$$

$$(at_{int} \wedge \bigwedge_{n' \in N} \neg move_{in't}) \implies rc_{it+1} = rc_{it} + C$$

$$\forall i \in V, n \in F, t = 0, \ldots, T-1 \tag{5.23}$$

(5.20) defines the domain of the rc variables; (5.21) states that, if a vehicle is moving, its energy decreases at each time-step according to the discharge coefficient; (5.22) states that if a vehicle is not moving, its energy level does not change. It is necessary to specify that the vehicle is at some node because if the vehicle is at no node, it means it is moving; (5.23) states that if a vehicle is at a charging station, its energy level increases at each step according to the charging coefficient.

Objective Function

Finally, the cost function for the sub-problem to minimize (5.24) is the total travelled distance, calculated as the cumulative distance travelled by each and every vehicle:

$$\sum \text{If}((at_{int} \wedge move_{in't}), d_{nn'}, 0) \qquad \forall i \in V, n \in N, t = 0, \ldots, T-1 \tag{5.24}$$

5.3 The Compositional Algorithm

In this section the compositional algorithm is presented: the overall VRP problem presented in Chapter 2 is broken down into sub-problems that represent different stages in the algorithm. In the following, a subsection is dedicated to each sub-problem, to describe its input and output and how it connects to the previous and/or following sub-problem.

For the algorithm to work, two additional jobs, having both only one task, are added: *start* and *end*; they are needed in the routing problem, to make sure that routes begin and end at the depot. They are both located at the origin node O and while the single task of job *start* has no time window, the time window of the single task of job *end* is $[0, B]$.

The Path Finder

The *Path Finder* model selects a sequence of edges connecting any two points of interest that minimizes the total sum of distances of each path. Let P be the set of all pairs of points of interest. For any pair of points $p_i \in P$, all possible non-cyclic paths are computed and then stored in a list f_i. The paths are then stored in the list $Paths : p_i \mapsto f_i$.

The variables $path_{qr}$ are Booleans that evaluate to *True* if the r-th path of the q-th pair of points is selected. The following constraint ensures that only one path per each pair can be selected:

$$\mathrm{EO}_{r \in f_q}(path_{qr}) \qquad \forall q \in P \tag{5.25}$$

Since the solution found may not be feasible for the following steps of the algorithm, it is necessary to store it so that it can be ruled out in the next iteration. Let $S_{path} = \bigcup_{\substack{q \in P \\ r \in f_q}} \{path^*_{qr}\}$ be one feasible solution to the *Path Finder* model; also, let *UsedPaths* be a list containing all the previous solutions to the *Path Finder* model. In order to find another feasible solution, the following constraint must be added to the model:

$$\bigvee_{path_{qr} \in S_{path}} \neg path_{qr} \qquad \forall S_{path} \in UsedPaths \tag{5.26}$$

Finally, the cost function to minimize is the overall length of the paths (in

terms of nodes to visit); let $|r|$ be the length of a path:

$$\sum \text{If}(path_{qr}, |r|, 0) \qquad \forall q \in P, r \in f_q \tag{5.27}$$

Based on the model described above, it is possible to define the function *path_selection* that takes the list *Paths* of all non-cyclic paths between any two points and the list *UsedPaths* that contains the previously used paths as inputs, and returns *current_path*, the shortest feasible combination of non-cyclic paths that has not been selected yet, else it returns *unsat*.

The Routing Problem

The goal is now to find feasible routes using the non-cyclic paths currently provided by the *path_selection* function to calculate the distance $d_{j_1 k_1 j_2 k_2}$ between points of interest. Also, let M_j be the set of mutually exclusive jobs for job j, and let $Perm_j$ be the set of possible orderings of tasks belonging to job j, where each possible ordering $ord_m \in Perm_j$ contains all tasks of job j sorted differently. The set of variables used to build the model for the routing problem are:

$dir_{j_1 k_1 j_2 k_2}$: Boolean variable that evaluates to true if a vehicle travels from the location of task k_1 of job j_1 to the location of task k_2 of job j_2

cs_{jk}: integer variable that keeps track of the arrival time of a vehicle at the location of task k of job j

rc_{jk}: integer variable that keeps track of the remaining charge of a vehicle when at the location of task k of job j

The model is as follows:

$$(cs_{jk} \geq 0 \wedge rc_{jk} \geq 0 \wedge rc_{jk} \leq OR) \qquad \forall j \in J, k \in K_j \tag{5.28}$$

$$\neg dir_{jkjk} \qquad \forall j_1 \in J, k_1 \in K_{j_1} \tag{5.29}$$

$$\neg dir_{j,k,start,k_{start}} \qquad \forall j \in J, k \in K_j \tag{5.30}$$

$$\neg dir_{end,k_{end},j,k} \qquad \forall j \in J, k \in K_j \tag{5.31}$$

$$dir_{j_1k_1j_2k_2} \implies cs_{j_2k_2} \geq cs_{j_1k_1} + d_{j_1k_1j_2k_2} \qquad \forall j_1, j_2 \in J, k_1 \in K_{j_1}, k_2 \in k_{j_2} \tag{5.32}$$

$$(cs_{jk} \geq l_{jk} \wedge cs_{jk} \leq u_{jk}) \qquad \forall j \in J, k \in K_j \tag{5.33}$$

$$dir_{j_1k_1j_2k_2} \implies rc_{j_2t_2} \leq rc_{j_1t_1} - D \times d_{j_1k_1j_2k_2} \qquad \forall j_1, j_2 \in J, k_1 \in K_{j_1}, k_2 \in k_{j_2} \tag{5.34}$$

$$\text{EO}_{\substack{j_2 \in J \\ k_2 \in K_{j_2}}} (dir_{j_1k_1j_2k_2}) \qquad \forall j_1 \in J, j_1 \neq j_2, k_1 \in K_{j_1} \tag{5.35}$$

$$EN_{\substack{j_2 \in J \\ k_2 \in K_{j_2}}} (dir_{j_1k_1j_2k_2}, n) \implies EN_{\substack{j_2 \in J \\ k_2 \in K_{j_2}}} (dir_{j_2k_2j_1k_1}, n) \qquad \forall j_1 \in J, k_1 \in K_{j_1}, n = 1, \ldots, |J| \tag{5.36}$$

$$\bigvee_{ord_i \in Perm_j} \left(\bigwedge_{\substack{j_1, j_2 \in ord_i \\ j_1 \geq j_2}} dir_{jk_1j_2k} \right) \qquad \forall j \in J \tag{5.37}$$

$$\bigwedge_{P_{jk}} cs_{jk} \geq cs_{jk'} \qquad \forall j \in J \tag{5.38}$$

(5.28) restricts the variables to be positive integers and the remaining charge to be lower than the maximum operating range; (5.29) forbids to travel from and to the same location; (5.30) and (5.31) state that a vehicle can never travel to the start, nor thravel from the end: *start* and *end* are physically located at the same node, but they play different roles in the routing problem, hence two different jobs; (5.32) regulates the difference in the arrival time based on the distance for a direct travel between two points; (5.33) enforces the time windows on the routes; (5.34) defines the decrease of charge based on the distance travelled; (5.35) states that each customer must be visited exactly once; (5.36) guarantees the flow conservation between start and end; (5.37) states that if a number of tasks belongs to one job, they have to take place in sequence; (5.38) guarantees that deliveries take place after pickups.

Objective Function

Finally, the cost function for the model to minimize (5.39) is the total number of vehicles:

$$\sum \text{If}(dir_{start,k_{start},j,k}, 1, 0) \qquad \forall j \in J, k \in K_j \tag{5.39}$$

If the solution of the routing problem turns out to be inconsistent with the vehicles' assignment or the conflict-free constraints in the next two phases, a new solution must be computed in order to find alternative routes for the same combination of non-cyclic paths. Therefore it is necessary to keep track of the combinations of routes that have already been generated so that we can rule them out when solving the routing problem again. Let $R = \bigcup_{\substack{j \in J \\ k \in K_j}} \{cs^*_{jk}\}$ be the optimal solution to the routing problem found at iteration n and *PreviousRoutes* the set containing the optimal solutions found until the $n-1$-th iteration. In order to find the optimal set of routes, different from the ones found before, the following constraint must be added:

$$\bigvee_{dir_{j_1k_1j_2k_2} \in Routes} \neg dir_{j_1k_1j_2k_2} \qquad \forall Routes \in PreviousRoutes \tag{5.40}$$

Based on the model described above, it is possible to define the function *router* that takes the current combination of non-cyclic paths *current_path* and the set *PreviousRoutes*, and returns a set of routes that have not been selected yet R, if such exists, *unsat* otherwise. The set of routes R contains the variables $dir_{j_1k_1j_2k_2}$ that evaluated to *True* in the solution of the routing problem; through them, it is possible to construct the routes.

The Assignment Problem

The *assignment problem* allocates vehicles to the routes R generated in the routing problem, based on the time window and eligibility requirements. In the previous phase routes were generated based only on the time windows and on the vehicles' operating range; now the actual availability of each type of vehicle is given. Moreover, the *router* may generate routes that involve mutually exclusive jobs and, while it would be possible to avoid this by adding additional constraints, it would be inconvenient to do in the *routing problem*, since there is no information about the vehicles assigned to the routes. On

the other hand, once a set of routes is given, it can be easily checked in the *assignment problem* whether a vehicle is actually eligible for a route.

Therefore, for each route r, we can define a list of jobs $J^r \subseteq J$ that are executed by the vehicle assigned to r; then the list of eligible vehicles $El_r = \bigcap_{j \in J^r} V_j$ is given. Also, based on the time windows of the jobs forming the routes, it is possible to work out the latest start of a route $late_r$; for instance, if the time window's upper bound for the delivery task of a job is at time t and the distance between the task's location and the depot is d, then the latest start is $t - d$ (remember that we assume time and distance travelled to be interchangeable). Since a route can include more than one job, the stricter deadline will define the latest start for the route.

The *assignment problem* is therefore treated as a *job shop problem* where routes are jobs (whose duration depends on their length $length_r$) and vehicles are resources, with some additional requirements on the jobs staring time. The set of variables used to build the model are:

$allo_{ir}$: Boolean variables that evaluate to *True* if vehicle i is assigned to route r;

$start_r$: Non-negative integer variables that keep track of the start time of route r;

end_r: Non-negative integer variables that keep track of the end time of route r.

The model formulation for the assignment problem is as follows:

$$end_r = start_r + length_r \qquad \forall r \in R; \tag{5.41}$$

$$start_r \leq late_r \qquad \forall r \in R; \tag{5.42}$$

$$\mathrm{EO}_{i \in V}(allo_{ir}) \qquad \forall r \in R; \tag{5.43}$$

$$\bigvee_{i \in El_r} allo_{ir} \qquad \forall r \in R; \tag{5.44}$$

$$(allo_{ir} \wedge allo_{ir'}) \implies$$
$$((start_r \geq end_{r'} + C \cdot length_r) \vee (start_{r'} \geq end_r + C \cdot length_{r'}))$$
$$\forall i \in V, r, r' \in R, r \neq r' \tag{5.45}$$

(5.41) connects the *start* and *end* variables based on the route's length; (5.42) sets the latest start time of a route based on the stricter time window among

the ones of its jobs; (5.43) states that exactly one vehicle must be assigned to a route; (5.44) states that one (or more) among the eligible vehicles must be assigned to a route; (5.45) states that any two routes assigned to the same vehicle cannot overlap in time; either one ends before the other starts or the other way around.

Based on the (5.41)-(5.45) it is possible to define the function *assign* that takes the routes R from the routing problem as input and returns a feasible *assignment* if such exists, *unsat* otherwise. The assignment states which vehicle will drive on which route (and, of course, execute its jobs) and when it starts.

The Scheduling Problem

Finally, in this phase a feasible schedule is found for the vehicles, meaning that the routes they are assigned to are checked to verify that they are conflict-free. In order to do this, generate a list of nodes that each route visits $AN_r \ \forall r \in R$, and one list of edges $AE_r \ \forall r \in R$, since so far the focus was only on the points of interest. Note that for the same route r, AE_r will always be one element shorter than AN_r since there is an edge in between two nodes: this way the first edge in AE_r is always the edge following the first node in AN_r, the second edge in AE_r is always the edge following the second node in AN_r and so on. Also, for each node in AN_r it is necessary to specify whether there exist a time window, since some of the nodes are only spots on the map while others are actual pickup or delivery points). Let l_{rn} and u_{rn} be the earliest and latest arrival time at node n on route r respectively. Finally, let $len(e)$ be the length to travel of edge e and let $e(1)$ and $e(2)$ be the source and sink node of the edge respectively (since it is a directed graph, the edge connecting n and n' is different from the one connecting n' and n).

This phase is also treated as a *job shop problem*, where jobs are routes, tasks within the jobs are the different nodes to visit along the route while nodes and edges themselves are the resources. While nodes have no capacity limit, meaning that they can accommodate an unlimited number of vehicles at the same time (they are considered to be hubs), edges do. Also, each route has a starting time $start_r$ defined by the *assignment model*. The variables in the *scheduling problem* model are:

$node_{rn}$: Non-negative integer variables to keep track of when route r is

using node n;

$edge_{re}$: Non-negative integer variables to keep track of when route r is using edge e;

The model for the *scheduling problem* is defined as follows:

$$node_{rO} \geq start_r \qquad \forall r \in R \tag{5.46}$$

$$edge_{ri} \geq node_{ri} \qquad \forall r \in R, i = 1, \ldots, |AE_r| \tag{5.47}$$

$$node_{ri+1} \geq edge_{ri} + len(e) \qquad \forall r \in R, i = 1, \ldots, |AE_r| \tag{5.48}$$

$$(node_{ri} \geq l_{ri} \wedge node_{ri} \leq u_{ri}) \qquad \forall r \in R, i \in AE_r \tag{5.49}$$

$$(edge_{r_1 i} \geq edge_{r_2 i} + 1 \vee edge_{r_2 i} \geq edge_{r_1 i} + 1) \quad \forall r_1, r_2 \in R, r_1 \neq r_2, i \in AE_{r_1} \cap AE_{r_2} \tag{5.50}$$

$$(edge_{r_1 i_1} \geq edge_{r_2 i_2} + len(e_2) \vee edge_{r_2 i_2} \geq edge_{r_1 i_1} + len(e_1)) \quad \forall r_1, r_2 \in R, i_1 \in AE_{r_1}, i_2 \in AE_{r_2}, r_1 \neq r_2, e_1(1) = e_2(2), e_1(2) = e_2(1) \tag{5.51}$$

(5.46) constraints the beginning time of a route; (5.47) and (5.48) define the precedence among nodes and edges to visit in a route; (5.49) enforces time windows on the nodes that correspond to pickup or delivery points; (5.50) and (5.51) constraints the transit of vehicles over the same edge. If two vehicles are crossing the same edge from the same node, one has to start at least one time step later than the other and if two vehicles are traversing the same edge from opposite nodes, one has to be done transiting, before the next one can start.

Based on the (5.46)-(5.51) it is possible to define the function *scheduler* that takes the *assignment* from the assignment problem as input and returns a feasible *schedule* if such exists, *unsat* otherwise. The schedule expresses where each vehicle is at each time step.

The Algorithm

In this section, the compositional algorithm to solve the vehicle routing problem with time windows, charging station and pickup-delivery points is presented. The input for the algorithm is a *problem instance*, consisting of a weighted, directed *graph* representing the plant layout and a list of *jobs*, as described in Section 5.1. The function *path_enumerator* (Line 1 of the algo-

rithm) takes the graph and the jobs as input and, using a breadth first search algorithm [56] returns the previously mentioned list *Paths*.

The output of the compositional algorithm is twofold: the variable *solution*, which is initialized as *unknown* and will eventually become either *sat* or *unsat*, and the *schedule*, which will contains the information about the location of each vehicle at each time step if the problem is *sat* or be empty otherwise.

First, the algorithm finds a feasible combination of non-cyclic paths to connect any two tasks and between depot and all tasks: this is done through the function *path_selection*; if no combination of paths can be found (either because there are none or because all feasible solutions have been used already), the algorithm terminates and the *solution* is *unsat*. If a path list can be found, then such solution is added to the *UsedPaths* and it is used as an input to generate feasible routes, if such exist (line 5); if no solution exists to the routing problem, there are two possible outcomes depending on the list *PreviousRoutes*:

- *PreviousRoutes* is empty: the algorithm terminates and returns *unsat* (lines 7 and 8);
- *PreviousRoutes* is non-empty: a new combination of non-cyclic paths is computed (go back to line 2).

The *if* condition on the emptiness of *PreviousRoutes* can save time based on one assumption: every time a new combination of non-cyclic paths is computed, it is the shortest still available. If no routing is possible, i.e. time windows could not be met with the current paths, there is no other combination of paths that will satisfy the routing problem, since they will be longer than the current one. On the other hand, if *PreviousRoutes* is not empty, this means that routing is possible with the current combination of non-cyclic paths and it would be premature to declare the instance *unsat*. Instead, the list *PreviousRoutes* is emptied (line 10), since it only makes sense to store the old routes as long as the combination of non-cyclic paths is the same. If a solution to the routing problem does exist, the *Routes* are added to the *PreviousRoutes* and then checked against the *assignment problem* and the *scheduling problem* (lines 12 through 14). If one of these problems turns out to be unsatisifable, then the function *router* will look for another solution; otherwise, when both *assign* and *scheduler* return feasible solutions (lines 12-14) a feasible (sub-optimal) schedule for the overall problem has been found.

Algorithm 1 Compositional Algorithm

Input: graph, jobs
Output: instance, schedule

```
1  UsedPaths ← []
   PreviousRoutes ← []
   instance ← unknown
   PF ← unknown
   RF ← unknown
   AF ← unknown
   SF ← unknown
   Paths ← path_enumerator(graph, jobs)
   while instance = unknown do
2      RF ← unknown
       PF, current_path ← path_selection(Paths, UsedPaths)
       if PF = unsat then
3          instance ← PF
4      else
5          UsedPahts ← current_path
           while RF ≠ unsat ∧ instance = unknown do
6              RF, Routes ← router(current_path, PreviousRoutes)
               if RF = unsat then
7                  if PreviousRoutes = [] then
8                      instance = unsat
9                  end
10                 PreviousRoutes ← []
11             else
12                 PreviousRoutes ← Routes
                   AF, assignment ← assign(Routes)
                   if AF = sat then
13                     SF, schedule ← scheduler(assignment)
                       if SF = sat then
14                         instance ← SF
15                     end
16                 end
17             end
18         end
19     end
20 end
```

5.4 Test Cases

In this section four test cases are presented. The first two test cases are designed to show the behaviour of the monolithic model (Section 5.2) and the compositional algorithm (Section 5.3) when conflicting situations arise. The other two test cases involve a larger graph, a larger number of vehicles and a larger number of possible combinations, to stress-test the monolithic model and compositional algorithm. In all the test cases the depot is located at node *0*; the charging and discharging coefficients are both *1*; in the figures representing the plant in the following subsections, edge weights are reported next to the edge and tasks are next to their locations, together with their time windows, if any. Also, a plain line connecting two nodes n and n' symbolizes that travelling in between those nodes is allowed in both directions, i.e. there is edge *(n,n')* and edge *(n',n)* connecting nodes n and n'.

All the tests are performed on an *Intel Core i7 6700K, 4.0 GHZ, 32GB RAM* running *Ubuntu-18.04LTS* and Z3 4.8.7-64bit. The monolithic model and compositional algorithm performance are compared in terms of running time and quality of the solution (total travelled distance).

Test Case I

In this first test case the goal is to show that two vehicles traversing the same edge do not start at the same time. The layout of the plant is represented in Figure 5.2.

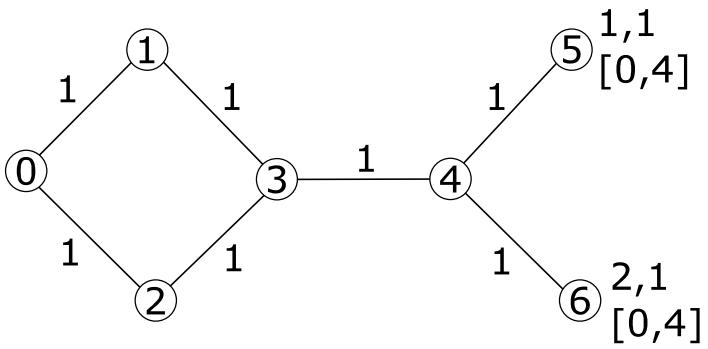

Figure 5.2: Test Case I.

The trivial bound for B would be 16 (the time for the vehicles to execute all jobs in sequence and go back to the depot). However, the problem is so small that pen-and-paper calculation is possible and we decided a reasonable value for B to be 10.

There are only two jobs to execute, each involving only one task. Normally, a job should have at least two tasks, one pickup and one delivery. However, in order to test the correctness of the model in terms of conflict-free routing, this requirement is not necessary and can be relaxed. Also, the weight of each edge, representing the lenght of each segment is 1. The location of task *1* of job *1* is at node *5* and the time window for delivery is $[0, 4]$. The location of task *1* of job *2* is at node *6* and the time window for delivery is also $[0, 4]$. Job *1* and Job *2* must be served by vehicle V_1 and vehicle V_2 respectively, which are both available.

The test case is per se unfeasible. In fact, in order to make it to their destination in time, both vehicles should start traversing node *3* at time-step *2*, which is forbidden by constraints (5.50) and (5.18). When tested, both the monolithic model and compositional algorithm declare the problem *unsat*. In both cases, the running time is below one second. An additional test can be made by extending the time window of one of the jobs by one time-unit, to $[0, 5]$. This time the problem is satisfiable and both methods declare the problem *sat* with a travelled distance of *16*. Once again the solving time is shorter than one second.

Test Case II

In the second test case, the goal is to show that a vehicle can never traverse an edge while another vehicle is coming from the opposite direction. The layout of the plant is represented in Figure 5.3. There are two jobs, each involving two tasks and each job must be executed by a different vehicle, vehicle V_1 executes job *1* and vehicle V_2 executes job *2*. Both job *1* and *2* have one pickup task, located at node *3* and *1* respectively, and one delivery task, located at node *2* and *5* respectively. The time window for both delivery tasks is $[0, 4]$. The trivial bound B is 16; once again we decided to reduce it to 10 after some pen-and-paper calculations. This time travelling between nodes *0* and *3* is only allowed in the direction of the arrow, from node *0* to node *3*. The need for a one-direction segment is to make sure that vehicle V_1 cannot go back to node *0* to after executing task *1* of job *1*; the only way to reach

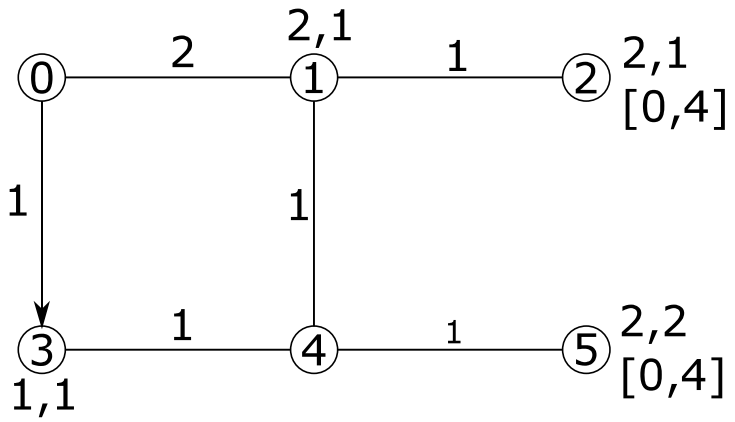

Figure 5.3: Test Case II.

node *2* is via nodes *4* and *1*.

Under these conditions the problem is unfeasible because, in order to reach the delivery location in time, the vehicles should traverse the edge in between nodes *1* and *4* at time-step *2*. Both the monolithic model and the compositional algorithm declare the instance *unsat* in less than a second.

It is possible to make the problem is feasible by extending one of the delivery time windows by at least one time-unit, i.e. $[0, 5]$; in this case both the monolithic model and the compositional algorithm can declare the instance *sat* in less than a second and return the same travelled distance of *15*.

Test Case III

The third case presents a more complex scenario (The layout of the plant is showed in Figure 5.4), involving a graph with twelve nodes, three jobs, each having a different number of tasks, and two vehicles, V_1 and V_2. Also, job *2* can be executed either by vehicle V_1 or V_2, while job *1* must be executed by vehicle V_1 and job *3* by vehicle V_2. A reasonable value for B is 30 (pen-and-paper calculations). Since this problem is computationally more demanding than the previous ones, we decided to test the monolithic algorithm using two different settings, *Solver* and *Optimizer*. When running in *Solver* mode, Z3 will terminate after the first feasible solution is found, whereas if the *Optimizer* mode is on, Z3 will continue the search until the optimal solution is found.

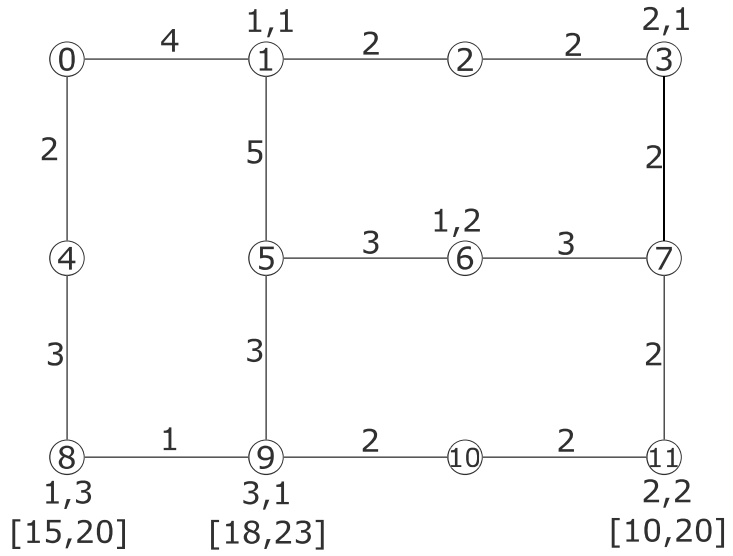

Figure 5.4: Test Case III.

The running time for the monolithic model to find a feasible solution is *13.18* seconds, and the corresponding travelled distance is *61*. On the other hand, when looking for the optimum, the running time rises to *17.99* seconds and the travelling distance is *46*. The compositional algorithm terminates after only *0.62* seconds and also returns the value *46*, which is the true optimum.

Test Case IV

The fourth case is an extension of the third one, where the number of jobs has been increased to four, as well as the number of vehicles and the number of time window's values. Having larger values for the time windows leads to a larger value of B which, together with the a higher number of jobs and vehicles, will increase the model size significantly. As for the previous test cases, a reasonable value of B has been calculated manually and it is 70. In this test case, the value of B can affect the performance of the monolithic model significantly so using the trivial bound was not really an option.

Figure 5.5 shows the layout of the plant. Each job is executed by a different vehicle. As for the previous case, when testing the monolithic model, Z3

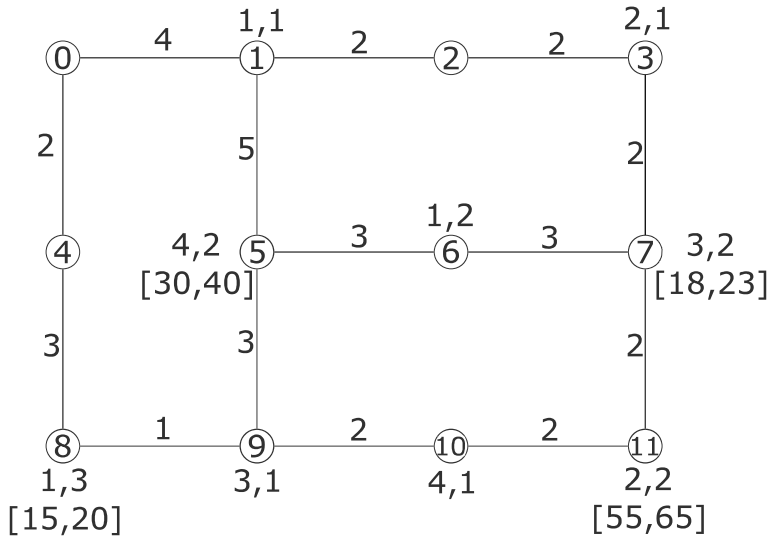

Figure 5.5: Test Case IV.

is run both in *Solver* and *Optimizer* mode. It takes *897* seconds to find the first feasible solution and the corresponding travelling distance is *178*. When running in *Optimizer* mode, the solver was killed after 24 hours and no solution was found. On the other hand, it only takes *1.18* seconds for the compositional algorithm to find a solution with the corresponding travelling distance *90*.

5.5 Results Discussion

In the previous section we evaluated both the correctness and the performance of the monolithic model and the compositional algorithm by formulating four test cases and solve them using the two approaches. Test cases I and II represent small systems where the solution could easily be found even without using a computer aided method; we used them to check that the two approaches work as expected and also to provide an example on how the constraints about conflict-free routing affect the resulting schedule of the vehicles.

Test cases III and IV represent systems where the number of possible schedules is significantly larger; the purpose of these tests was to evaluate the

performance of both the monolithic approach and compositional algorithm in terms of running time, and to compare the results of the monolithic approach, in terms of quality of the solution, with the compositional algorithm.

In the compositional algorithm, the optimization is only in the selection of non-cyclic paths between the points of interest, and in the routing problem based on these paths; in the assignment problem and the scheduling problem the goal is only to check that the routes are consistent with the availability of vehicles and the conflict-free constraints, respectively. Therefore, we have no guarantee on the quality of the overall solution. Nonetheless, the compositional algorithm performed well on Test Case III, since it yielded a solution equal to the solution of the monolithic model, i.e. the true optimum. Unfortunately there is no term of comparison for Test Case IV.

The monolithic model proved, as expected, too slow to handle even relatively small systems such as test cases III and (especially) IV. On the other hand, more testing is required to prove (or disprove) the efficiency of the compositional algorithm but the results we have so far, at least in terms of running time, are promising.

CHAPTER 6

Summary of included papers

This chapter provides a summary of the included papers. In Paper A, the *disjunctive*, *time-index*, and *rank-based* models for the standard job shop problem are adapted for SMT solvers and their performances are compared using Z3; the *disjunctive* model turns out to be the fastest. The performance of the MILP solver Gurobi is then compared with Z3 when both solvers are running the *disjunctive* model, and Z3 outperforms Gurobi on each and every problem instance. In Paper B, the *disjunctive*, *time-index*, and *rank-based* model for the flexible job shop problem are adapted for SMT solvers and once again the *disjunctive* model turns out to be the fastest. In Paper C, a novel formulation of the *time-index* model using bit-vectors is compared to the standard *time-index* formulation; the comparison shows the novel formulation to be faster than the standard one, and also that it results a considerably smaller model. In Paper D and E (not included), a novel formulation for the bin covering and bin packing problem (Paper D only) is presented and compared against the standard formulation using Gurobi (papers D and E) and Z3 (Paper E only). The MILP solver turns out to be much faster than the SMT solver.

6.1 Paper A

Sabino Francesco Roselli, Kristofer Bengtsson, Knut Åkesson
SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation
Proceedings of 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE),
pp. 547–552, Dec. 2018.
 DOI: 10.1109/COASE.2018.8560344 .

This paper presents a comparison of different model formulations for the standard job shop problem, namely the *disjunctive*, *time-index*, and *rank-based* models over a set of generated and benchmark problems. The models are implemented for the state-of-the-art SMT solver Z3. Since the *disjunctive* model shows the best performance in terms of solving time, it is selected for further comparison against the state-of-the-art MILP solver Gurobi; Z3 outperforms Gurobi on each and every instance and, in general, is able to solve to optimality medium size problems within 1200 seconds.

6.2 Paper B

Sabino Francesco Roselli, Kristofer Bengtsson, Knut Åkesson
SMT Solvers for Flexible Job-Shop Scheduling Problems: A Computational Analysis
Proceedings of 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE),
pp. 673–678, Sep. 2019.
 DOI: 10.1109/COASE.2019.8843025 .

Based on the results achieved in Paper A, Paper B presents an evaluation of the extended versions of the *disjunctive*, *time-index*, and *rank-based* models, adapted for the flexible job shop problem. Once again the implementation is done for the SMT solver Z3; the computational analysis is run over a set of generated and benchmark instances and shows the *disjunctive* model to be the fastest.

6.3 Paper C

Sabino Francesco Roselli, Kristofer Bengtsson, Knut Åkesson
Compact Representation of Time-Index Job Shop Problems Using a Bit-Vector Formulation
Published in 2020 IEEE 16th International Conference on Automation Science and Engineering (CASE) .

This paper presents a novel implementation of the *time-index* model for the standard job shop problem, based on bit-vectors. The new formulation can compress the high number of variables (a consequence of time discretization) into bit-vectors; dealing with bit-vectors allow to use bit-wise operators that reduce drastically the amount of constraints in the model. As a result, the new model is much smaller in size, making the model generation phase negligible compared to the solving phase. Moreover, when evaluated with two different SMT solvers, the bit-vector based model outperformed the standard *time-index* model in both cases.

6.4 Paper D

Sabino Francesco Roselli, Fredrik Hagebring, Sarmad Riazi, Martin Fabian, Knut Åkesson
On the Use of Equivalence Classes for Optimal and Sub-Optimal Bin Packing and Bin Covering
Conditionally accepted to 2020 IEEE Transactions on Automation Science and Engineering (TASE) .

This paper presents a study of the bin sorting problem and how to improve the performance by cutting off portions of the search-space. The improvement is designed for an enumerative model, where decision variables represent potential bins; reasoning about the problem constraints allows to eliminate all those potential bins that, given their size (in terms of cumulative value of the items within them), could never be part of an optimal solution. The new approach shows good performance when compared to the standard formulation and can be implemented for both *packing* and *covering* problems. It is supposed to scale very well in terms of number of items, though being based on potential combinations of items, a wide range of values for the items can quickly lead to a state-space explosion.

CHAPTER 7

Concluding Remarks and Future Work

The work presented in this thesis is focused on finding efficient solutions to solve the vehicle routing problem. Since this problem can present itself in many different forms, we tried to develop a general framework that can be flexible enough to be quickly adapted to handle additional features that can arise in real world scenarios. As flexibility is a requirement, we decided to test the performance of general purpose SMT solvers for well-known combinatorial optimization problems, in order to find out about strenghts and weaknesses of this technology. As a base-line for comparison, we used MILP solvers, since these are a well established general framework for linear optimization problems.

When testing these two technologies on the job shop problem, SMT turned out to be significantly faster than MILP, while for the bin sorting problem it was the other way around. In order to understand why in some cases MILP outperforms SMT while in some other cases it is the opposite, we investigated the algorithms running under the hood of these two technologies. We found that MILP solvers can handle conjunctions of linear constraints but are not as efficient to handle logical constraints. On the other hand, SMT solvers are not as efficient as MILP solvers when dealing with models involving only

conjunctions of linear constraints, but they can handle logical constraints much better.

Based on the acquired knowledge, and based on the fact that the vehicle routing problem we deal with in the thesis presents several logical constraints, we decided to use the SMT solver Z3 to develop a decomposition algorithm for the VRP. We also formulated a monolithic model to handle all the constraints of the problem at once; we did not expect the monolithic model to be very efficient in solving instances of the VRP but we needed something to compare with, both in terms of running time and solution quality (total travelled distance).

We developed some test cases to test the two approaches and compare their performance. When solving larger instances, the monolithic model became rather slow; on the other hand, the decomposition algorithm could provide a solution in just a few seconds. For one of the larger test cases we were able to compare the *true optimum* yielded by the monolithic model and it turned out to be the same value yielded by the compositional algorithm; hence, the algorithm found the *true optimum*.

Based on our findings, we can give an answer to the research questions we formulated in Chapter 1:

RQ1 What are the strengths and weaknesses of SMT solvers when used to solve industrial problems?

Given that MILP and SMT solvers can model the same problems, SMT solvers are always preferable in terms of ease of modelling, since they allow for logical constraints over literals of different theories, including combinations of them; when using MILP solvers instead, only conjunctions of inequalities over reals and integers are allowed, therefore modelling may be trickier. For instance, when modelling the JSP, the non-overlapping of operations sharing the same resource can be directly modelled with an SMT solver as a disjunction of two linear inequalities. On the other hand, modelling the non-overlapping constraint with MILP is only possible with the support of an auxiliary binary variable and a large enough constant (the *big M* method), as shown in Section 3.3.

When it comes to performance, SMT solvers are preferable when the problem structure involves logical constraints such as the disjunctive constraints for the JSP, as discussed in papers A and B. On the other hand, MILP solvers

are faster if the problem mainly involves conjunctions of inequalities; an example of this is the equivalence class method presented in Paper D. The method involves a linear model with only conjunctions of inequalities over integer variables; the comparison of Z3 and Gurobi for problems formulated using the equivalence class method showed that Gurobi outperformed Z3.

RQ3 How can the strengths of SMT and/or MILP solvers be exploited and combined to design an efficient algorithm for a real-world VRP?

When breaking down the problem into sub-problems, using SMT for the parts that involve logical constraints and MILP when the sub-problem is mainly a conjunction of inequalities can make the best of each technology. For instance, in the compositional algorithm presented in Chapter 5 the *path finder* can be modelled using only conjunctions of inequalities, hence a MILP solver could be the best option; the *routing problem* involves logical constraints, therefore an SMT solver would be a better choice; the *assignment* and *scheduling* problems are modelled as variants of the JSP and our computational analysis presented in papers A and B shows the superiority of SMT over MILP when solving a JSP, hence SMT is the best option to solve these two sub-problems.

7.1 Future Work

As of today, the compositional algorithm can only be used on specific types of problems, where certain restrictions apply, as described in Chapter 5. Also, the search for feasible solutions is currently random. Whenever an unfeasible route or combination of paths is found, the solution is stored so that it will be ruled out in the next iteration. While this strategy will eventually lead to finding a solution, if such exist, or declare the problem *unsat*, it is not very efficient; the search could be driven by information acquired during computation of the previous infeasible solutions.

Furthermore, when comparing the quality of the solutions yielded by the two approaches we used the total travelled distance. We did so because it is a standard parameter to optimize in a VRP. However, the total travelled distance may not be the best option. Other possibilities could be the minimization of the make-span, or the number of vehicles out on the plant at the same time (to avoid congestions).

Finally, the decomposition algorithm only makes use of an SMT solver, while using MILP, for instance in the *path finder sub-problem*, could provide better performance.

In the future work we plan to generalize the algorithm so it can be applied to different versions of the VRP; we want to explore other possibilities to measure the quality of the solution; we also want to investigate how to make the algorithm more efficient, making use of the knowledge gained during the previous iterations to drive the search for new solutions and possibly using a MILP solver in combination with the SMT solver.

References

[1] W. Zhang, *State-space search: Algorithms, complexity, extensions, and applications*. Springer Science & Business Media, 1999.

[2] M. Sakawa and T. Mori, "An efficient genetic algorithm for job-shop scheduling problems with fuzzy processing time and fuzzy duedate", *Computers & industrial engineering*, vol. 36, no. 2, pp. 325–341, 1999.

[3] D. C. Mattfeld and C. Bierwirth, "An efficient genetic algorithm for job shop scheduling with tardiness objectives", *European journal of operational research*, vol. 155, no. 3, pp. 616–630, 2004.

[4] C. S. Chong, M. Y. H. Low, A. I. Sivakumar, and K. L. Gay, "A bee colony optimization algorithm to job shop scheduling", in *Proceedings of the 2006 winter simulation conference*, IEEE, 2006, pp. 1954–1961.

[5] E. Kondili, C. C. Pantelides, and R. W. Sargent, "A general algorithm for short-term scheduling of batch operations—I. MILP formulation", *Computers & Chemical Engineering*, vol. 17, no. 2, pp. 211–227, 1993.

[6] C. Wouters, E. S. Fraga, and A. M. James, "An energy integrated, multi-microgrid, MILP (mixed-integer linear programming) approach for residential distributed energy system planning–a south australian case-study", *Energy*, vol. 85, pp. 30–44, 2015.

[7] M. S. Dasika, A. Gupta, and C. D. Maranas, "A mixed integer linear programming (MILP) framework for inferring time delay in gene regulatory networks", in *Biocomputing 2004*, World Scientific, 2003, pp. 474–485.

[8] P. Luathep, A. Sumalee, W. H. Lam, Z.-C. Li, and H. K. Lo, "Global optimization method for mixed transportation network design problem: A mixed-integer linear programming approach", *Transportation Research Part B: Methodological*, vol. 45, no. 5, pp. 808–827, 2011.

[9] C. Audet and W. Hare, *Derivative-free and blackbox optimization*. Springer, 2017, ISBN: 978-3-319-68912-8.

[10] M. W. Krentel, "The complexity of optimization problems", in *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, 1986, pp. 69–76.

[11] G. B. Dantzig, *Linear programming and extensions*. Princeton university press, 1998, vol. 48.

[12] Y. Xia and J. Wang, "A general projection neural network for solving monotone variational inequalities and related optimization problems", *IEEE Transactions on Neural Networks*, vol. 15, no. 2, pp. 318–328, 2004.

[13] S. Wang, X. Liao, M. Wang, L. Chang, H. Yang, and T. Wang, "An improved SMT-based scheduling for overloaded real-time systems.", *Engineering Letters*, vol. 28, no. 1, 2020.

[14] M. Hekmatnejad, G. Pedrielli, and G. Fainekos, "Task scheduling with nonlinear costs using SMT solvers", in *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, IEEE, 2019, pp. 183–188.

[15] D. Beyer, M. Dangl, and P. Wendler, "A unifying view on SMT-based software verification", *Journal of Automated Reasoning*, vol. 60, no. 3, pp. 299–335, 2018.

[16] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, *et al.*, "Satisfiability modulo theories", *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[17] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon, "The international SAT solver competitions", *Ai Magazine*, vol. 33, no. 1, pp. 89–92, 2012.

[18] A. S. Manne, "On the job-shop scheduling problem", *Operations Research*, vol. 8, no. 2, pp. 219–223, 1960.

[19] M. G. S. Furtado, P. Munari, and R. Morabito, "Pickup and delivery problem with time windows: A new compact two-index formulation", *Operations Research Letters*, vol. 45, no. 4, pp. 334–341, 2017.

[20] L. V. Kantorovich, "Mathematical methods of organizing and planning production", *Management science*, vol. 6, no. 4, pp. 366–422, 1960.

[21] J. F. Bard, L. Huang, M. Dror, and P. Jaillet, "A branch and cut algorithm for the VRP with satellite facilities", *IIE transactions*, vol. 30, no. 9, pp. 821–834, 1998.

[22] S. Erdoğan and E. Miller-Hooks, "A green vehicle routing problem", *Transportation Research Part E: Logistics and Transportation Review*, vol. 48, no. 1, pp. 100–114, 2012.

[23] M. Schneider, A. Stenger, and D. Goeke, "The electric vehicle-routing problem with time windows and recharging stations", *Transportation Science*, vol. 48, no. 4, pp. 500–520, 2014.

[24] J. Lin, W. Zhou, and O. Wolfson, "Electric vehicle routing problem", *Transportation Research Procedia*, vol. 12, no. Supplement C, pp. 508–521, 2016.

[25] G. Hiermann, J. Puchinger, S. Ropke, and R. F. Hartl, "The electric fleet size and mix vehicle routing problem with time windows and recharging stations", *European Journal of Operational Research*, vol. 252, no. 3, pp. 995–1018, 2016.

[26] A. Wallar, J. Alonso-Mora, and D. Rus, "Optimizing vehicle distributions and fleet sizes for shared mobility-on-demand", in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 3853–3859.

[27] M. Salazar, F. Rossi, M. Schiffer, C. H. Onder, and M. Pavone, "On the interaction between autonomous mobility-on-demand and public transportation systems", in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, IEEE, 2018, pp. 2262–2269.

[28] N. N. Krishnamurthy, R. Batta, and M. H. Karwan, "Developing conflict-free routes for automated guided vehicles", *Operations Research*, vol. 41, no. 6, pp. 1077–1090, 1993.

[29] G. Desaulniers, J. Desrosiers, and M. M. Solomon, *Column generation*. Springer Science & Business Media, 2006, vol. 5.

[30] A. I. Corréa, A. Langevin, and L.-M. Rousseau, "Scheduling and routing of automated guided vehicles: A hybrid approach", *Computers & operations research*, vol. 34, no. 6, pp. 1688–1707, 2007.

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

[32] T. Miyamoto and K. Inoue, "Local and random searches for dispatch and conflict-free routing problem of capacitated AGV systems", *Computers & Industrial Engineering*, vol. 91, pp. 1–9, 2016.

[33] K. Murakami, "Time-space network model and MILP formulation of the conflict-free routing problem of a capacitated AGV system", *Computers & Industrial Engineering*, p. 106 270, 2020.

[34] E. Thanos, T. Wauters, and G. Vanden Berghe, "Dispatch and conflict-free routing of capacitated vehicles with storage stack allocation", *Journal of the Operational Research Society*, pp. 1–14, 2019.

[35] D.-H. Lee, J. X. Cao, Q. Shi, and J. H. Chen, "A heuristic algorithm for yard truck scheduling and storage allocation problems", *Transportation Research Part E: Logistics and Transportation Review*, vol. 45, no. 5, pp. 810–820, 2009.

[36] M. R. Lambrecht, P. L. Ivens, and N. J. Vandaele, "ACLIPS: A capacity and lead time integrated procedure for scheduling", *Management Science*, vol. 44, no. 11-part-1, pp. 1548–1561, 1998.

[37] J. Lundgren, M. Rönnqvist, and P. Värbrand, *Optimization*. Lund, Sweden: Studentlitteratur, 2010, ISBN: 9789144053080.

[38] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver", in *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*, IEEE, 2001, pp. 279–285.

[39] M. J. Heule, M. Järvisalo, and M. Suda, "SAT competition 2018", *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 11, no. 1, pp. 133–154, 2019.

[40] G. Tseitin, "On the complexity of proofs in propositional logics", in *Seminars in Mathematics*, vol. 8, 1970, pp. 466–483.

[41] L. Hadarean, K. Bansal, D. Jovanović, C. Barrett, and C. Tinelli, "A tale of two solvers: Eager and lazy approaches to bit-vectors", in *International Conference on Computer Aided Verification*, Springer, 2014, pp. 680–695.

[42] R. Bruttomesso, "An extension of the davis-putnam procedure and its application to preprocessing in SMT", in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, 2009, pp. 14–19.

[43] D. Kroening and O. Strichman, *Decision procedures - An Algorithmic Point of View*. Springer, 2016.

[44] R. Nieuwenhuis and A. Oliveras, "On SAT modulo theories and optimization problems", in *International conference on theory and applications of satisfiability testing*, Springer, 2006, pp. 156–169.

[45] N. Bjørner and A.-D. Phan, "νZ-maximal satisfaction with Z3.", *SCSS*, vol. 30, pp. 1–9, 2014.

[46] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νZ-an optimizing SMT solver", in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2015, pp. 194–199.

[47] B. Dutertre and L. De Moura, "A fast linear-arithmetic solver for DPLL (T)", in *International Conference on Computer Aided Verification*, Springer, 2006, pp. 81–94.

[48] R. Sebastiani and P. Trentin, "OptiMathSAT: A tool for optimization modulo theories", in *International Conference on Computer Aided Verification*, Springer, 2015, pp. 447–454.

[49] F. Trespalacios and I. E. Grossmann, "Improved big-M reformulation for generalized disjunctive programs", *Computers & Chemical Engineering*, vol. 76, pp. 98–103, 2015.

[50] H. Fisher, "Probabilistic learning combinations of local job-shop scheduling rules", *Industrial scheduling*, pp. 225–251, 1963.

[51] R. H. Storer, S. D. Wu, and R. Vaccari, "New search spaces for sequencing problems with application to job shop scheduling", *Management science*, vol. 38, no. 10, pp. 1495–1509, 1992.

[52] A. Mascis and D. Pacciarelli, "Job-shop scheduling with blocking and no-wait constraints", *European Journal of Operational Research*, vol. 143, no. 3, pp. 498–517, 2002.

[53] W.-Y. Ku and J. C. Beck, "Mixed integer programming models for job shop scheduling: A computational analysis", *Computers & Operations Research*, vol. 73, pp. 165–173, 2016.

[54] IBM. (2015). 12.6 user's manual, [Online]. Available: `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.2/ilog.odms.studio.help/pdf/usrcplex.pdf`.

[55] Gurobi. (2018). Gurobi optimizer reference manual, [Online]. Available: `http://www.gurobi.com`.

[56] E. F. Moore, "The shortest path through a maze", in *Proc. Int. Symp. Switching Theory, 1959*, 1959, pp. 285–292.

Part II

Papers

PAPER A

SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation

Sabino Francesco Roselli, Kristofer Bengtsson, Knut Åkesson

Proceedings of 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE),
pp. 547–552, Dec. 2018.
 DOI: 10.1109/COASE.2018.8560344

The layout has been revised.

Abstract

The optimal assignment of jobs to machines is a common problem when implementing automated production systems. A specific variant of this category is the job-shop scheduling problem (JSP) that is known to belong to the class of NP-hard problems. JSPs are typically either formulated as Mixed Integer Linear Programming (MILP) problems and solved by general-purpose-MILP solvers or approached using heuristic algorithms specifically designed for the purpose.
During the last decade a new approach, satisfiability (SAT), led to develop solvers with incredible abilities in finding feasible solutions for hard combinatorial problems on Boolean variables. Moreover, an extension of SAT, Satisfability Modulo Theory (SMT), allows to formulate constraints involving linear operations over integers and reals and some SMT-solvers have been also extended with an optimizing tool.
Since the JSP is a well-known hard combinatorial problem, it is interesting to evaluate how SMT-solvers perform in solving it and how they compare to traditional MILP-solvers. We therefore evaluate state-of-the-art MILP and SMT solvers on benchmark JSP instances and find that general-purpose open-source SMT-solvers are competitive against commercial MILP-solvers.

1 Introduction

The Job-shop scheduling problem (JSP) is a well known problem within the Operation Scheduling Community, where the target is to allocate resources to operations while minimizing some cost function. In [1] a thorough study on the subject is presented, providing information about the evolution of techniques and algorithms to deal with the JSP whether the target was the true optimal or an approximation of it.

Industrial scheduling problems are typically extensions of the pure job-shop problem described above. Additional extensions are the possibility to have alternative resources that can process an operation, constraints on setup-time

for resources, constraints on the idle-time between operations, etc. In many applications it might be desired to not minimize the make-span but instead make scheduling decisions based on given deadlines for operations and then minimize the tardiness or maximum lateness.

A recent study by [2] shows promising results in solving JSP problems by employing Constraint Programming (CP) to implement Local-Search (LS) algorithms. A similar solution has also been implemented by Beck et Al. [3], leading again to good results. Apparently the combination of CP and LS is a powerful instrument to takle the JSP, although also other techniques are used in practice. In fact, according to Beck himself, as shown in [4], both in industry and academia, Mixed Integer Linear Programming (MILP) is largely employed for the JSP. Based on this the authors did a benchmark on three popular MILP solvers, IBM ILOG CPLEX [5], Gurobi [6], and SCIP [7]. CPLEX and Gurobi are considered to be state-of-the-art commercial solvers [8], while SCIP is a fast non-commercial solver. In [4] the authors also compared alternative MILP problem formulations that have been proposed in the literature for the classical job-shop problem. The tools were evaluated on a large set of benchmark job-shop problems. The result of the benchmark is that for the tools evaluated, the performance of CPLEX and Gurobi were comparable and significantly better than SCIP and that the traditional disjunctive formulation of the job-shop problem was superior to both Time-Index and Rank-Based formulations.

While many industrial problems can be solved by general purpose MILP-solvers, the combinatorial complexity of the JSP implies that for sufficiently large problems it will not be possible to find an optimal solution within reasonable time. For such problems, specific-purpose methods have been developed: they often make use of heuristics as well as hybrid techniques including local search and genetic algorithms, e.g. [9] and [10]), that lead to *good enough* solutions in a reasonable amount of time [11].

An alternative technology to solve combinatorial problems have emerged within the community of formal verification of hardware and software: satisfability search (SAT) [12]. This approach consists of expressing the problem through Boolean variables in Conjunctive Normal Form (CNF) and determine whether there exists an assignment to these variables such that the formula evaluates to true. Today SAT-solvers can deal with large combinatorial problems by efficiently exploiting their structure and generate a valid model in a reasonable time, even when the problem happens to be NP-complete. Of

course this does not mean that there are not NP-complete problems that will take exponential time also for SAT-solvers, but that many man-made models are no longer intractable.

On the other hand, Boolean logic is in many cases not expressive enough for representing many real-world problems where first-order logic is used together with integers and reals. To handle this type of models and at the same time take advantage of the very performant SAT-solvers, a new approach, called Satisfability Modulo Theory (SMT), [13], [14] has been developed. SMT-solvers implements special decision procedures, so called theories, to extend to the original Boolean satisifiability problem.

Both SAT and SMT are employed to prove satisfability of formulas; it is often the case that not only one but several satisfable solutions exist for a given formula and each of them has a cost referring to some parameter related to the problem itself. It is therefore possible to turn the satisfability problem into an optimization one by setting an objective function where one not only wants to find a satisfable solution, but the optimal one in respect of the cost we assign to the variables.

Ever since SAT and SMT solvers started spreading and being employed in real-world applications, users have built their own custom loops to achieve optimality, often based on heuristics tuned for their specific problems. Lately, some of these solvers have been extended by including optimizing tools that make use of state-of-the-art algorithms to deal with a list of different problems on SMT formulas with linear objective functions on Boolean, rational and integer domain (or a combination of them).

The performance of SMT-solvers are evaluated annually in a benchmark competition. The latest competition was the The 12th International Satisfiability Modulo Theories Competition (SMT-COMP 2017). The SMT-solvers compete in different categories in which Z3 [15], MathSAT5 [16], and CVC4 [17] have shown consistently good performance. Among these solvers Z3 and MathSat5 have versions that support optimization as well. The optimizing version of Z3 is described in [18] and is included in the latest version of Z3. MathSat5 optimizing tool is a special version of it named OptiMathSAT [19]. CVC4 does not have support for optimization and is thus excluded from this benchmark.

In order to have a valid comparison with the current technology, we also tested the models on a MILP solver. We chose Gurobi, since it is considered

to be among the state-of-the-art solvers in such field.

The contributions in this paper are. (i) Adapting three existing formulations of the classical job-shop problem to make them suitable for SMT-solvers. (ii) Benchmarking two optimizing SMT-solvers on a set of benchmark job-shop problems.

In the next section the problem is described in detail, providing all necessary information to understand the models. In section three, we describe Z3 implementation for such models, we compare Z3 performance with Gurobi and OptiMathSAT using the Disjunctive model and we discuss the results. Finally we draw conclusions in section four.

2 Problem Description

The JSP problem consists of a set of n jobs $J = \{j_i\}_{i=1}^{n}$, where each job has its own processing order through a set of k machines, $M = \{m_i\}_{i=1}^{k}$. Operations are defined as the execution of a job on a certain machine and, as each job has to visit each machine, the total number of operations in the problem is $n \cdot k$. Each job will go through all machines sequentially. Let o_i^j model the index of the machine to be used for job j executing operation i in sequence. The index of the machines for each step in the job sequence is thus given by $(o_1^j, \ldots, o_i^j, \ldots, o_k^j)$. Also, let d_i^j model the duration of the execution of the same operation.

Finding a solution to the job-shop scheduling problem means to assign operations to machines so that all jobs are completed. The constraints in this kind of problem are two:

- as there exist a sequence of operations for each job, operations belonging to the same job must be executed in the right order;
- operations requiring the same machine and belonging to different jobs cannot overlap in time.

Given these two constraints, the target is to find a feasible schedule such that the overall make-span is minimized. Finding an optimal schedule is an NP-complete problem, [20].

We now present the three model formulations, (i) the Disjunctive model, (ii) the Time-Index model, and (iii) the Rank-Based model for the classical JSP, based on [4]. They are adapted to fit the SMT language.

2.1 Time-Index Model

In this model the execution time is split into steps, whose length is the minimum time unit. For instance, if the duration of an operation is n-seconds (or minutes), n steps will be taken since it starts and until it is completed. In order to create a model with such feature, we need to have a guess of the overall execution time, in other words, an upper bound H. A trivial upper bound is the sum of all operations duration as if they were executed in a sequence (the worst case scenario). The greater the upper bound, the longer it takes to create the model and therefore the slower the overall execution. Nevertheless, as finding good upper bounds for such model is beyond the scope of this paper, the trivial one is used. In this model the decision variable is:

- s_{mjt} is a Boolean variable that is true if job j starts on machine m at time t.

minimize T_{max} subject to

$$\bigvee_{t=0}^{H} s_{mjt} \qquad \forall m \in M, j \in J \tag{A.1}$$

$$s_{mjt} \rightarrow \bigwedge_{t'=0}^{t-1} \neg s_{mjt'} \wedge \bigwedge_{t'=t+1}^{H} \neg s_{mjt'}$$
$$\forall t = 1, \ldots, H, m \in M, j \in J \tag{A.2}$$

$$s_{mjt} \rightarrow \bigwedge_{t'=t}^{t+p_{mj}} \neg s_{mj't'}$$
$$\forall j, j' \in J, j \leq j', t = 1, \ldots, H, m \in M \tag{A.3}$$

$$s_{mjt} \rightarrow T_{max} \geq t + d_{mj}$$
$$\forall m \in M, j \in J, t = 1, \ldots, H \tag{A.4}$$

$$x_{o^j_{i-1}jt} \rightarrow \bigwedge_{t'=0}^{t+d^j_{i-1}} \neg x_{o^j_{i-1}jt'}$$
$$\forall i = 2, \ldots, k, t = 1, \ldots, H, j \in J \tag{A.5}$$

Equation (A.1) and (A.2) are needed to make sure that one and only one

operation is executed on a machine per each time step. Equation (A.3) allows each machine to execute only one operation at a time. Equation (A.4) sets the objective function as larger than operations completion times. Equation (A.5) takes care of the precedence constraint among operations of the same job.

2.2 Disjunctive Model

The decision variables in this model are as follows:

- s_{mj} is an integer variable and models the start time of job j on machine m.

$$\begin{aligned}
&\text{minimize } T_{max} \text{ subject to} \\
&s_{mj} \geq 0 && \forall j \in J, m \in M && \text{(A.6)} \\
&T_{max} \geq s_{o^j_k,j} + d_{o^j_k,j} && \forall j \in J && \text{(A.7)} \\
&s_{o^j_i,j} \geq s_{o^j_{i-1},j} + d_{o^j_{i-1},j} && \forall j \in J, i = 2, \ldots, k && \text{(A.8)} \\
&s_{mj} \geq s_{mj'} + d_{mj'} \vee s_{mj'} \geq s_{mj} + d_{mj} \\
& && j, j' \in J, j \leq j', m \in M && \text{(A.9)}
\end{aligned}$$

In this model equation (A.6) restricts variables domain to be larger than or equal to zero, as they represent the start time of operations and thus they can not be negative. Equation (A.7) impose the objective function to be lager than or equal to the start time of the last operation of each job plus its duration. Equation (A.8) is about precedence constraints among the operations belonging to the same job: one can not start until the previous one is over. Equation (A.9) takes care of resource sharing by stating that two operations sharing the same resource cannot take place at the same time: either one starts once the other is over or the other way around.

2.3 Rank-Based Model

In this model the focus is on the machine side: each machine has as many positions as operations in a job or, in other words, a position is the cardinal step in the execution sequence of all jobs. Finding a schedule for this model

means to find in which position a job is executed on a certain machine. The decision variables are defined as follows:

- x_{mjt} is a Boolean variable indicating whether job j is executed on machine m at the t-th position
- s_{mt} is an Integer variable representing the starting time of position t of machine m

minimize T_{max} subject to

$$s_{mt} \geq 0 \qquad \forall m \in M, t = 1 \dots k \tag{A.10}$$

$$x_{o_k^j jt} \rightarrow T_{max} \geq s_{o_k^j t} + d_{o_k^j j} \qquad \forall j \in J, t = 1 \dots k \tag{A.11}$$

$$x_{mjt} \rightarrow s_{mt} + d_{mj} \leq s_{m,t+1} \qquad \forall t = 1 \dots k, m \in M, j \in J \tag{A.12}$$

$$(x_{o_{i-1}^j jt_1} \wedge x_{o_i^j jt_2}) \rightarrow s_{o_{i-1}^j t_1} + d_{o_{i-1}^j j} \leq s_{o_i^j t_2} \qquad \forall t_1, t_2 = 1 \dots k, i = 2 \dots k, j \in J \tag{A.13}$$

$$\left(x_{mjt} \rightarrow \bigwedge_{t'=0}^{t-1} \neg x_{mjt'} \wedge \bigwedge_{t'=t+1}^{k} \neg x_{mjt'}\right) \wedge \bigvee_{k=0}^{k} x_{mjt} \qquad \forall t = 1 \dots k, m \in M, j \in J \tag{A.14}$$

$$\left(x_{mjt} \rightarrow \bigwedge_{j'=0}^{j-1} \neg x_{mj't} \wedge \bigwedge_{j'=j+1}^{n} \neg x_{mj't}\right) \wedge \bigvee_{t=0}^{k} x_{mj't} \qquad \forall t = 1 \dots k, m \in M, j \in J \tag{A.15}$$

Equation (A.10) restricts the start variable domain to the natural numbers. Equation (A.11) sets the objective function. Equation (A.12) ensures the precedence constraint among the operations executed on a machine, while equation (A.13) takes care of the operations precedence within the same job. Equation (A.14) ensures that each job can be assigned only once to a certain machine, while equation (A.15) states that a position can be assigned only to one job.

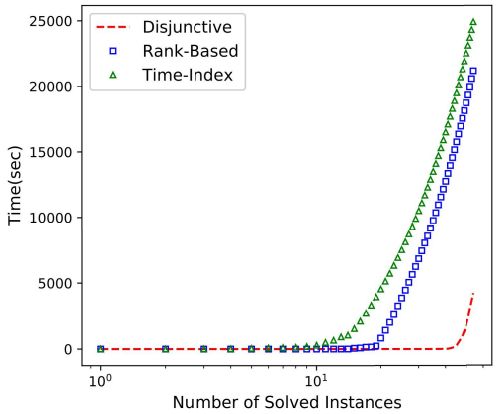

Figure 1: Performance comparison between Disjunctive, Time-Index and Rank-Based models over the benchmark instances. The maximum time allowed for each instance is 600 seconds.

3 Experiments

The solvers whose performance were compared are Z3-4.6.0, Gurobi-7.5.2 and OptiMathSAT-1.5.0. The time limit is 600 seconds. Solvers are run in their default setting. The instances used for the comparison are either generated through an instance generator or taken from benchmark sets. All the experiments were performed on an *Intel Core i7 6700K, 4.0 GHZ, 32GB RAM* running *Ubuntu-16.04*.

An instance is a matrix of integers where each row represents a job; for each row the odd elements represent the machine needed to execute the operation whose duration is pointed out by the next even element. The execution order is given by the position within the row. The instances used in the experimental phase are:

- Generated instances: instances generated according to Taillard instance generator[21] of small-medium size (from 3x3 to 9x9);
- Lawrence [22]: forty instances of increasing size from 10x5 to 30x10;
- Applegate and Cook [23]: ten instances of size 10x10;

Table 1: Comparison of models implemented using Z3. The time showed in the table is the geometric mean calculated over all the instances belonging to the category they refer to. For each class the number of solved instances (out of the total number of instances belonging to such class) is given. The symbol '-' means that no instance has been solved. The symbol '*' means that the time limit for the model translation into SMT-lib2 has been exceeded.

Problems	Disjunctive		Rank-Based		Time-Index	
	Time	**Opt**	**Time**	**Opt**	**Time**	**Opt**
Generated Instances						
3x3	0.01	5/5	0.18	5/5	3.66	5/5
4x4	0.01	5/5	0.110	5/5	32.76	5/5
5x5	0.02	5/5	1.196	5/5	164.67	5/5
6x6	0.04	5/5	38.904	5/5	528.31	2/5
7x7	0.12	5/5	535.25	1/5	-	0/5
8x8	0.27	5/5	-	0/5	*	*
9x9	0.18	5/5	-	0/5	*	*
Applegate Instances						
10x10	7.28	10/10	-	0/5	*	*
Taillard Instances						
15x15	240.84	7/10	-	0/5	*	*

- Storer [24]: five instances of size 20x10;
- Taillard [21]: ten instances of size 15x15 and ten of size 20x15.
- Fisher&Thompson [25]: one instance of size 20x5.

3.1 Models Comparison

In this phase the three models presented in the previous section are compared Z3 (Figure 1). We decided to employ the geometric mean to reduce the effect of outliers. The instance generation has been carried out by randomly assigning values belonging to the interval $[1, 20]$ to the operations. Five instances have been generated per each class going from the size 3x3 to 9x9.

The models have been created through the Python API for Z3 and then translated into the SMT-lib2 format [26] to be run directly and avoid possible delays due to the conversion while measuring the execution time. Since the Time-Index model scales up very bad in size an additional timeout has been

set for the model translation into SMT format and the * symbol in the table denotes that such time limit has been exceeded. The results show that the Time-Index model can easily solve very small-size instances but it scales bad and provides no solution for problems larger than 6x6. The Rank-Based model is more efficient in terms of time but it can solve only some instances more than Time-Index.

The Disjunctive model, on the other hand, takes only few milliseconds to solve the smaller instances and it stays far below one second while dealing with instances up to 9x9 size. The average time increases by over forty times for solving Applegate instances due to some outliers that take over one minute but the result is still remarkable if compared to the other two models, even when it comes to Taillard Instances, although only seven out ten instances can be solved within ten minutes.

An additional test has been run on some hard-to-solve instances, to check how the best solution increases over time, as shown in Figure 2. Usually a solution close to the optimal is found quickly but then it is hard to improve it. This might be due to the solver getting stuck in some local optimum.

3.2 Solvers Comparison

The second phase is about comparing different solvers using the Disjunctive model. When running the Disjunctive model on Applegate instances, OptiMathSAT could solve six out of ten of them and it was on average four times slower than Gurobi, which could solve seven. Z3 could solve all of them in less than eight seconds each. All the solvers could easily deal with the 10x5 instances from Lawrence and while Z3 solution was almost instantaneous Gurobi and OptiMathSAT had similar performance taking around twenty seconds to produce a solution. The only other class of instances Gurobi and OptiMathSAT were able to find a solution for is Lawrence 10x10 and also in this case Gurobi was quite faster than OptiMathSAT (around ten times) and both much slower than Z3, which was also able to solve some of Lawrence instances 15x10 in less than five minutes, 15x15 in less than three, and seven of the Taillard instances 15x15, as shown also in the previous section. No other solution was found within the given time.

Table 2: Comparison of SMT and MILP solvers running the Disjunctive Model over the benchmark instances. The time showed in the table is the geometric mean calculated over all the instances belonging to the category they refer to. For each class the number of solved instances (out of the total number of instances belonging to such class) is given. The symbol '-' means that no instance has been solved.

	Z3		**OptiMathSAT**		**Gurobi**	
Problems	**Time**	**Opt**	**Time**	**Opt**	**Time**	**Opt**
Applegate Instances						
10*10	7.28	10/10	276.14	6/10	60.13	7/10
Lawrence Instances						
10x5	0.43	5/5	18.41	5/5	16.21	5/5
15x5	-	0/5	-	0/5	-	0/5
20x5	-	0/5	-	0/5	-	0/5
10x10	0.56	5/5	215.85	5/5	18.20	5/5
15x10	263.61	3/5	-	0/5	-	0/5
20x10	-	0/5	-	0/5	-	0/5
30x10	-	0/5	-	0/5	-	0/5
15x15	156.37	2/5	-	0/5	-	0/5
Storer Instances						
20x10	-	0/5	-	0/5	-	0/5
Taillard Instances						
15x15	240.84	7/10	-	0/10	-	0/10
20x15	-	0/10	-	0/10	-	0/10

3.3 Results Discussion

The results presented in the previous section show a big difference in performance between the two SMT solvers. When compared to Gurobi, Z3 is typically faster and can provide an optimal solution to larger instances within the given time limit. OptiMathSAT, although employing the same technology, is considerably slower than Z3 and, in most cases, also than Gurobi. Since Z3 and OptimathSAT have shown comparable performance in previous works [19], it might be the case that Z3 heuristic and linear optimization algorithms suits better the JSP problem than OptiMathSAT's. By having a look under the hood of νZ we found out it employs a portfolio of different approaches to solve optimization problems [18]. Among them are some very efficient algorithms to deal with linear arithmetic using Simplex over non-standard numbers to find unbounded objectives, as explained by Z3 developers in [27].

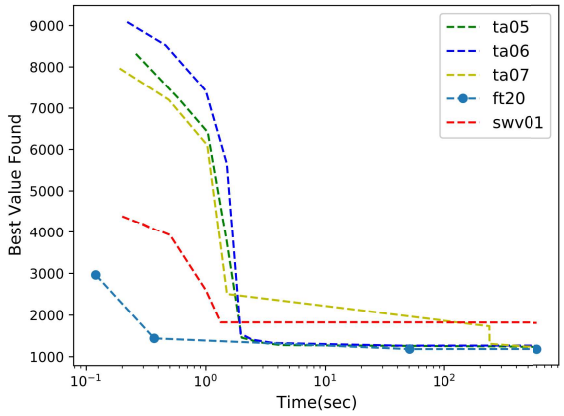

Figure 2: Relation between time and best value found for benchmark instances 'ta05, 'ta06', 'ta07', 'ft20', 'swv01' running the Disjunctive model in Z3 during 3600 seconds.

This method allows to find the solution in one call without need for iterating over potentially many of them.

Unlike νZ, that uses Z3 has a black box and it is built on top of it, OptiMathSAT has an inline architecture that calls the SMT solver only once and within it the SAT solver is then modified to handle the optimization. An insight about the optimizing algorithms running within the solver is given in Sebstiani's work [28]. Improved versions of the Branch&Bound algorithm are developed to exploit the features of MathSAT5 when dealing with linear algebra over Reals (LRA), integers (LIA) or a combination of both ($LRIA$).

4 Conclusions

In this paper we have compared three models for JSP suitable for optimizing SMT-solvers. We also compared the disjunctive model formulation in Z3 and OptiMathSAT with Gurobi, a state-of-the-art commercial MILP solver. On the benchmark examples Z3 outperforms Gurobi, and Gurobi outperforms OptiMathSAT. The results are very interesting because SMT-solvers are general purpose solvers that can easily

include additional constraints that are relevant in industrial applications

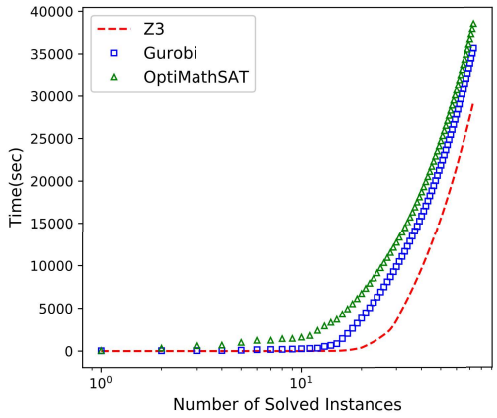

Figure 3: Performance comparison between SMT solvers Z3 and OptiMathSAT and MILP solver Gurobi over the benchmarck instances. The maximum time allowed for each is 600 seconds.

making them an attractive choice for real applications. There exist options such as dedicated algorithms based on CP and local search that provide better performance; nevertheless the results shown in this paper classify SMT-solvers, and Z3 in particular as a good alternative, especially since it is available under an open-source license (MIT License) and hence it is possible to use it for commercial purpose. Today, the licensing costs for commercial MILP-solvers can be substantial, something that will restrict the numbers of applications where scheduling can be motivated. Since optimization was added very recently to SMT-solvers and the research on SMT-solvers is a very active field with rapid progress it is reasonable to expect that the performance of optimizing SMT-solvers will continue to improve.

References

[1] A. S. Jain and S. Meeran, "Deterministic job-shop scheduling: Past, present and future", *European journal of operational research*, vol. 113, no. 2, pp. 390–434, 1999.

[2] Y. Nagata and I. Ono, "A guided local search with iterative ejections of bottleneck operations for the job shop scheduling problem", *Computers & Operations Research*, vol. 90, pp. 60–71, 2018.

[3] J. C. Beck, T. Feng, and J.-P. Watson, "Combining constraint programming and local search for job-shop scheduling", *INFORMS Journal on Computing*, vol. 23, no. 1, pp. 1–14, 2011.

[4] W.-Y. Ku and J. C. Beck, "Mixed integer programming models for job shop scheduling: A computational analysis", *Computers & Operations Research*, vol. 73, pp. 165–173, 2016.

[5] IBM. (2015). 12.6 user's manual, [Online]. Available: `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.2/ilog.odms.studio.help/pdf/usrcplex.pdf`.

[6] Gurobi. (2015). Gurobi optimizer reference manual, [Online]. Available: `http://www.gurobi.com`.

[7] A. Gleixner, L. Eifler, T. Gally, G. Gamrath, P. Gemander, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, F. Serrano, Y. Shinano, J. M. Viernickel, S. Vigerske, D. Weninger, J. T. Witt, and J. Witzig, "The SCIP optimization suite 5.0", eng, ZIB, Takustr.7, 14195 Berlin, Tech. Rep. 17-61, 2017.

[8] "Miplib 2010: Mixed integer programming library version 5", English (US), *Mathematical Programming Computation*, vol. 3, no. 2, pp. 103–163, 2011, ISSN: 1867-2949.

[9] J. F. Gonçalves, J. J. de Magalhães Mendes, and M. G. Resende, "A hybrid genetic algorithm for the job shop scheduling problem", *European journal of operational research*, vol. 167, no. 1, pp. 77–95, 2005.

[10] E. Balas and A. Vazacopoulos, "Guided local search with shifting bottleneck for job shop scheduling", *Management science*, vol. 44, no. 2, pp. 262–275, 1998.

[11] A. S. Jain and S. Meeran, "A state-of-the-art review of job-shop scheduling techniques", Technical report, Department of Applied Physics, Electronic and Mechanical Engineering, University of Dundee, Dundee, Scotland, Tech. Rep., 1998.

[12] J. Franco and J. Martin, "A history of satisfiability.", *Handbook of satisfiability*, vol. 185, pp. 3–74, 2009.

[13] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, *et al.*, "Satisfiability modulo theories", *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[14] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications", *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011, ISSN: 0001-0782.

[15] ——, "Z3: An efficient SMT solver", in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[16] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver.", in *TACAS*, N. Piterman and S. A. Smolka, Eds., ser. Lecture Notes in Computer Science, vol. 7795, Springer, 2013, pp. 93–107, ISBN: 978-3-642-36741-0.

[17] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4", in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11, Snowbird, UT: Springer-Verlag, 2011, pp. 171–177, ISBN: 978-3-642-22109-5.

[18] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νZ-an optimizing SMT solver", in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2015, pp. 194–199.

[19] R. Sebastiani and P. Trentin, "OptiMathSAT: A tool for optimization modulo theories", in *International Conference on Computer Aided Verification*, Springer, 2015, pp. 447–454.

[20] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling", *Math. Oper. Res.*, vol. 1, no. 2, pp. 117–129, May 1976, ISSN: 0364-765X.

[21] E. Taillard, "Benchmarks for basic scheduling problems", *European Journal of Operational Research*, vol. 64, no. 2, pp. 278–285, 1993.

[22] S. Lawrence, "Resouce constrained project scheduling: An experimental investigation of heuristic scheduling techniques (supplement)", *Graduate School of Industrial Administration, Carnegie-Mellon University*, 1984.

[23] D. Applegate and W. Cook, "A computational study of the job-shop scheduling problem", *ORSA Journal on computing*, vol. 3, no. 2, pp. 149–156, 1991.

[24] R. H. Storer, S. D. Wu, and R. Vaccari, "New search spaces for sequencing problems with application to job shop scheduling", *Management science*, vol. 38, no. 10, pp. 1495–1509, 1992.

[25] H. Fisher, "Probabilistic learning combinations of local job-shop scheduling rules", *Industrial scheduling*, pp. 225–251, 1963.

[26] C. Barrett, A. Stump, C. Tinelli, *et al.*, "The SMT-lib standard: Version 2.0", in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, vol. 13, 2010, p. 14.

[27] N. Bjørner and A.-D. Phan, "νZ-maximal satisfaction with Z3.", *SCSS*, vol. 30, pp. 1–9, 2014.

[28] R. Sebastiani and P. Trentin, "Pushing the envelope of optimization modulo theories with linear-arithmetic cost functions", in *Proc. Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'15*, ser. LNCS, vol. 9035, Springer, 2015.

PAPER B

SMT Solvers for Flexible Job-Shop Scheduling Problems: A Computational Analysis

Sabino Francesco Roselli, Kristofer Bengtsson, Knut Åkesson

Proceedings of 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE),
pp. 673–678, Sep. 2019.
 DOI: 10.1109/COASE.2019.8843025

The layout has been revised.

Abstract

In this paper we evaluate different formulations for solving the Flexible Job-shop Scheduling Problem (FJSP) to optimality. Heuristic methods resulting in sub-optimal solutions are traditionally used to solve this problem since it is a known NP-hard problem. Since industrial problems often have additional constraints that need to be considered during optimization, the heuristic methods need to be adapted to deal efficiently with the additional constraints. For this reason general-purpose solvers that are often used in industrial applications. There are different approaches to formulate FJSPs as Mixed-Integer Linear Programming problems (MILP) that can be solved using generic MILP-solvers. In recent years, satisfiability solvers, i.e. SAT- and SMT-solvers, have evolved within the formal verification community and shown to be able to efficiently solve large instances of well-known NP-hard problems. In our previous work we have shown that SMT-solvers extended with optimization techniques can be a competitive alternative to commercial MILP-solvers on traditional job-shop scheduling problems. In this work we have adapted three formulations used for formulating FJSPs for MILP solvers into SMT-formulations. The three formulations are used to solve benchmark FJSPs using the open-source Z3 SMT-solver. We show that the a formulation based on the Manne formulation adapted for SMT-formulations for FJSPs is a competitive alternative for solving large-scale FJSPs, and might be considered as a viable alternative for solving industrial problems.

1 INTRODUCTION

The Flexible Job-shop Scheduling Problem (FJSP) is an extension of the traditional Job-shop Scheduling Problem (JSP) and belongs to the class of NP-hard combinatorial optimization problems. The FJSP is characterized by a number of jobs, where each job is defined by a sequence of operations. Each operation can be executed by any one machine from a set of possible

machines with a predefined execution time. A machine can be used by multiple operations but only one operation is allowed to execute on a single machine at a time. The typical use of FJSP is to assign operations to machines such that the total time to complete all jobs is minimized, i.e. minimization of the makespan. In many academic benchmarking problems the challenge is typically to minimize the makespan, while in industrial problems, it is often more important to minimize tardiness with respect to given due dates for each job and/or operation. However, in this work we consider minimization of the makespan due to the availability of published benchmark problems for this class of problems. Since general-purpose solvers are used in this work it is straightforward to extend the formulations to handle due dates and/or additional constraints.

A comprehensive overview on the FJSP is presented by [1], where the authors present a variety of techniques ranging from exact methods to hybrid techniques that have been used to tackle the problem. According to the authors, a major effort has been spent on academic benchmark and generated instances rather than real industrial problems.

From a modelling point of view there are different paths to be chosen. Ever since the job-shop problems began to draw the attention of the academic community, three main model formulations has been introduced. These models date back to the 60s and were first proposed by Manne [2], Bowman [3] and Wagner [4] and has been the main models in field of operation scheduling.

During the following decades a large number of derived formulations and extensions have been presented to fit particular problems and special cases for both the JSP and the FJSP, but their main features coincide with these three models. A computational analysis of these formulations is shown in [5].

Regarding the FJSP, a large number of papers has been published where mainly Bowman and Manne's model, henceforth *time-index* and *disjunctive* models, are employed and there are divergent opinions about which exhibits the best performance. In our view, it is necessary to take the characteristics of specific solvers into account, since the performance of model formulations is dependent on the strategy the solver uses during the optimization. A technology that has emerged from the formal verification community and proved efficient in handling combinatorial problems is Satisfiability Modulo Theory (SMT) [6] solvers. This has motivated us to transform the three classical FJSP formulation into SMT-formulations and to use Z3 [7],[8], an optimizing

state-of-the-art SMT-solver for solving benchmark problems.

In our previous work [9], we evaluated different formulations of the JSP on benchmark problems using commercial state-of-the-art MILP solvers and optimizing SMT-solvers. We saw that the disjunctive formulation had in general the best performance irrespectivly if MILP or SMT-solvers are used. We also observed that the difference in performance of the IBM ILOG CPLEX and Gurobi optimizers are minor. Surprisingly, we found that the Z3 solver extended with optimization techniques can solve the JSP benchmark problems as efficiently or faster than the commerical MILP-solvers. These finding has motivated us to further investigate how FJSP can be transformed into SMT-optimization problems, and to evaluate the performance of different formulations on FJSP using Z3.

In the work [10], the disjunctive model is used to solve both the FJSP and an extension of it, including routing and process plan flexibility. Both the models and the algorithms presented gave credit to the hypothesis of the disjunctive model being an efficient solution to represent the FJSP. In the work [11] the time-index model is being adapted to handle large instances of the FJSP using a commercial MILP solver. In this work the author highlights the importance of having a tight upper-bound on the maximum time required to solve the problem before formulating the problem. This is important in order to reduce the problem formulation size and consequently the time necessary to solve the optimization problem. This is because in the time-index model the time is split into steps and an initial guess is required in order to declare all decision variables and constraints. Having a bad initial guess will lead to generate a model that is larger than necessary and therefore the optimization phase will take longer than necessary. Since upper-bounds can be computed using heuristic methods it is important with a tight upper bound for the comparison with the disjunctive model to be fair.

Based on the results we achieved in [9], it is interesting to see how the different model formulations will perform for the FJSP and at the same time we want to test the SMT solvers performance when dealing with FJSP.

The contributions in this paper are as follows. (i) Adapting the existing formulations of the disjunctive, time-index, and rank-based formulation of the FSJP to a model that can be solved by optimizing SMT-solvers. (ii) Evaluating the performance of the three different models on FJSP-benchmark using the Z3 SMT-solver.

In the next section, the problem is described in detail, and in section 3, we describe Z3 implementation for such models and in section 4 we discuss the results.

2 PROBLEM DESCRIPTION

The *Flexible Job-shop Scheduling Problem* (FJSP) is defined by a set of n jobs $J = \{j_i\}_{i=1}^{n}$, a set of k machines, $M = \{m_i\}_{i=1}^{k}$ and a set of operations, grouped in ordered subsets such that: $O^i = \langle O_f^i \dots O_j^i \dots O_l^i \rangle$ where O_f^i is the first operation and O_l^i is the last one and $O = \{O^i\}_{i=1}^{n}$. An operation is defined as a job being processed by a machine and since they belong to an ordered set, they define the sequence of machines to visit to complete a job; different jobs belonging to the same instance problem can have a different number of operations (a different number of machines to visit).

Moreover, for each operation, there is a set $M_i \subseteq M$ of machines to choose from for the execution. Also, each operation O_{ij} has a set of processing times $d_{mij} \forall m \in M_i$ because the execution time of an operation depends on which machine it is performed on. When finding a feasible solution to the FJSP, operations are assigned to machines so that all jobs are completed while the constraints are observed:

- an operation can only start if the machine belongs to its set of suitable machines;
- operations duration depends on the machine they are assigned to (*relative* duration constraint);
- operation using the same machine can not occur at the same time (*non-overlap* constraint);
- operations belonging to the same job must be executed according to the predefined sequence given in the instance (*precedence* constraint).

The goal is to find a schedule that satisfies the above mentioned constraints and at the same leads to the shortest possible makespan. Therefore the cost function Z must be set as larger than or equal to the completion time of the last operation of each job. minimizing the makespan is thus a *min-max* problem.

We now present the model formulations, (i) the *disjunctive* model, (ii) the *time-index* model and (iii) the *rank-based* model for the FJSP. These are extended models from [9] and based on [2], [3] and [4], respectively. To make them suitable for solving using an optimizing SMT-solver we have adapted them to an SMT formulation.

2.1 Disjunctive Model

In this model, time is described by integer variables that keep track of the start and completion time of each operation:

- s_{ij} is an integer variable and models the start time of operation i job j.
- e_{ij} is an integer variable and models the completion time of operation i job j.
- x_{mij} is a Boolean variables that becomes *True* if machine m is allocated to operation i of job j

minimize Z_{max} subject to

$$s_{ij} \geq 0 \qquad \forall j \in J, i \in O^j \tag{B.1}$$

$$Z_{max} \geq e_{o_l^j,j} \qquad \forall j \in J \tag{B.2}$$

$$s_{o_i^j,j} \geq e_{o_{i-1}^j,j} \qquad \forall j \in J, i = 2, \ldots, k \tag{B.3}$$

$$x_{mij} \rightarrow \bigwedge_{m' \in M_i - \{m\}} \neg(x_{m'ij}) \qquad \forall j \in J, i \in O^j \tag{B.4}$$

$$x_{mij} \rightarrow e_{ij} = s_{ij} + d_{mij} \qquad \forall j \in J, i \in O^j, m \in M \tag{B.5}$$

$$(x_{mij} \wedge x_{mi'j'}) \rightarrow (s_{ij} \geq e_{i'j'} \vee s_{i'j'} \geq e_{mj}) \qquad i \in O^j, i' \in O^{j'}, j, j' \in J, j \leq j', m \in M \tag{B.6}$$

In this model equation B.1 restricts the variables domain to the natural set, since it would not make sense to have negative time. Equation B.2 sets the objective function as larger than or equal to the completion time of each job's last operation. Equation B.3 sets the precedence constraint. Equation B.4 states that only one resource from the resource set of a specific operation can be assigned to it. Equation B.5 sets the relative-duration constraint of operations based on the resources they are allocated to. Finally, equation B.6 states the non-overlap constraint.

2.2 Rank-Based Model

In this model the focus is on the machine side: instead of calculating the sequence of machines that a job has to visit in order to be completed, the target is to figure the sequence of jobs that has to visit each machine to achieve the same goal. An additional complexity when comparing this formulation to the standard JSP is that a machine does not have to be visited exactly once by a job: it could be the case that it is not convenient for a job to visit it at all, or that all its operations must be executed on that machine to achieve a shorter makespan. Therefore, the total amount of operations P that a machine has to execute is not known beforehand and it can theoretically vary from *zero* to the number of all operations. Nonetheless, being the operations duration randomly distributed over a given interval, this value does not get far from the average number of operations per each job. Nonetheless, if the number of given positions is not large enough, the optimization might result in a value that is not the true optimal because a sub-optimal resource will be used. We thus have to make sure that the number of positions is large enough for the model to produce the true optimal: a trivial solution is that of having as many positions as the number of resource sets in which a machine is listed. Again, in instances where flexibility and operations duration is homogeneously (or equally) distributed, this value is relatively close to the number of operations per job, therefore it is likely to not affect the model performance.

The decision variables are defined as follows:

- x_{ijmp} is a Boolean variable that is *true* if operation i of job j is assigned to the p-th position of machine m;
- s_{mp} is an Integer variable representing the start time of the operation at the p-th position of machine m;

- e_{mp} is an Integer variable representing the completion time of the operation at the p-th position of machine m;
- r_{mij} is a Boolean variable that becomes *True* if operation i of job j is assigned to machine m;
- P is the number of positions available on a machine to allocate jobs.

minimize Z_{max} subject to

$$Z_{max} \geq e_{mp} \qquad \forall m \in M, p \in P \tag{B.7}$$

$$x_{ijmp} \rightarrow \bigwedge_{\substack{i' \in O^j - \{i\} \\ j' \in J - \{j\}}} x_{i'j'mp} \qquad \forall m \in M, p \in P \tag{B.8}$$

$$x_{ijmp} \rightarrow \bigwedge_{\substack{m' \in M - \{m\} \\ p' \in P - \{p\}}} x_{ijm'p'} \qquad \forall i \in O^j, j \in J \tag{B.9}$$

$$e_{mp_1} \leq s_{mp_2} \qquad \forall m \in M, p_1, p_2 \in P \tag{B.10}$$

$$\bigvee_{m \in M_i} r_{mij} \qquad \forall i \in O^j, j \in J \tag{B.11}$$

$$r_{mij} \rightarrow \bigvee_{p \in P} x_{ijmp} \qquad \forall i \in O^j, j \in J, m \in M \tag{B.12}$$

$$(x_{ijm_1p_1} \wedge x_{(i+1)jm_2p_2}) \rightarrow e_{m_1p_1} \leq s_{m_2p_2}$$
$$\forall i \in O^j - \{O^j_l\}, j \in J, m_1, m_2 \in M, p_1, p_2 \in P \tag{B.13}$$

$$x_{ijmp} \rightarrow e_{mp} = s_{mp} + t_{mij}$$
$$\forall i \in O^j, j \in J, m \in M, p \in P \tag{B.14}$$

Equation B.7 sets the objective function. Equation B.8 states that only one operation can be assigned to a certain position of a certain machine. Equation B.9 states that an operation can be assigned only once. Equation B.10 takes

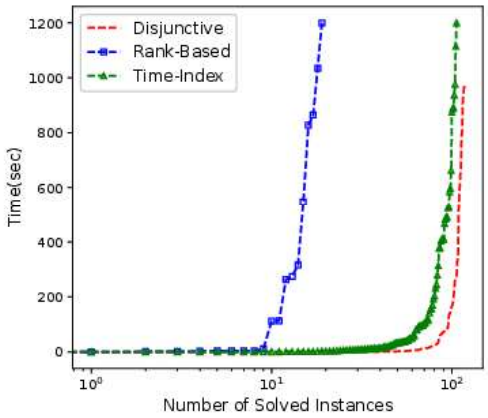

Figure 1: Performance comparison between Disjunctive, Time-Index and Rank-Based models over the generated instances. The maximum time allowed for each instance is 1200 seconds.

care of the precedence constraint for positions of the same machines. Equation B.11 provides the input about the resource set for each operation. Equation B.12 states that if an operation is assigned to a resource, it must be executed by such resource at some position. Equation B.13 takes care of the precedence constraint. Equation B.14 takes care of the relative duration.

2.3 Time-Index Model

In this model the execution time is split into steps, whose length is the minimum time unit. Unlike the Disjunctive model, where decision variables are integers, in this model they are either binary or Boolean (the choice is arbitrary and it can be influenced by the solver employed) and they represent events happening at a particular step.

The decision variables used in this model are:

- s_{ijt} is a Boolean variable that becomes *True* if operation i of job j starts at time-step t
- e_{ijt} is a Boolean variable that becomes *True* if operation i of job j ends at time-step t

- x_{mij} is a Boolean variables that becomes *True* if machine m is allocated to operation i of job j

minimize Z_{max} subject to

$$x_{mij} \rightarrow \bigwedge_{m' \in M_i - \{m\}} \neg(x_{m'ij}) \qquad \forall j \in J, i \in O^j \tag{B.15}$$

$$s_{ijt} \rightarrow \bigwedge_{t' \in T - \{t\}} \neg(s_{ijt'}) \qquad \forall i \in O^j, j \in J \tag{B.16}$$

$$e_{ijt} \rightarrow \bigwedge_{t' \in T - \{t\}} \neg(e_{ijt'}) \qquad \forall i \in O^j, j \in J \tag{B.17}$$

$$e_{o_l^j jt} \rightarrow Z_{max} \geq t \qquad \forall j \in J, i \in O^j, t \in T \tag{B.18}$$

$$x_{mij} \rightarrow \bigwedge_{t=0}^{T-d_{mij}} \neg(s_{ijt}) \qquad \forall j \in J, i \in O^j, m \in M \tag{B.19}$$

$$\bigwedge_{k=0}^{t} \neg e_{ijk} \rightarrow \bigwedge_{k=0}^{t} \neg s_{ij+1k} \qquad \forall j \in J, i \in O^j, t \in T \tag{B.20}$$

$$s_{ijt} \rightarrow \bigwedge_{k=o}^{t} \neg e_{ijk} \qquad \forall j \in J, i \in O^j, t \in T \tag{B.21}$$

$$(x_{mij} \wedge s_{ijt}) \rightarrow e_{ijt+d_{nij}} \qquad \forall j \in J, i \in O^j, m \in M, t \in T \tag{B.22}$$

$$(x_{mij} \wedge x_{mi'j'}) \rightarrow (\bigwedge_{k=0}^{t} \neg e_{ijk} \rightarrow \bigwedge_{k=0}^{t} \neg s_{i'j'k}) \vee (\bigwedge_{k=0}^{t} \neg e_{i'j'k} \rightarrow \bigwedge_{k=0}^{t} \neg s_{i'k}) \qquad j \in J, i \in O^j, t \in T \tag{B.23}$$

Equation B.15 states that only one machine can be assigned to an operation. Equations B.16 and B.17 state that an operation can only start at one time-step. Equation B.18 sets the objective function. Equation B.4 prevents

Table 1: Comparison of models implemented using Z3. The time showed in the table is the geometric mean calculated over all the instances belonging to the category they refer to. For each class the number of solved instances (out of the total number of instances belonging to such class) is given. The symbol '-' means that no instance has been solved within the time limit of 1200 seco.

	Disjunctive		**Time-Index**		**Rank-Based**	
Problems	**Time**	**Opt**	**Time**	**Opt**	**Time**	**Opt**
Generated Instances						
3x3x2	0.02	5/5	0.26	5/5	1.08	5/5
3x3x3	0.02	5/5	0.4	5/5	5.05	5/5
4x4x2	0.07	5/5	1.09	5/5	196	5/5
4x4x3	0.09	5/5	2.00	5/5	891	3/5
5x5x2	0.37	5/5	3.88	5/5	1164	1/5
5x5x3	0.56	5/5	7.28	5/5	-	0/5
5x5x4	0.56	5/5	10.27	5/5	-	0/5
5x5x5	0.55	5/5	12.79	5/5	-	0/5
6x6x2	1.34	5/5	17.97	5/5	-	0/5
6x6x3	2.48	5/5	25.71	5/5	-	0/5
6x6x4	2.95	5/5	32.79	5/5	-	0/5
6x6x5	2.67	5/5	41.38	5/5	-	0/5
7x7x2	6.71	5/5	106	5/5	-	0/5
7x7x3	10.88	5/5	87.18	5/5	-	0/5
7x7x4	18.42	5/5	106.51	5/5	-	0/5
7x7x5	18.52	5/5	127	5/5	-	0/5
8x8x2	23.92	5/5	283	5/5	-	0/5
8x8x3	100	5/5	542	4/5	-	0/5
8x8x4	96	5/5	431	5/5	-	0/5
8x8x5	103	5/5	406	5/5	-	0/5
9x9x2	83	5/5	925	3/5	-	0/5
9x9x3	399	5/5	1079	3/5	-	0/5
9x9x4	941	3/5	1126	1/5	-	0/5
9x9x5	625	5/5	-	0/5	-	0/5

operations to start at a time step when they could not be completed. Equation B.5 takes care of the precedence constraint. Equation B.6 states that an operation can not end before it starts. Equation B.22 sets the operations-relative duration based on the resource that is allocated to them. Equation B.23 takes care of the non-overlapping constraint. In order to create a model with such feature, we need to have a guess of the overall execution time, in other words, an upper bound T. A trivial upper bound is the sum of all operations duration (picking the resource that implies the shortest execution time) as if they were executed in a sequence (the worst case scenario).

The greater the upper bound, the longer it takes to create the model and therefore the slower the overall execution. After running some preliminary tests using this option to calculate the upper bound, we realized that it was affecting the performance too much and that would make unfair the comparison with the other models. Therefore we decided to use some simple heuristic to calculate a good-enough first guess: as a first step the FJSP is simplified by selecting a resource among the ones available for each operation; this will reduce the problem to a traditional JSP. The choice of the resource is arbitrary and can be done, for example, randomly, or picking the fastest resource. Then a greedy algorithm is run over the JSP and the solution is provided. This method is not iterative and it does not guarantee any margin about the solution goodness; yet it is fast and we noted empirically that its results are close enough to the optimum as to make the comparison fair.

3 Experiments

All tests whose results are presented in this paper were performed on an *Intel Core i7 6700K, 4.0 GHZ, 32GB RAM* running *Ubuntu-16.04*. The SMT solver employed for the benchmark evaluation is Z3-4.6.0 and the time limit set to evaluate each instance is 1200 seconds. The models have been created through the Python API for Z3 and then translated into the SMT-lib2 format [12] to be run directly and avoid possible delays due to the conversion while measuring the execution time. A heuristic [13] is employed to generate a first guess. With such tool it is possible to simplify the problem by choosing a random resource for each operation or the resource taking least time. For each instance both methods are used and the best approach giving the lowest makespan is chosen.

Table 2: Evaluation of the *disjunctive model* over *Fattahi, Barnes, Dauzere* and *Brandimarte* benchmark instances implemented using Z3. The value for each instance is either the true optimal, and the time employed to find it is shown, or the best incumbent found within the time-limit of 1200 seconds and then the relative error is shown. The symbol '-' means that incumbent was found within the time-limit.

Fattahi			Barnes			Dauzere			Brandimarte		
Instance	Value	Time/RE	Instance	Value	Time/RE	Instance	Value	Time/RE	Instance	Value	Time/RE
1	66	0.01	mt10c1	927	108	1a	3517	40%	mk01	40	22
2	107	0	mt10cc	908	44	2a	3571	60%	mk02	29	12%
3	221	0.01	mt10x	918	82	3a	3856	73%	mk03	280	37%
4	335	0.01	mt10xx	918	69	4a	3348	34%	mk04	63	5%
5	119	0.01	mt10xxx	918	77	5a	3880	76%	mk05	207	20%
6	320	0.01	mt10xy	905	47	6a	4380	102%	mk06	137	140%
7	407	0.01	mt10xyz	847	42	7a	4268	89%	mk07	218	57%
8	253	0.01	setb4c9	914	313	8a	6275	204%	mk08	704	35%
9	210	0.02	setb4cc	907	309	9a	9364	354%	mk09	851	177%
10	516	0.02	setb4x	925	943	10a	4109	83%	mk10	854	342%
11	468	0.08	setb4xx	925	1036	11a	7967	291%			
12	446	0.06	setb4xxx	925	955	12a	-	-			
13	446	0.17	setb4xy	910	469	13a	6184	176%			
14	554	0.33	setb4xyz	902	315	14a	-	-			
15	514	0.48	seti5cc	1147	1%	15a	-	-			
16	634	1.12	seti5x	1246	4%	16a	7054	216%			
17	879	6.89	seti5xx	1251	5%	17a	-	-			
18	884	25	seti5xxx	1228	3%	18a	-	-			
19	1064	-	seti5xy	1147	1%						
20	1244	-	seti5xyz	1160	3%						

An instance of the FJSP is a matrix of integers where each row represents a job. Within each row the first elements tells how many operations that particular job is composed of, since in some instances different jobs have a different number of operations (unlike the Traditional JSP). The following element tells the resource set size for the first operation of the job or, in other words, how many resources are suitable to execute that operation. Such number will be followed by a set of pairs as large as the number itself, here each pair tells about one of the eligible resources and its relative execution time (the time it will take to execute the operation if such resource is chosen). For instance, if a job has three operations, each having a set of three resources available to be executed, then the first element of the row is a *three* (number of operations), the second element is a *three* (resource set size for the first operation) and the following six elements are as follows: element number *one, three, five* are the indexes of resources to be chosen for the operation; elements *two, four, six* are the relative execution times. Then the next element will tell about the second operation-resource-set-size and will be followed by another tuple of pairs. The same is said about the third operation.

Based on the representation mentioned above, the instances employed for the evaluation are either downloaded from [1] or generated having random subsets of resources assigned to each operation and random-relative-execution-time related to each assignment, ranging from one to twenty seconds.

During the experimental phase both generated and benchmark instances are grouped as follows:

- Generated Instances: one hundred and twenty instances ranging from 2x2 to 9x9 in size and from 2 to 5 in flexibility (resource-set-size);
- Barnes [14]: twenty-one instances ranging from 10x11 to 15x18 in size and from 1 to 2 in flexibility;
- Brandimarte [15]: ten instances ranging from 10x6 to 20x15 in size and from 2 to 3 in flexibility;
- Dauzere [16]: eighteen instances ranging from 10x5 to 20x10 in size and from 1 to 7 in flexibility;

[1] `https://www.quintiq.com/optimization/flexible-job-shop-scheduling-problem-results.html`

- Hurink [17]: four sets of sixty-six instances based of JSP-benchmark-instances, each set with higher flexibility than the one before (from E to V)

- Fattahi [18]: eighteen instances ranging from 2x2 to 12x8 in size and from 2 to 3 in flexibility.

4 Results Discussion

In the first experimental phase, the three model formulations have been compared over generated instances. the results of such comparison are presented in table 1 and they show a huge gap between the performance of the models: both the disjunctive and the time-index outperform the rank-based model, which is only able to solve instances up to size 4x4, plus one of size 5x5. It provides no solution for larger instances within the time-limit. The time-index can easily handle small size instances and find the true optimal in less than one minute for instances of size 6x6. It is not able to solve to optimality all instances of size 8x8 and for the 9x9 instances with larger flexibility it can not find a solution at all within 1200 seconds. On the other hand, the disjunctive model is able to handle all generated instances but two, being faster than the time-index model by more than one order of magnitude. the gap becomes smaller on the instances of size 8x8 and 9x9, but the disjunctive model is still four to five times faster. Figure 1 shows the overall time required to each of the formulations to run all the generated instances, including the one for which an optimal was not found (in that case, the running time is equal to the time limit).

The second experimental phase consists of running the disjunctive model over a set of 332 benchmark instances, ranging from a size of 2x2 to 20x15 and a flexibility (machine set size) of 1 to 7 machines. Again the time-limit is set to 1200 seconds.

In table 2 the results of such phase are reported: for the instances whose optimal value was found whtin 1200 seconds, the running time is presented as well; for the once whose value was not found, the best incumbent found is presented, along with the relative error, calculated by comparing the best values taken from the literature.

The relative error has been calculated as follows:

$$RE = \frac{(best\ incumbent) - (best\ value)}{best\ value} * 100$$

In general, in some cases, even with relatively small-size instances, the optimal was hard to find, as for the instance *mk02* from the *Brandimarte* set. At the same time, for some larger instances, the optimal (or an incumbent that was very close to the optimal) has been found in a relatively short time. Therefore the instance size is not the only aspect that affects the problem complexity, although after a certain threshold, in terms of number of jobs, the problem becomes intractable anyway within the given time-limit.

A possible explanation for the existence of these outliers is that, there could exist several local sub-optimals that slow down the state-space search, while in some easy-to-solve instances there ist a relatively small number of variable combinations that lead to the optimal.

5 Conclusion

In this paper we have adapted three standard formulations to represent the FJSP to be suitable for SMT solvers. We have compared their performance while running on the state-of-the-art SMT solver Z3 and our results show that the *disjunctive* model outperforms both the *time-index* and the *rank-based* models. We have also tested the performance of Z3 over a large number of benchmark instances, proving that it is able to solve many of them to optimality within a considerably short time. These results confirm what we claimed in our previous work of Z3 being an attractive solution to tackle industrial problems, since it is available under an open-source licence, therefore usable for commercial purpose and because industrial applications are typically extensions of the models we presented and can be easily formulated in the SMT language.

References

[1] I. A. Chaudhry and A. A. Khan, "A research survey: Review of flexible job shop scheduling techniques", *International Transactions in Operational Research*, vol. 23, no. 3, pp. 551–591, 2016.

[2] A. S. Manne, "On the job-shop scheduling problem", *Operations Research*, vol. 8, no. 2, pp. 219–223, 1960.

[3] E. H. Bowman, "The schedule-sequencing problem", *Operations Research*, vol. 7, no. 5, pp. 621–624, 1959.

[4] H. M. Wagner, "An integer linear-programming model for machine scheduling", *Naval Research Logistics Quarterly*, vol. 6, no. 2, pp. 131–140, 1959.

[5] W.-Y. Ku and J. C. Beck, "Mixed integer programming models for job shop scheduling: A computational analysis", *Computers & Operations Research*, vol. 73, pp. 165–173, 2016.

[6] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications", *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[7] ——, "Z3: An efficient SMT solver", in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340.

[8] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νZ-an optimizing SMT solver", in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2015, pp. 194–199.

[9] S. Roselli, K. Bengtsson, and K. Åkesson, "SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation", in *Congress of CASE, the Portuguese Operational Research Society*, 2017.

[10] C. Özgüven, L. Özbakır, and Y. Yavuz, "Mathematical models for job-shop scheduling problems with routing and process plan flexibility", *Applied Mathematical Modelling*, vol. 34, no. 6, pp. 1539–1548, 2010.

[11] K. Thörnblad, A.-B. Strömberg, M. Patriksson, and T. Almgren, *A competitive iterative procedure using a time-indexed model for solving flexible job shop scheduling problems*. Department of Mathematical Sciences, Division of Mathematics, Chalmers University of Technology and University of Gothenburg, 2013.

[12] C. Barrett, A. Stump, C. Tinelli, *et al.*, "The SMT-lib standard: Version 2.0", in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, vol. 13, 2010, p. 14.

[13] S. Barrech and J. Poncy, *Flexible job shop scheduling problem*, `https://github.com/BlyxxFR/Flexible-Job-Shop-Scheduling-Problem`, 2018.

[14] J. Barnes and J. Chambers, "Flexible job shop scheduling by tabu search", *Graduate Program in Operations and Industrial Engineering, The University of Texas at Austin, Technical Report Series, ORP96-09*, 1996.

[15] P. Brandimarte, "Routing and scheduling in a flexible job shop by tabu search", *Annals of Operations research*, vol. 41, no. 3, pp. 157–183, 1993.

[16] S. Dauzère-Pérès and J. Paulli, "An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search", *Annals of Operations Research*, vol. 70, pp. 281–306, 1997.

[17] J. Hurink, B. Jurisch, and M. Thole, "Tabu search for the job-shop scheduling problem with multi-purpose machines", *Operations-Research-Spektrum*, vol. 15, no. 4, pp. 205–215, 1994.

[18] P. Fattahi, M. S. Mehrabad, and F. Jolai, "Mathematical modeling and heuristic approaches to flexible job shop scheduling problems", *Journal of intelligent manufacturing*, vol. 18, no. 3, pp. 331–342, 2007.

PAPER C

Compact Representation of Time-Index Job Shop Problems Using a Bit-Vector Formulation

Sabino Francesco Roselli, Kristofer Bengtsson, Knut Åkesson

Published in 2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)

The layout has been revised.

Abstract

The Job Shop Scheduling Problem (JSP) is a combinatorial optimization problem where jobs visit single-capacity machines while minimizing a cost function, typically the makespan. The problem can be extended to fit typical industrial scenarios such as flexible assembly shop floors or for coordinating fleets of automated vehicles. General purpose optimizers can handle extended versions of the problem that typically arise in industrial problems. Mixed Integer Linear Programming (MILP) solvers and recently optimizing Satisfiability Modulo Theory (SMT) solvers can be used as general solvers for JSP problems. There exist different formulations of JSP problems, among them the time-index (TI) model. The TI offers the advantage of providing strong lower bounds, though its drawback is the model size.

In this paper we present a new formulation of the TI model suitable for optimizing SMT-solvers that support bit-vector theories. The new formulation is significantly more compact than the standard TI formulation and is thus reducing one of the major issues with the TI model.

We benchmark two different optimizing SMT solvers supporting bit-vector theories, comparing the standard formulation of the TI to the new formulation on a set of benchmark instances. The computational analysis shows that the new formulation outperforms the standard one, being up to twice faster and regardless of the solver employed; moreover the model generated with the new formulation is considerably smaller than with the standard formulation.

1 INTRODUCTION

The Job Shop Problem (JSP) is the assignment of jobs to resources, where each job is a sequence of operations, such that the makespan is minimized. Resources have single capacity, thus two operations cannot use the same resource at the same time. The problem is NP-hard and has been studied since

the 60s, yet it is often emerging when developing automation systems and because of its complexity, it cannot be solved to optimality for large problem instances.

A thorough study on the subject is presented in [1], where the authors point out that, many different techniques have been applied to solve the problem: both approximation and exact methods. Whether the target is the true optimal or an approximate solution, it is important to develop general techniques that would be able to handle variations of the problem, since real-world scenarios typically involve additional requirements and as soon as a new constraint is added, tailor-made algorithms might no longer be feasible.

Thus general purpose algorithms like Mixed Integer Linear Program (MILP) solvers are often used in industrial applications since they can handle additional constraints without changing the main optimization algorithm. Note that new constraints could have a significant impact on the performance, but they will not disrupt the solution method.

MILP solvers can be used to provide approximate solutions by terminating the optimization procedure when the best incumbent (current best solution) is within a predefined gap from a lower bound to the problem, or after a time limit.

Another general purpose approach that can offer similar flexibility is optimizing Satisfiability Modulo Theory (SMT) solvers [2], [3]. SMT solvers support different theories, from reals, linear arithmetics, arrays to bit-vectors; these theories allow for a flexible modelling environment to easily instantiate quite complicated constraints.

In our previous works, [4] and [5], we showed that SMT solvers are a viable option to deal with the JSP both in its standard and flexible variants, since they could handle medium-large size problems (instances counting up to 15 concurrent jobs and 10 machines) in a relatively short time (time limit set to 1200 seconds). Our analysis showed that the SMT solver Z3 [6] outperformed a state-of-the-art MILP solver in terms of running time, by more than one order of magnitude for large instances.

We also implemented three model formulations of the JSP for SMT and benchmarked them over a set of instances generated according to predefined rules. One of the models we compared is known as the time-index model (TI) and was first presented by [7]; it is based on boolean (or binary) variables to represent the possibility for each job on each machine (operation) to start at

a given time-step. Time is discretized and for each operation, one variable will evaluate to True to indicate at which step such operation is starting.

Though our comparison proved the *disjunctive* model [8] to have the best performance, the TI is still widely employed to tackle the problem, in academia, [9] and [10] as well as in industrial applications [11]. This is due to the strong lower bounds that LP relaxations of the model provide [12], since they can be used to implement branch and bound or list-scheduling algorithms. The downside of the TI is its size: the number of constraints required to build the model becomes prohibitive even for relatively small instances, making the model-generation a bottleneck.

In [13], different optimization problems are tackled using both boolean (BL) and bit-vector (BV) models and, while in general a bit-vector formulation results in a much more compact representation, for some problems, it also leads to increase in performance by more than one order of magnitude.

Given these premises, it is reasonable to assume that a bit-vector based formulation could provide a more compact model and outperform the standard TI model based on boolean (or binary) variables.

The contributions of this paper are: (i) comparison of a new formulation of the TI model, based on bit-vectors, with the standard formulation over a set of benchmark and generated instances; the comparison is carried out both on the model generation phase (in terms of model size) and the running time to solve such models. (ii) Benchmark of two different state-of-the-art SMT solvers, namely Z3 and OptiMathSAT [14], chosen because they have shown consistently good performance in the latest SMT competitions and they come with a built-in optimizing tool.

In the next section, a formal mathematical description of the problem is given; in sections 3 and 4 the two model formulations are presented; in section 5 a complexity analysis over the two models is given; in section 6, the experimental methods are described and in the following section, results are discussed. Finally, conclusions are drawn in section 7.

2 Problem Formulation

The JSP problem consists of a set of n jobs $J = \{j_i\}_{i=1}^{n}$, where each job has its own processing order through a set of k machines, $M = \{m_i\}_{i=1}^{k}$. Also, let d_{mj} model the duration of the execution of the same operation. Operations

are defined as the execution of a job on a certain machine and, as each job has to visit each machine, the total number of operations in the problem is nk. Each job will go through all machines sequentially. Let o_i^j model the index of the machine to be used for job j executing operation i in sequence. The index of the machines for each step in the job sequence is thus given by $(o_1^j, \ldots, o_i^j, \ldots, o_k^j)$.

Finding a solution to the job-shop scheduling problem means to assign operations to machines so that all jobs are completed. The constraints in this kind of problem are two:

- as there exist a sequence of operations for each job, operations belonging to the same job must be executed in the right order;
- operations requiring the same machine and belonging to different jobs cannot overlap in time.

Given these two constraints, the target is to find a feasible schedule such that the overall makespan is minimized. Other variants of the problem involve different cost functions such as tardiness or lateness, but in order to compare our results to the optimal values available in the literature we followed the standard of the makespan.

We are now to present the TI model formulation, first in its standard form and then implemented using BV. An important feature of TI models is that they require an upper bound of the makespan. Since time is discretized, the upper bound is necessary to define a sufficient number of variables to describe the behaviour of the system at each time-step. If the upper bound is not large enough, the model will yield an infeasible result. On the other hand, the larger the upper bound, the more variables and constraints are needed. A trivial upper bound is the sum of all operation durations but tighter bounds can be calculated quickly using heuristic algorithms. However, computing tight upperbounds is beyond the scope of this paper. In the following we will assume that H is a given upper bound for a problem instance.

3 Standard Time-Index Model

In this model the execution time is split into steps, whose length is the minimum time-step. For instance, if the duration of an operation is n time-steps, n steps will be taken since it starts and until it is completed. The time-steps

will be $T = \{0, \ldots, H\}$. Let d_{mj} be a natural number that models the number of steps it takes for machine m to execute job j. The decision variables are $T_{\max}$ and s_{mjt}, where $T_{\max}$ is an integer variable and s_{mjt} are boolean variables that evaluate to true if job j starts on machine m at time t. The model formulation for minimizing the makespan is given by:

minimize $T_{\max}$ subject to

$$\bigvee_{t=0}^{H} s_{mjt} \qquad \forall m \in M, j \in J \tag{C.1}$$

$$s_{mjt} \rightarrow \bigwedge_{t' \in T \setminus \{t\},} \neg s_{mjt'} \qquad \forall m \in M, j \in J, t \in T \tag{C.2}$$

$$s_{mjt} \rightarrow \bigwedge_{t'=t}^{t+d_{mj}} \neg s_{mj't'} \qquad \forall j, j' \in J, j \leq j', m \in M, t \in T \setminus \{0\} \tag{C.3}$$

$$s_{mjt} \rightarrow T_{\max} \geq t + d_{mj} \qquad \forall m \in M, j \in J, t \in T \setminus \{0\} \tag{C.4}$$

$$x_{o_{i-1}^j jt} \rightarrow \bigwedge_{t'=0}^{t+d_{o_{i-1}^j j}} \neg x_{o_i^j jt'} \qquad \forall i = 2, \ldots, k, j \in J, t \in T \setminus \{0\} \tag{C.5}$$

Equations (C.1) and (C.2) ensure that start time for each operation occurs only once; equation (C.3) prevents other operations to start on a machine while it is already executing one; equation (C.4) defines the variable used in the objective function; equation (C.5) models precedence among the operations of a job: if the $(i-1)^{th}$ operation of job j starts at time-step t, the i^{th} operation of the same job cannot start before time-step $t + d_{o_{i-1}^j j}$.

4 Time-Index Model with Bit-Vectors

This model formulation is based on the time-index model presented in the previous subsection; instead of having boolean variables for each time-step and operation, there is a fixed sized bit-vector for each operation, whose size is given by H.

A bit-vector of size n is an element $\vec{b} = (b_{n-1}, \ldots, b_0) \in \mathbb{B}^n$. The index i maps the $(i)^{th}$ component of the vector, i.e. $b[i] = b_i$. Conversion from (to) a integer number is defined by $int : \mathbb{B}^n \to Z$ ($bv : Z \to \mathbb{B}^n$) with $Z = [-2^{n-1}, 2^{n-1}) \subset \mathbb{Z}$ and $int(\vec{b}) := -2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$ ($bv := int^{-1}$).

Constraints can be defined by using bit-vector operations as well as arithmetic and logic operations. Let $\vec{a}, \vec{b} \in \mathbb{B}^n$ be two-bit vectors. Then, the bit-vector operation $\circ \in \{\wedge, \vee, \cdots\}$ is defined by $\vec{a} \circ \vec{b} := (\vec{a}[0] \circ \vec{b}[0], \cdots, \vec{a}[n-1] \circ \vec{b}[n-1])$.

In the following model, some of the constraints are defined based on bit-vector manipulation formulas presented in [15].

The decision variables are defined as follows:

- $\vec{s}_{mj}$ is a bit-vector variable of size H that has exactly one bit set. The position of such bit defines the step at which job j starts on machine m;
- $\vec{w}_{mj}$ is a bit-vector variable of size H that has as many bits set as time-steps the job j takes to be executed on machine m. The rightmost bit in the trail corresponds to the bit set on the variable $\vec{s}_{mj}$;
- $\vec{e}_{mj}$ is a bit-vector variable of size H that has all bit sets from the time-step the operation is completed until the last position on the vector.
- $T_{\max}$ is a bit-vector variable of size H.

Also, $\underline{\vec{d}}_{mj}$ is a bit-vector constant of size H whose leftmost bits are set based on the duration of job j on machine m.

maximize $T_{\max}$ subject to

$$\vec{s}_{mj} \neq 0 \qquad \forall j \in J, m \in M \tag{C.6}$$

$$\vec{s}_{mj} \ \wedge \ (\vec{s}_{mj} - 1) = 0 \qquad \forall j \in J, m \in M \tag{C.7}$$

$$\vec{w}_{mj} = \bigvee_{i=0}^{d_{mj}} (\vec{s}_{mj} \gg i) \qquad \forall j \in J, m \in M \tag{C.8}$$

$$\vec{e}_{mj} = \neg\vec{w}_{mj} \ \wedge \ (\vec{w}_{mj} - 1) \qquad \forall j \in J, m \in M \tag{C.9}$$

$$\vec{s}_{jm} \ \wedge \ \vec{\underline{d}}_{mj} = 0 \qquad \forall j \in J, m \in M \tag{C.10}$$

$$\neg\vec{e}_{mo^j_{i-1}} \wedge \vec{s}_{mo^j_i} = 0 \qquad \forall j \in J, m \in M, i = 2 \dots k \tag{C.11}$$

$$\bigwedge_{\substack{j,j' \in J \\ j \leq j'}} (\vec{w}_{mj} \ \wedge \ \vec{w}_{mj'}) = 0 \qquad \forall m \in M \tag{C.12}$$

$$T_{\max} = \bigvee_{\substack{j \in J \\ m \in M}} \vec{e}_{mo^j_k} \tag{C.13}$$

Equation (C.6) makes sure that at least one bit is set and (C.7) makes sure that at most one bit is set for the bit-vector modeling the starting time for job j on machine m. Thus together they guarantee that each job will start exactly once on each machine. This is achieved by forcing the bit-vector to be a power of *two*; equations (C.8) and (C.9) are used to define the variables $\vec{w}_{mj}$ and $\vec{e}_{mj}$; equation (C.10) sets the latest start time for each operation: since a time horizon is given, operation cannot start too late, otherwise they will not be completed within the given number of steps; equation (C.11) sets the precedence constraint among operations belonging to the same job; equation (C.12) sets the objective function as the bit-wise operation $\vee$ among the $\vec{e}_{mj}$ operations occupying the last position in the sequence for each job: the larger the value of such vectors, the sooner the operation is completed, therefore the problem is a maximization one; finally equation (C.13) is taking care of non-overlapping constraint by setting the conjunction of the $\vec{w}_{mj}$ variables representing operations using the same machine equal to zero.

4.1 Example on the use of bit-vectors

In order to make the concept about fixed sized bit-vector variables and constants clearer, we provide a short example on how the values are assigned: Let the time horizon be set to 10 time-steps. This implies that the size of the vectors will be of 10 bits. Let's assume that for the operation of job j visiting machine m the starting time is set (by the solver) on the 5-th time-step and that such operation has a duration equal to three time-steps. Therefore, the variables for such operation will be set as follows:

$\vec{s}_{mj}$	0	0	0	0	1	0	0	0	0	0
$\vec{w}_{mj}$	0	0	0	0	1	1	1	0	0	0
$\vec{e}_{mj}$	0	0	0	0	0	0	0	1	1	1
$\vec{\underline{d}}_{mj}$	0	0	0	0	0	0	0	0	1	1

The variable $\vec{s}_{mj}$ has the 5th left most bit set, the variable $\vec{w}_{mj}$ has as many bits set as time-steps in the duration of the operation, starting from the 5th left most bit, the variable $\vec{e}_{mj}$ has all bits set form the time-step the operation is completed. Finally, the constant $\vec{\underline{d}}_{mj}$ has as many bits set as time-steps in the duration of the operation minus one, starting from the right most bit. This constant is required to set the constraint about the latest start of an operation. In the model, constraints (C.6) to (C.9) are needed to define the variables and constraint.

Here are given some examples of how constraints are enforced: In the following, the black x represent the possible assignments for the variables, while the red ones point out the forbidden assignments.

$\vec{s}_{mj}$	x	x	x	x	x	x	x	x	x	x
$\vec{\underline{d}}_{mj}$	0	0	0	0	0	0	0	0	1	1

Constraint (C.10) prevents an operation from starting too late by imposing the bit-wise operation $\wedge$ equal to zero between the variable $\vec{s}_{mj}$ and its duration constant $\vec{\underline{d}}_{mj}$.

$\vec{e}_{mo^j_{i-1}}$	0	0	0	0	0	0	0	1	1	1
$\neg\vec{e}_{mo^j_{i-1}}$	1	1	1	1	1	1	1	0	0	0
$\vec{s}_{mo^j_i}$	x	x	x	x	x	x	x	x	x	x

Constraint (C.11) sets the precedence constraints by allowing the following operation to start only after the previous one is completed. Assuming operation o^j_{i-1} is completed at time-step 7, operation o^j_i cannot start until that time-step.

$\vec{e}_{mo^j_k}$	0	0	0	0	0	0	0	1	1	1
$\vec{e}_{mo^{j'}_k}$	0	0	0	0	0	0	1	1	1	1
$\vec{e}_{mo^{j''}_k}$	0	0	0	0	1	1	1	1	1	1
$T_{\max}$	0	0	0	0	1	1	1	1	1	1

The objective function is set as the bit-disjunction of the variable $\vec{e}_{mj}$ for the last operation in each job. Maximizing $T_{\max}$ means having the latest operation completed as early as possible.

$\vec{w}_{mj}$	0	0	0	0	1	1	1	0	0	0
$\vec{w}_{mj'}$	x	x	x	x	x	x	x	x	x	x

Constraint (C.13) prevents operations employing the same machine to overlap, as shown in the example, where the red crosses define the forbidden execution times for the operation belonging to job *j'*

Finally, the result is presented as a bit-vector whose set bits represent the steps left before the time horizon is reached, since the last operation was completed. The integer value that represents the make-span to the JSP is given by the bit-vector $T_{\max}$ as

$$H - \text{number of bits set in } T_{\max}$$

Since the duration of an operation is positive we can conclude that not all operations could be finished at time 0, meaning that the left-most bit in T_{max} has to be 0. From how T_{max} is defined we know that it will have a sequence of zeros followed by a sequence of ones. Thus, with an increasing number of ones the signed/unsigned interpretation of T_{max} will result in a larger value. To minimize the make-span we will thus maximize T_{max}.

4.2 Bit-vector manipulation

In general, one must be careful when performing operations on bit-vectors since some of them will produce a different result, depending on whether the bit-vector is signed or not. The *right-shift* operation, for instance, comes in two different versions; other operators are unaffected by the interpretation: the bit-wise operators.

In this work, the actual value of the number represented by a bit-vector is not of interest, since we are only interested in the bit-patterns (the sequence of zeros and ones in a bit-vector) to represent time-steps: if the left-most bit of a bit-vector of size *four* is set, we are not interested in its value in decimals (it would be 8 for a signed and -8 for unsigned), it tells us that something is happening at time-step 0. Only in constraint (C.13) we are interested in the actual value of T_{max}, since this is the value that is maximized. For T_{max} it would make a difference to have signed or unsigned bit-vectors only if we were to set its leftmost bit. This, in turn, could only happen if it was possible to complete all jobs at time-step zero, which is by definition impossible, since operations have a duration larger than 0. Hence, the constraint is valid.

Also, the difference between signed and unsigned bit-vectors lies in the interpretation of bit-patterns and the tricks used to manipulate the bit-vectors are designed to set and unset bits regardless of the interpretation. In fact, they involve bit-wise operators such as $\wedge$, $\vee$ and $\neg$ with the exception of constraints (C.7) and (C.9): in both cases a subtraction is performed. However, subtracting a bit-vector means to add its negation plus *one*; since addition and negation both work independently of the interpretation, there is no risk of producing invalid results, as long as the operands involved in the subtraction are positive. This is always the case, since one of them is the value *one* and the other, w_{mj}, is inferred from s_{mj}, which is always positive because of constraints (C.6) and (C.7).

5 Models Size

The formulations presented in the previous sections, though similar in many aspects, lead to a significant difference in the model size. The reason for such difference lies in the way the time horizon, H, is handled by the two models: in the BV model the time horizon is used only to define the size of the bit-vector variables; therefore, equations (C.6)-(C.10) in the BV model

only generates nk constraints (n jobs, k machines), while equation (C.11) generates $nk(k-1)$ constraints, equation (C.12) generates k constraints and (C.13) generates one. Also, the length of the constraints, in terms of number of clauses for each constraint, is short: only one clause for constraints (C.6), (C.7), (C.9), (C.10) and (C.11). For the constraints expressed by equation (C.8), the number of clauses depends of the duration of job j visiting machine m, for equation (C.12), the length is ${}^{n^2}/_2$, and for equation (C.13), the length is nk.

On the other hand, in the BL model, there is one variable for each job, machine and time-step. Therefore constraints are dependent on the time horizon as well, i.e. equation (C.1) generates nk constraints, each of length H. Equation (C.2) generates nkH constraints each of length H, since for each machine m, job j and time-step t, it is necessary to iterate the $\wedge$ connective over all time-steps but t. Equation (C.3) generates the largest amount of constraints, since it iterates over any two jobs j and j', for each machine m and time-step t. Also, for each value of these indexes, the $\wedge$ connector has to be iterated for as many times as the duration of operation j on machine m, leading to ${}^{n^2}/_2 \cdot kH$ constraints each of length d_{mj}. Equation (C.4) generates nkH constraints of length one and, finally, equation (C.5) generates one constraint for each machine, job and time-step, and within each constraint an additional iteration of the $\wedge$ connective for as many times as the duration of operation j on machine m is required, leading to nkH constraints, each of length d_{mj}.

Given these premises, we can infer that:

BL model size Given a JSP, with n jobs and k machines with time horizon H, the number of variables for the BL model is nkH, the number of constraints is proportional to $O(n^2kH)$ and their length in number of clauses is $O(H)$.

BV model size Given a JSP with n jobs and k machines, the number of decision variables for the BV model is $3nk$, and the size of each variable is H bits, thus the total size is proportional to nkH. The number of constraints is $O(nk^2)$ and their length in number of clauses is $O(n^2)$.

Note that H is typically much larger than k and n. Therefore the number of constraints, as well as their length is expected to be significantly larger for the BL model. The favorable size of BV models compared to BL models is confirmed empirically in the following section.

Table 1: Comparison of models implemented using Z3 and OptiMathSAT. The time showed in the table is the geometric mean calculated over all the instances belonging to the category they refer to. For each class the number of solved instances (out of the total number of instances belonging to such class) is given. The symbol '-' means that no instance has been solved. The model size is also reported in Megabytes.

Problems	Model Size		Z3-BV		Z3-BL		Opti-BV		Opti-BL	
	BV	BL	Time	Opt	Time	Opt	Time	Opt	Time	Opt
3x3	0.009	0.243	0.13	5/5	0.13	5/5	0.25	5/5	32.3076	2/5
4x4	0.016	0.900	0.49	5/5	0.94	5/5	0.25	5/5	-	-
5x5	0.027	2.050	1.83	5/5	5.05	5/5	0.64	5/5	-	-
6x6	0.040	3.837	4.06	5/5	20.88	5/5	1.47	5/5	-	-
7x7	0.058	6.739	13.05	5/5	68.51	5/5	4.22	5/5	-	-
8x8	0.079	11.526	32.50	5/5	203.31	5/5	13.12	5/5	-	-
9x9	0.104	18.696	176.15	5/5	-	-	71.69	5/5	-	-
10x10	0.132	24.818	609.07	4/5	-	-	410.21	5/5	-	-
11x11	0.168	37.307	1117.60	1/5	-	-	740.85	3/5	-	-
12x12	0.205	48.927	-	-	-	-	-	-	-	-
13x13	0.249	69.687	-	-	-	-	-	-	-	-

6 Computational Analysis

We evaluate the properties of the two models by generating problem instances using the Taillard instance generator specification [16]. In total 55 problems are generated from size 3x3 to 13x13 with 5 problems of each size.

6.1 Experimental setup

The solvers whose performance were compared are Z3-4.8.7 and OptiMathSAT-1.5.1. The time limit is 1200 seconds. Solvers are run in their default setting. All the experiments were performed on an *Intel Core i7 6700K, 4.0 GHZ, 32GB RAM* running *Ubuntu-18.04 LTS*.

Since finding a good upper bound for the model is beyond the scope of this paper, and the optimum is known for all instances evaluated, we used as a value for H the optimum increased by 10%.

Since both Z3 and OptiMathSAT can read input in the SMT standard language [17], it has been possible to translate the models into such language and then run the solvers directly from the terminal, to avoid delays due to the API's use. Also, this allows to run the solvers on exactly the same models,

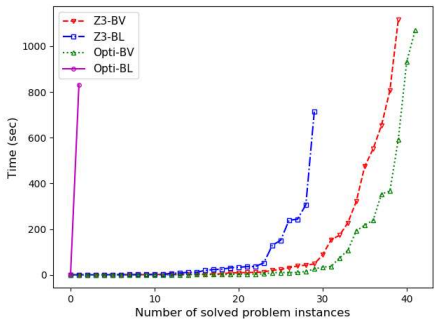

Figure 1: Performance comparison between bit-vector (BV) and boolean (BL) time-index models over the generated instances using Z3 and OptiMathSAT. The maximum time allowed for each instance is 1200 seconds.

and to keep track of the models size for comparison. The implementation of the scheduler is available at [18].

6.2 Experimental Results

Table 1 summarizes the results of the computational analysis: instances are sorted by size (5 instances for each class) and for each different combination of model and solver, the number of solved instances is reported as well as the average time to find the optimum of the solved ones. We decided to employ the geometric mean to reduce the effect of outliers. The average model size for both the BL and BV model is also shown for each class.

The evaluation of the BV model implemented with Z3 showed that all instances could be solved within the time-limit up to size 9x9, while only 4 of size 10x10 and 1 of size 11x11 could be solved to optimality; no larger instance is solved within 1200 seconds. The performance of BV implemented with OptiMathSAT is significantly better, being OptiMathSAT roughly twice as fast as Z3 and able to solve all instances of size 10x10 and 3 of size 11x11. When it comes to the BL model, Z3 outperforms OptiMathSAT by more than one order of magnitude, being able to solve instances up to size 8x8 in a relatively short time: less than a second for size 3x3 and 4x4, 5 seconds for size 5x5 and respectively 20, 70 and 200 for the remaining sizes. On the other hand, the BL model implemented with OptiMathSAT is only able to solve 2 instances

out of 55 (of size 3x3) and it still takes 30 seconds to do so.

When it comes to the model size, it turns out (as expected) that the BL model quickly scales up, going from 0.2 Megabytes for instances of size 3x3 to almost 70 MB for the larger ones. On the other hand, the BV model size is barely affected by the instance size, being still largely under 1MB for the larger instances. The time required to generated the model may be strictly dependent on the implementation, but it is still related to the model size, so the larger the model, the slower the generation time. With our implementation, the time to generate the BV model for the instances of size 13x13 was still below three seconds, while for the BL model it was around 600 seconds.

6.3 Results Discussion

The Computational Analysis proves the BV model to be faster than the BL model regardless of the solver it is implemented with, as shown in Figure 1; the best combination of solver-model is OptiMathSAT running the BV model, while the solver that showed the best performance when running the BL model was Z3. The reason for this behaviour lies not only in the efficiency of the underlying SAT engine within the SMT solver, but also in the way the particular theory the model belongs to [19]: some solvers simply *bit-blast* the model, meaning that they generate boolean variables to be able to handle it with Propositional Logic (eager approach), while others combine the SAT solver with a *Theorem Prover* to use specific Procedures to check feasibility (lazy approach). The latter method can, in some cases, save a lot of computational effort, increasing the efficiency of the overall approach. So, depending on the strategy employed by each solver (eager or lazy) and the efficiency of the procedure for the specific theory, one solver can be very good at solving one model, while being rather slow for another.

Another interesting result is the model size: the BV model proved to be extremely compact, increasing only linearly with the instance size (and with a very low coefficient), while the BL model's increase is roughly quadratic (see Figure 2). For other problem formulations, the model generation can usually be neglected, since it requires a very short time, compared to the solving time itself. But with time discretization, the number of variables is much higher, and the number of constraints generated out of them is even higher.

7 Conclusion

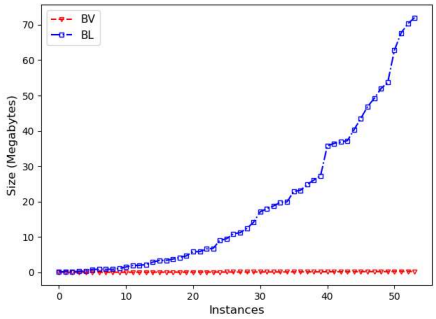

Figure 2: Comparison between model sizes of the bit-vector and The boolean implementations of the time-index model, over the generated instances.

In this paper we have presented a new approach to implement the time-index model for the Job Shop Problem using bit-vectors. We benchmarked two state-of-the-art SMT solvers over a set of instances generated according to a standard method and the resulting model turned out to be more efficient than the existing one based on boolean variables, regardless of the solver employed. We also showed that the more compact formulation leads to a drastic decrease in the model size, which in turn affects the model generation time (a bottleneck for the time-index model).

Since the time-index model is widely employed both in industry and academia to deal with the JSP, this contribution can improve the performance in many applications.

References

[1] I. A. Chaudhry and A. A. Khan, "A research survey: Review of flexible job shop scheduling techniques", *International Transactions in Operational Research*, vol. 23, no. 3, pp. 551–591, 2016.

[2] R. Sebastiani and P. Trentin, "Pushing the envelope of optimization modulo theories with linear-arithmetic cost functions", in *Proc. Int.*

Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'15, ser. LNCS, vol. 9035, Springer, 2015.

[3] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νZ-an optimizing SMT solver", in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2015, pp. 194–199.

[4] S. F. Roselli, K. Bengtsson, and K. Åkesson, "SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation", in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, IEEE, 2018, pp. 547–552.

[5] ——, "SMT solvers for flexible job-shop scheduling problems: A computational analysis", in *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, IEEE, 2019.

[6] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver", in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[7] E. H. Bowman, "The schedule-sequencing problem", *Operations Research*, vol. 7, no. 5, pp. 621–624, 1959.

[8] A. S. Manne, "On the job-shop scheduling problem", *Operations Research*, vol. 8, no. 2, pp. 219–223, 1960.

[9] E. G. Birgin, P. Feofiloff, C. G. Fernandes, E. L. De Melo, M. T. Oshiro, and D. P. Ronconi, "A MILP model for an extended version of the flexible job shop problem", *Optimization Letters*, vol. 8, no. 4, pp. 1417–1431, 2014.

[10] L. Jin, Q. Tang, C. Zhang, X. Shao, and G. Tian, "More MILP models for integrated process planning and scheduling", *International Journal of Production Research*, vol. 54, no. 14, pp. 4387–4402, 2016.

[11] K. Thörnblad, A.-B. Strömberg, M. Patriksson, and T. Almgren, "Scheduling optimisation of a real flexible job shop including fixture availability and preventive maintenance", *European Journal of Industrial Engineering*, vol. 9, no. 1, pp. 126–145, 2015.

[12] J. Van den Akker, C. A. Hurkens, and M. W. Savelsbergh, "Time-indexed formulations for machine scheduling problems: Column generation", *INFORMS Journal on Computing*, vol. 12, no. 2, pp. 111–124, 2000.

[13] R. Wille, D. Große, M. Soeken, and R. Drechsler, "Using higher levels of abstraction for solving optimization problems by boolean satisfiability", in *2008 IEEE Computer Society Annual Symposium on VLSI*, IEEE, 2008, pp. 411–416.

[14] R. Sebastiani and P. Trentin, "OptiMathSAT: A tool for optimization modulo theories", in *International Conference on Computer Aided Verification*, Springer, 2015, pp. 447–454.

[15] H. S. Warren, *Hacker's delight*. Pearson Education, 2013.

[16] E. Taillard, "Benchmarks for basic scheduling problems", *European Journal of Operational Research*, vol. 64, no. 2, pp. 278–285, 1993.

[17] C. Barrett, A. Stump, C. Tinelli, *et al.*, "The SMT-lib standard: Version 2.0", in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, vol. 13, 2010, p. 14.

[18] *Benchmark code*, `https://github.com/sabinoroselli/Job_Shop.git`, Accessed: 2020-07-06.

[19] D. Kroening and O. Strichman, *Decision procedures - An Algorithmic Point of View*. Springer, 2016.

PAPER D

On the Use of Equivalence Classes for Optimal and Sub-Optimal Bin Packing and Bin Covering

Sabino Francesco Roselli, Fredrik Hagebring, Sarmad Riazi, Martin Fabian, Knut Åkesson

Conditionally accepted to 2020 IEEE Transactions on Automation Science and Engineering (TASE)

The layout has been revised.

Abstract

Bin packing and *bin covering* are important optimization problems in many industrial fields, such as packaging, recycling, and food processing. The problem concerns a set of *items*, each with its own *value*, that are to be sorted into *bins* in such a way that the total value of each bin, as measured by the sum of its item values, is not above (for packing) or below (for covering) a given *target* value. The optimization problem concerns minimizing, for bin packing, or maximizing, for bin covering, the number of bins. This is a combinatorial NP-hard problem, for which true optimal solutions can only be calculated in specific cases, such as when restricted to a small number of items. To get around this problem, many sub-optimal approaches exist. This paper describes formulations of the bin packing and covering problems that allows to find the true optimum for instances counting hundreds of items using general purpose MILP-solvers. Also presented are sub-optimal solutions that come within less than 10% of the optimum, while taking significantly less time to calculate, even ten to 100 times faster, depending on the required accuracy.

Note to Practitioners

A typical case for bin covering is in food processing where food items are automatically sorted into trays of similar weight, so that the overweight is minimized. Another application is in recycling, where items like batteries should be put in crates of similar weight, so that the crates do not exceed a target weight, due to later manual handling, but at the same time we want as few crates as possible. This is a bin packing problem. On an industrial scale these tasks are fully automated. Though modern software tool's efficiency to solve bin sorting problems have increased significantly in later years, the problems are inherently tough in the sense that the solution time grows exponentially with the number of items. This limits the problem sizes that can be solved to optimality within reasonable time. Therefore, much research has focused on heuristic rules that give reasonable solving times while not giving

the true optimal number of bins. However, in many cases the true optimal solution is preferable, and sometimes even necessary, so this is an industrially interesting problem. This paper describes an approach to solve the bin packing and covering problems to the true optimum that increases the limit of the number of items that can typically be handled. This is done by observing that items of same value need not be distinguished. Instead, we can formulate packing/covering problems over item values rather than individual items, and sort integer numbers of these values into bins, which allows to solve to optimum for more than 500 hundred items in reasonable time. In addition, by redefining what we mean by *same value*, we can consider more items to have the same value and achieve even better computational efficiency.

1 Introduction

The (one-dimensional) *bin sorting* problem concerns sorting items with given values into *bins* such that the value of a bin, counted as the sum of its included values, conforms to a specified *target* value, while at the same time optimizing the total number of bins. Two (dual) variants of this problem exist, *bin packing* [1] where the bin values cannot go over, and *bin covering* [2] where the bin values cannot go under, respectively, the target value.

The problems are NP-hard [1] combinatorial optimization problems, meaning that there is no general algorithm that can solve either problem for an arbitrary number of items in reasonable time.

Due to this, different heuristic algorithms to provide sub-optimal solutions while guaranteeing a bound from the optimum and having polynomial complexity have been a long-time active field of research. Recent surveys of related problems are presented in [1] for approximative algorithms and in [3] for exact algorithms.

Recent work on bin sorting is based on branch-and-price based algorithms, see e.g. [4]. Pseudo-polynomial formulations, [5]–[7], allow a more compact formulation and avoid the complexity introduced in the implementation of branch-and-price based algorithms. Of interest in some practical applications is also temporal extensions, [8], where the capacity of a bin must be consumed within a given time window.

In our previous work on the subject [9], we presented a model to solve the bin covering problem to optimality by means of *Mixed Integer Linear*

Programming (MILP), and we showed through a computational analysis of more than 800 problem instances that a MILP solver is able to handle quickly large sized instances. In this work we extend the analysis to also bin packing, and we provide mathematical proof of the validity of our claims on top of which we implemented the above-mentioned formulation. We also present a sub-optimal approach based on a simplification of the original instances that exploits the feature of our new formulation and guarantees close to optimal results.

In industrial problems there are often additional constraints that are not included in text book formulations of the sorting problems. An example from an industrial application: for bin covering problems it might be allowed to go a few percent below the target weight for certain bins as long as the average bin value is on or above target. Algorithms that are tailor made for textbook formulations of the sorting problems might have problems to generalize to such modified versions of the problem. In industrial applications general purpose solvers are thus often preferred due to their ability generalize to new problem formulations. So, though we in this paper treat only the text book versions of bin sorting, we do so by focusing on formulations that can be given as input to general-purpose MILP solvers, and we evaluate the efficiency of these formulations.

First is presented the standard formulation that can be found in most text books. This formulation was first introduced by [10] and is typically useful only for a small number of items. Then is given the "subset" formulation, which removes the identities of the bins and thus allows to solve for a much larger number of items and bins. Thirdly is given the "equivalence class" formulation, which further removes the identities between items of same values, and hence allows to solve for even larger numbers of items and bins. As explained later on in the paper, the idea of *equivalence classes* leads to define combinations of items that meet the target requirement (i.e. the cumulative value of such items is below the target for the bin packing and above it for the bin covering); such combinations are given the name of *packages* in this paper and, to the best of our knowledge, they have first been introduced by [11] to implement a specific purpose algorithm to solve bin covering problem, and then used in [12] to implement a branch and price algorithm, while we use it to develop a linear-integer model for a general purpose solver. Also, we improve the concept by defining a subset of packages among which bins can

be selected that still leads to the optimal solution. In other words, we do not need to enumerate all possible combinations of items in order to find the optimal solution, but only a rather small portion of it.

The contributions to this paper are: (i) improving the concepts of equivalence classes by introducing the notion of *skinny* and *fit* bins for the bin covering and the bin packing respectively; proving that the the optimal solution of an instance of the bin packing (covering) is only composed by *fit* (*skinny*) bins; (iii) develop a heuristic method for both the bin covering and bin packing problem, based on equivalence classes, that significantly reduces the computation time while still guaranteeing close to optimum results; (iiii) evaluating the method against other algorithms for bin sorting problems by running it over different sets of benchmark instances.

In the next section the general bin sorting problem is described, giving three different MILP formulations, two of which exploit the fact that prospective bins can be pre-calculated to make the MILP solver's job easier. Then Section 3 presents how the combinatorial explosion can be further mitigated by restricting the number of pre-calculated bins, while still guaranteeing optimal solutions. Section 4 then describes how the number of pre-calculated bins can be made even smaller, but then not guaranteeing optimal solutions. The experimental results of Section 5 shows the computational benefits of both the optimal formulations, and the sub-optimal formulation that comes within less than 10% of the optimum, while achieving a significant reduction in computation time. The paper is concluded in Section 6.

2 Bin Sorting

Bin sorting is a generic term for the two (dual) problems of bin *packing* and bin *covering*. The bin sorting problem concerns a set of *items* $V = \{v_1, v_2, \ldots, v_n\}$, each with a *value* so that there can be defined an ordering between the items, such that $v_1 \geq v_2 \geq \cdots \geq v_n$. For notational simplicity, except for in a few places, the distinction between an item and its value will not be made; note though that items are unique, whereas two items can have the same values. Given a subset $V' \subseteq V$ we denote its minimum and maximum values as $\min(V')$ and $\max(V')$, respectively.

A *bin* $b_j \subseteq V$ is a subset of V. For a bin b_j we can define its value B_j as the sum of the item values it contains, that is, $B_j = \sum_{v_i \in b_j} v_i$.

The bin sorting problem can now be defined as a tuple $\langle V, t, \bowtie \rangle$, where t is a target value that defines a bound on the bin values, and $\bowtie$ is $\leq$ for bin packing, and $\geq$ for bin covering. The problem is now to find an optimal solution $B_{opt} = \{b_1, b_2, \ldots, b_m\}$, which is a partition of V with the minimum, for packing, or maximum, for covering, number m of bins, such that $\forall b_j \in B_{opt},\ B_j \bowtie t$. It is assumed that $\sum_{v_i \in V} v_i > t$, and $\forall v_i \in V\ v_i < t$.

Since we in large parts of the paper simultaneously deal with both covering and packing, we have introduced a non-standard notation of our own (such as "target" t instead of "capacity" c). This so, since the communities dealing with the respective problems do not always agree on the notation. Furthermore, the term "bin covering" is sometimes used to denote a different problem, where the number of bins is fixed and the problem is maximizing the number of packed items while not exceeding the target value for any bin [13].

2.1 The Standard Formulation

One way to formulate the bin sorting problem is as a *mixed linear integer programming* (MILP) problem, where the decision variables represent bins and the allocation of items to the bins. Let b_j $(j = 1, \ldots, n)$ be 0-1-variables representing whether a certain bin is used $(b_j = 1)$ or not $(b_j = 0)$, and let x_{ij} $(i, j = 1, \ldots, n)$ be 0-1-variables representing whether the value v_i is assigned to the j'th bin $(x_{ij} = 1)$, or not $(x_{ij} = 0)$. The MILP problem can then be formulated as:

$$\min / \max \sum_{j=1}^{n} b_j \text{ subject to} \tag{D.1}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i = 1, \ldots, n \tag{D.2}$$

$$\sum_{i=1}^{n} x_{ij} \cdot v_i \bowtie b_j \cdot t \qquad \forall j = 1, \ldots, n \tag{D.3}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j = 1, \ldots, n \tag{D.4}$$

$$b_j \in \{0, 1\} \qquad \forall j = 1, \ldots, n \tag{D.5}$$

The objective function (D.1) is the sum over all the variables representing whether a certain bin is used or not, and since these are binary 0-1-variables,

the sum is the number of used bins; this sum is to be minimized for bin packing and maximized for bin covering. Constraint (D.2) guarantees that each item is assigned to exactly one bin. Constraint (D.3) guarantees that the value of each used bin is on or below ($\bowtie$ is $\leq$) the target value t for bin packing, and on or above ($\bowtie$ is $\geq$) the t for bin covering. Constraints (D.4) and (D.5) define the domains of the decision variables.

For bin covering, Constraint (D.2) can actually be relaxed to ≤ 1, since not all values are necessarily placed in some bin. However, such *surplus* values (see below) cannot constitute a bin on their own, and since there is no upper bound on the bins, the surplus values may be put on any bin without altering the optimal solution. In fact, with a rigorous definition of the surplus values, we can always remove the surplus values from the optimal solution, and the surplus-free solution will still be optimal.

2.2 The Subset Formulation

A less trivial approach to formulate the bin sorting problem as a MILP problem is to enumerate the prospective bins by sort the items into *packages* that fulfill the target constraint, and then formulate a problem of choosing the smallest or largest number of such packages. If we generate *all possible* such packages, the MILP-solver will have freedom enough to find the optimal number of packages.

Let $p_j \subseteq V$ be a package such that $\sum_{v_i \in p_j} v_i \bowtie t$. Note that contrary to bins, different packages may share items, that is, for some packages p_i and p_j with $i \neq j$ it may hold that $p_i \cap p_j \neq \varnothing$. Let $P_i = \{p_j | v_i \in p_j\}$ be the set of all packages that include the item v_i. We call the elements of P_i *overlapping*, and there are n such sets.

Given the set of k generated packages, and with a slight abuse of notation we use p_j to denote a 0-1-variable representing whether the package p_j is used ($p_j = 1$) or not ($p_j = 0$), we can formulate the MILP problem as:

$$\min / \max \sum_{j=1}^{k} p_j \text{ subject to} \tag{D.6}$$

$$\sum_{p_j \in P_i} p_j = 1 \qquad \forall i = 1, \ldots, n \tag{D.7}$$

$$p_j \in \{0, 1\} \qquad \forall j = 1, \ldots, k \tag{D.8}$$

Similarly to (D.1), the objective function (D.6) sums over all the variables representing whether a package is used or not, and since these are 0-1-variables, the sum is the number of used packages; this sum is to be minimized, or maximized, for bin packing and bin covering, respectively. Constraint (D.7) guarantees that exactly one of the overlapping packages is used, which prohibits multiple inclusion of the same item into the optimal solution, and so guarantees that the chosen set of packages partition V (this is what makes the chosen packages bins). Again for the bin covering the constraint may be relaxed into a less then equality, since the maximisation will make sure that as many packages as possible are chosen; on the other hand, without the more restrictive constraint, the bin packing problem would always yield a solution counting zero bins. Constraint (D.8) simply defines the domains of the p_j variables.

2.3 Equivalence class formulation

Though the subset formulation of Section 2.2 goes a long way to mitigate the computational complexity, observing that for large bin sorting problems we can have, and typically do have, many items with equal values, we can give an even more compact problem formulation, where equal values are not viewed as distinct items, but rather as a single item with a multiplicity equal to the number of actual such same-valued items. This is done by collecting equal items into *equivalence classes*, and instead of enumerating each item of such a class, formulate the optimization problem over integer decision variables related to the number of items in each equivalence class.

Consider a bin sorting problem $\langle V, t, \bowtie \rangle$. Let an equivalence class E_q be a subset of equal valued items of V; that is, $E_q = \{v_i \in V | v_i = w\}$ for a fixed value w. Obviously, $\min(E_q) = w$. Let p denote the number of all equivalence

classes. The set of p equivalence classes partition V.

We call a tuple $\langle E_q, f_q \rangle$ of an equivalence class E_q and a *factor* f_q, a *selection*. The factor is used to denote the number of items from the equivalence class that are selected in a certain situation. Of course, $0 \leq f_q \leq |E_q|$, and obviously there is a finite number of distinct selections.

Let a *package class* $PC_i = \{\langle E_1, f_{1,i} \rangle, \ldots, \langle E_p, f_{p,i} \rangle\}$ be a set of selections over all equivalence classes, such that

$$\sum_{\langle E_q, f_{q,i} \rangle \in PC_i} \min(E_q) \cdot f_{q,i} \bowtie t, \tag{D.9}$$

where $\bowtie$ is $\leq$ for bin packing and $\geq$ for bin covering, and the objective is to minimize and maximize, for packing and covering, respectively. Let k denote the number of all possible package classes.

Since all package classes contain all equivalence classes, albeit many with a zero factor, all package classes overlap, which then becomes an uninteresting observation (contrary to Section 2.2).

Given a set of k generated package classes, and with some abuse of notation let PC_i be an integer that represents how many "instances" of the package class PC_i that are included in the optimal solution, then the optimization problem can be formulated as:

$$\min / \max \sum_{i=1}^{k} PC_i \text{ subject to} \tag{D.10}$$

$$\sum_{i=1}^{k} f_{q,i} \cdot PC_i = |E_q| \qquad \forall q = 1, \ldots, p \tag{D.11}$$

$$PC_i \geq 0 \qquad \forall i = 1, \ldots, k \tag{D.12}$$

The objective function (D.10) sums over all the variables representing the number of each package class instance; this sum is to be minimized for bin packing and maximized for bin covering. Constraint (D.11) sums for each equivalence class over all the package classes and multiplies with the factor for each respective package class, to get the total number of used values from the specific equivalence class. Naturally, this number cannot exceed the number of values in the equivalence class for the package class, and has to be exactly equal to the number of values in the equivalence class in order to avoid trivial

solutions with zero bins. Constraint (D.12) simply sets zero as the lower bound for the number of instantiations of the respective package classes.

An upper bound for PC_i could be pre-calculated, but it is not clear whether this will have any impact on the computational complexity, and this has not been investigated.

For brevity, we will in the following use the term *package* in place of *package class*.

3 Optimal Solutions

Though theoretically possible, generating *all* packages is practically intractable; we can do this only for small n. And though generating *all* possible package classes is more tractable than generating all packages, it still amounts to a huge computational effort for large bin sorting problems. However, we can calculate the packages in a more clever way, by observing that the optimization criterion really means that the resulting bins of an optimal solution should be as close to the target value as possible. Calculating such packages saves a lot of computational effort, as is shown in Section 5.

In this section we will argue for why and how we can calculate a specific subset of all possible packages, but still get an optimal solution from the subset and equivalence class formulations of Section 2. For clarity, we will here treat packing and covering separately in their own subsections.

In both cases, we have a bin sorting problem $\langle V, t, \leq \rangle$ for packing, and $\langle V, t, \geq \rangle$ for covering. The total value over all n items of V is $W = v_1 + v_2 + \cdots + v_n$. A *feasible* solution $f_t = \{b_1, b_2, \ldots, b_m\}$ to a bin sorting problem, is a solution where for each bin, $B_j \leq t$ for packing, and $B_j \geq t$ for covering, and an *optimal* solution is a feasible solution where the number of bins m is minimized for packing, and maximized for covering.

Let us also note the distinction between *bins* and *packages*. Bins partition V so that no item in V can be in more than one bin, whereas packages can share items; it is the task of the bin sorting solver to select among the packages so that a single item does not appear in more than one bin.

3.1 Bin Packing

A feasible solution $f_t = \{b_1, b_2, \ldots, b_m\}$ for a bin packing problem $\langle V, t, \leq \rangle$ is said to be *true* if all items of V are sorted.

For a feasible solution, all bins b_j are on or below target, that is $B_j \leq t$. For bin packing we want to minimize the number of bins. Thus, it seems to make sense to have bins that are as close to the target (but not above) as possible.

Definition 1: *A bin (or package) is said to be* fit *if adding the least valued item from* V *gets it above the target value. That is, a bin* b_j *is fit if*

$$B_j + \min(V) > t.$$

Definition 2: *Let* V^v *be a set of* virtual *items, such that* $V^v \cap V = \varnothing$ *and* $v_k = \min(V)$ *for all* $v_k \in V^v$.

The virtual items are not in the original problem formulation but are introduced as possible items that can be put in bins to complete a bin into a fit bin, in order to guarantee fit solutions. However, we show later that virtual items can be removed from a solution and thus a *true* solution can be generated.

A feasible solution $f_f = \{b_1, b_2, \ldots, b_m\}$ for a bin packing problem $\langle V, t, \leq \rangle$ is said to be *fit* if all items of V, plus an arbitrary number of virtual items, are sorted, and all bins in f_f are fit.

Lemma 1: *For a bin packing problem* $\langle V, t, \leq \rangle$*, a feasible* true *solution* f_t *exists if and only if a feasible fit solution* f_f *exists.*

Proof. First we show that to a feasible true solution f_t we can add virtual items to non-fit bins to make a feasible fit solution f_f.

Assume a non-fit bin b_j, thus $B_j + \min(V) \leq t$. Add $\lfloor (t-(B_j+\min(V)))/\min(V) \rfloor$ number of virtual items to b_j, the bin is now fit by definition since adding one additional virtual item makes the value of the bin be larger than the target value t. This can be done for any non-fit bin in f_t.

Second, we show that we can remove (virtual) items from the bins of f_f and get feasible *true* solution.

Assume a fit solution f_f. Let B^f be the sum of the values of all bins for a feasible fit solution f_f. Then the total number of virtual items is equal to $\lfloor (B^f - W)/\min(V) \rfloor$. If this number of virtual items are removed from the bins in f_f, the total number of items of value $\min(V)$ in all bins will be equal to $|E_n|$ and thus the modified solution will be *true*. Note that removing virtual items from a bin can only decrease its value, and since a fit bin is by definition

already below the target, so will the *reduced* bin be. The total number of items of value $\min(V)$ will be equal to or larger than $\lfloor {}^{(B^f - W)}/_{\min(V)} \rfloor$, thus it is possible to remove $\lfloor {}^{(B^f - W)}/_{\min(V)} \rfloor$ items of value $\min(V)$. Consider any set of bins that is the result of removing $\lfloor {}^{(B^f - W)}/_{\min(V)} \rfloor$ items of value $\min(V)$ from the bins of the fit solution f_f. The set of reduced bins result in a feasible solution for the *true* problem, since all bins have a value less than the target value and the number of items for every value will be equal to the number of values in the equivalence class for the same value. □

Theorem 1: *Let B^t_{opt} be an optimal true solution to the bin packing problem $\langle V, t, \leq \rangle$, and let B^f_{opt} be an optimal fit solution to the same problem. Then*

$$|B^t_{opt}| = |B^f_{opt}|.$$

Proof. If there exists a feasible solution for the fit problem that is optimal, then no better solution than that exists and, according to Lemma 1, there must exist a feasible solution for the *true* problem that yields the same result and thus no better solution can exist. □

3.2 Fit package generation

As mentioned, a major issue with the equivalence classes approach to solve bin packing problems is the computation of all package classes that might form part of the optimal solution. Computing and then filtering the power set of V is computationally heavy even for relatively small size problems, therefore a less demanding procedure is required. Given the aforementioned notions, the computation of all *fit* package classes can be formulated as a Constraint Satisfaction Problem (CSP).

Regard the bin packing problem $\langle V, t, \leq \rangle$, with the equivalence classes E_q $(q = 1, \ldots, p)$. Let f_q $(q = 1, \ldots, p)$ be the factor for E_q, that is, an integer variable representing how many values from E_q that are chosen to form a package class. Let F be an integer number such that $F = \lfloor {}^{t}/_{\min(V)} \rfloor$. The

CSP formulation is as follows:

$$0 \leq f_q \leq F \qquad \forall q = 1, \ldots, p \tag{D.13}$$

$$\sum_{\substack{j=1 \\ q \neq j}}^{p} \Big(\min(E_j) \cdot f_j + (f_q + 1) \cdot \min(E_q) \Big) > t$$

$$\forall q = 1, \ldots, p \tag{D.14}$$

Constraint (D.13) limits the factor for each equivalence class to be at most as large as the value F; constraint (D.14) states that the sum of values contained in a *fit package* goes above the target value as soon as we increase the value of any factor by *one*.

Finding a satisfiable solution to this CSP problem means to find a combination of values that, together will result in a *fit* package class. To obtain the whole set of fit package classes, it is possible to set up another problem with the same constraints, plus one constraint ruling out the solution just found. Let $S = \{f_1^*, \ldots, f_p^*\}$ be the solution of the CSP problem, where $f_i^* \ \forall i = 1, \cdots, p$ is the factor selected for the equivalence class p, then the additional constraint is:

$$\neg\Big(\bigwedge_{f_i^* \in S} (f_i = f_i^*) \Big) \tag{D.15}$$

Constraint D.15 ensures that the solution found in the previous iteration cannot be selected in the current one, so that the solver has to find a new one. In order to find the complete set of *fit* package classes, one has to set up a loop and, for each iteration, add the constraint to rule out the last solution found for the new CSP problem. The loop goes on until the problem becomes unsatisfable, which means that all package classes have been found.

Example of Equivalence Class Formulation Consider the bin packing problem $\langle V, t, \leq \rangle$, with $V = \langle 50, 50, 40, 40, 10, 10 \rangle$ and $t = 100$.

There are three equivalence classes, all of size 2:

$E_1 = \langle 50, 50 \rangle$

$E_2 = \langle 40, 40 \rangle$

$E_3 = \langle 10, 10 \rangle$

And there will also be 8 virtual items of value 10. We need to be able to form feasible packages by using the CSP model in Section 3.2 and, since we only have two items of value 10 and the target is 100 we need 8 virtual items.

The CSP will now yield all packages that are "just below the target", meaning that if the smallest items from V is added, namely item of value 10, the total value will outreach the target value 100

$$PC_1 = \{\langle E_3, 10\rangle\}$$

$$PC_2 = \{\langle E_2, 1\rangle, \langle E_3, 6\rangle\}$$

$$PC_3 = \{\langle E_2, 2\rangle, \langle E_3, 2\rangle\}$$

$$PC_4 = \{\langle E_1, 1\rangle, \langle E_3, 5\rangle\}$$

$$PC_5 = \{\langle E_1, 1\rangle, \langle E_2, 1\rangle, \langle E_3, 1\rangle\}$$

$$PC_6 = \{\langle E_1, 2\rangle\}$$

For readability, only the equivalence classes E_q with factors $f_q > 0$ have been included above. Now, regarding only the non-zero factors for each equivalence class in the package classes, constraint (D.11) becomes:

$$PC_4 + PC_5 + 2\ PC_6 \geq 2$$

$$PC_2 + 2\ PC_3 + PC_5 \geq 2$$

$$10\ PC_1 + 6\ PC_2 + 2\ PC_3 + 5\ PC_4 + PC_5 \geq 2$$

We only have two items of value 10 and, in PC_4 and PC_5, E_3 has multiplicity 1, while in PC_6 E_3 has multiplicity 2, hence the coefficients in the first inequality. The bin packing requires that all items are placed into bins, therefore there should be an equality sign; when we generated packages though, we used virtual bins also, therefore now we need to allow for more items than actually available in V, but not less, hence the "$\geq$" sign. The same procedure applies to the other two equivalence classes as shown in second and third inequality. Since the problem is a minimization, most of the virtual items will not be used (the ones left are not enough to form another bin), therefore the solution will be optimal with respect to V (as explained in detail later on). Then, the objective is:

$$\min PC_1 + PC_2 + PC_3 + PC_4 + PC_5 + PC6.$$

3.3 Bin Covering

For bin covering we want to maximize the number of bins, and hence it seems to make sense to have bins that are as close to the target (but not below) as possible.

Definition 3: *A bin (or package) is said to be* skinny *if removing from it its least valued item gets it below the target value. That is, a bin b_j is skinny if*

$$B_j - \min(b_j) < t.$$

A feasible solution $f_s = \{b_1, b_2, \ldots, b_m\}$ for a bin covering problem $\langle V, t, \geq \rangle$ is said to be *skinny* if all bins in f_s are skinny, and an arbitrary number of items of V is sorted. The items of V that are not sorted are called the *surplus* items. A feasible solution $f_t = \{b_1, b_2, \ldots, b_m\}$ to a bin covering problem $\langle V, t, \geq \rangle$, is said to be *true* if all items are sorted.

Lemma 2: *For a bin covering problem $\langle V, t, \geq \rangle$, a feasible* true *solution f_t exists if and only a feasible skinny solution f_s exists.*

Proof. Regard a true feasible solution $f_t = \{b_1, b_2, \ldots, b_m\}$. For each non-skinny bin b_j, we can iteratively remove its least valued item $\min(b_j)$ until b_j becomes skinny. Obviously, the resulting skinny solution will have the same number of bins as f_t.

Regard a skinny feasible solution $f_s = \{b_1, b_2, \ldots, b_m\}$. This has a set of non-sorted *surplus* items, V_{sur}. These items can be put on arbitrary skinny bins to make the bins non-skinny. Doing so for a bin $b_j \in f_s$ will increase the value B_j which means that it is still on or above target. Thus, we can from the skinny feasible solution generate a true solution f_t.

For both implications, the number of bins is preserved. □

Theorem 2: *Let B^t_{opt} be an optimal true solution to the bin covering problem $\langle V, t, \geq \rangle$, and let B^s_{opt} be an optimal skinny solution to the same problem. Then*

$$|B^t_{opt}| = |B^s_{opt}|.$$

Proof. If there exists a feasible solution for the skinny problem that is optimal, then no better solution than that exists and, there must exist a feasible solution for the true problem that yields the same result and no better solution can exist. □

3.4 Skinny package generation

As for the bin covering, it is possible to setup a Constraint Satisfaction Problem based on the definition of Fit Bin and run it multiple time to produce, at each iteration, a different valid package, until the problem becomes unfeasible; then we know we have found all feasible packages.

Regard the bin covering problem $\langle V, t, \geq \rangle$, with the equivalence classes E_q $(q = 1, \ldots, p)$. Let f_q $(q = 1, \ldots, p)$ be the factor for E_q, that is, an integer variable representing how many values from E_q that are chosen to form a package class. The CSP formulation is as follows:

$$0 \leq f_q \leq |E_q| \qquad \forall q = 1, \ldots, p \tag{D.16}$$

$$f_q > 0 \Rightarrow \sum_{\substack{j=1 \\ q \neq j}}^{p} \Big(\min(E_j) \cdot f_j + (f_q - 1) \cdot \min(E_q) \Big) < t \qquad \forall q = 1, \ldots, p \tag{D.17}$$

Constraint D.16 limits the factor for each equivalence class to be at most as large as the size of the equivalence class itself; constraint D.17 states that the sum of values contained in a *skinny package* goes below the target value as soon as we reduce the value of any factor by *one*. Unlike the corresponding constraint for the bin packing problem, in this case it is necessary to specify that we can only reduce a value if it is larger than zero.

Let $S = \{f_1^*, \ldots, f_p^*\}$ be the solution of the CSP problem, where f_i^* $\forall i = 1, \cdots, p$ is the factor selected for the equivalence class p, then the additional constraint is:

$$\neg\Big(\bigwedge_{f_i^* \in S} (f_i = f_i^*) \Big) \tag{D.18}$$

Example of Equivalence Class Formulation Consider the bin covering problem $\langle V, t, \geq \rangle$, with V and t as in section 3.2 and so are the equivalence classes.

We can generate four skinny package classes:

$PC_1 = \{\langle E_1, 2 \rangle\}$

$PC_2 = \{\langle E_1, 1 \rangle, \langle E_2, 2 \rangle\}$

$PC_3 = \{\langle E_1, 1 \rangle, \langle E_2, 1 \rangle, \langle E_3, 1 \rangle\}$

$$PC_4 = \{\langle E_2, 2\rangle, \langle E_3, 2\rangle\}$$

For readability, only the equivalence classes E_q with factors $f_q > 0$ have been included above.
Now, looking only at the non-zero factors for each equivalence class in the package classes, constraint (D.11) becomes:

$$2\ PC_1 + PC_2 + PC_3 \leq 2$$

$$2\ PC_2 + PC_3 + 2\ PC_4 \leq 2$$

$$PC_3 + 2\ PC_4 \leq 2$$

Since the the bin covering does not require all items to be allocated to binsl, the equality constraint can be relaxed, hence the inequalities above.

Finally, the objective is:

$$\max PC_1 + PC_2 + PC_3 + PC_4.$$

One optimal solution is to select PC_1 once ($PC_1 = 1$) and PC_4 once ($PC_4 = 1$). However, another optimal solution is to select two "instances" of PC_3 ($PC_3 = 2$). Both of these are of course skinny solutions.

So, we really only need to generate fit (for packing) and skinny (for covering) packages and package classes to get optimal solutions from the subset and equivalence class formulations. This saves a lot of computational effort, as shown in Section 5.

4 Sub-Optimal Solutions

The equivalence class formulation significantly improves the runtime performance of the optimizer compared to the naïve formulation, as shown in Section 5. Nevertheless, BC is still a combinatorial NP-hard problem and for some problems, calculating the true optimum might be expensive in terms of computational effort. The alternative is to consider heuristic methods that can provide a sub-optimal solution in a shorter time. Based on the equivalence classes approach, a heuristic method was developed to simplify the problem and calculate a suboptimal solution faster. One hypothesis that led to the heuristic is that the number of package classes related to a BC problem $\langle V, t, \geq\rangle$ does not depend entirely on $|V|$, but rather on the number of

equivalence classes and their cardinality; the more different values, the more possible combinations, the more package classes.

Therefore, the goal of such method is to provide a set of approximated equivalence classes, C^*, where a certain number of consecutive exact equivalence classes are merged together. We call the approximated equivalence classes *chains*.

Let $E = \langle E_1, E_2, \ldots, E_p \rangle$ be the set of p number of equivalence classes over the values of V, ordered so that $\min(E_i) < \min(E_{i+1})$ (for $i = 1, \ldots, p-1$). Let C be a set of *chains*, sets of one or more consecutive equivalence classes from E, $C = \bigcup_{i=1}^{p-l+1} \{E_i, E_{i+1}, \ldots E_{i+l-1}\}$ for $l = 1, \ldots, p$, and let the size of a chain be the sum of the sizes of the equivalence classes it is composed of; for $C_j \in C$, $\|C_j\| = \sum_{E_i \in C_j} |E_i|$.

Note that chains will have common elements (equivalence classes), just as packages, therefore it is possible to define the set O_i of all chains containing a certain value.

Example of Chains Generation

$E = \langle E_1, E_2, E_3, E_4 \rangle$, $p = 4$, with $\min(E_i) < \min(E_{i+1})$ for $i = 1, \ldots, 3$,

$C = \bigcup_{i=1}^{p-l+1} \{E_i, E_{i+1}, \ldots E_{i+l-1}\}$ for $l = 1, \ldots, p$

$$
\begin{array}{ll}
\{E_1\}, \{E_2\}, \{E_3\}, \{E_4\}, & l = 1, i = 1, \ldots, p \\
\{E_1, E_2\}, \{E_2, E_3\}, \{E_3, E_4\}, & l = 2, i = 1, \ldots, p-1 \\
\{E_1, E_2, E_3\}, \{E_2, E_3, E_4\}, & l = 3, i = 1, \ldots, p-2 \\
\{E_1, E_2, E_3, E_4\} & l = 4, i = 1, \ldots, p-3
\end{array}
$$

4.1 Heuristic for the bin packing problem

When it comes to the bin packing problem, a way to generate the above mentioned chains is to merge exact equivalence classes as explained in the above section and to assign to all the values of the resulting approximated equivalence class the largest value of all classes that have been merged. This is done so that the solution generated when solving the simplified problem is valid: if a value smaller than the one belonging to the largest class merged were to be assigned to the approximated class, the optimal solution to the simplified problem might be smaller than the optimal solution to the original

problem and, therefore, unfeasible.

Definition 4: *Let define the* minimum theoretical number of bins M *as the number of bins we could achieve if we could break down the items into smaller ones and reallocate the overweight from each bin to form other ones:* $M = {}^{W}/_{t}$. *Such value can be achieved by relaxing the integrality constraint in the initial problem, as pointed out in [11] where such value is defined as* lower bound.

Let the chain E_{l-m} be the result of merging the equivalence classes E_l, E_m, where $\min(E_l) < \min(E_m)$. Then $\min(E_{l-m}) = \min(E_m), |E_{l-m}| = |E_l| + |E_m|$.

$$|E_{l-m}| \cdot \min(E_{l-m}) = (|E_l| + |E_m|) \cdot \min(E_m) > \\ |E_l| \cdot \min(E_l) + |E_m| \cdot \min(E_m)$$

The gain γ is then:

$$\min(E_m) \cdot (|E_l| + |E_m|) - \\ |E_l| \cdot \min(E_l) + |E_m| \cdot \min(E_m) = \\ |E_l| \cdot (\min(E_m) - \min(E_l))$$

The new minimum theoretical number of bins is $M' = {}^{(W+\gamma)}/_{t}$. The gain can be used to decide which classes is better to merge when simplifying an instance. Using a greedy algorithm it is possible to quickly generate all chains and calculate γ for each of them. It is then possible to formulate a MILP model to select the chains that produce the minimum loss, given a desired number of equivalence classes d.

Let x_i $(i = 1, \ldots, k)$ be 0-1-variables representing whether the chain C_i is

chosen ($x_i = 1$) or not ($x_i = 0$). The MILP formulation is as follows:

$$\min \sum_{i=1}^{k} x_i \cdot \gamma_i \tag{D.19}$$

$$\sum_{i=1}^{p} x_i = d \tag{D.20}$$

$$\sum_{x_i \in O_j} x_i = 1 \qquad \forall j = 1, \ldots, p \tag{D.21}$$

$$x_i \in \{0, 1\} \qquad \forall i = 1, \ldots, k \tag{D.22}$$

The objective function (D.19) is the sum of all gains for the chains that are chosen. Constraint D.20 states that exactly d chains have to be chosen. Constraint (D.21) states that overlapping chains are mutually exclusive. Finally, (D.22) sets the variable domains to be binary.

Note that the heuristic has been developed bearing in mind that, with a normal distribution, there are only a few values at the edges of the *bell curve*, while most of the values appear in the middle of it. Therefore, while containing the same number of values, the chains that are closer to the edges will contain more classes from E, involving a larger loss than the ones closer to the centre. However, as those values are smaller in number compared to the ones in the middle, the overall loss seems not to be significant.

4.2 Heuristic for the bin covering problem

Once again it is possible to draw inspiration from the results achieved for the bin packing problem to develop a working heuristic method for the bin covering problem too. By merging equivalence classes it is in fact possible to generate a simplified problem that provides a sub optimal solution. This time it is required to assign to the resulting equivalence class the value of the items from the class with the smallest values, in order to generate a valid solution. Since this is a maximization problem, it makes more sense to talk about *loss* σ (rather than a gain) that affects the maximum (instead of minimum) theoretical number of bins. Once again this value corresponds to the optimum of the cost function for the bin covering when relaxing the integrality constraint.

It is possible to calculate the *loss* σ related to the merging of two or more equivalence classes in terms of decrease in the total value W and, therefore,

of the maximum theoretical number of bins M.

Let the chain E_{l-m} be the result of merging the equivalence classes E_l, E_m, where $\min(E_l) < \min(E_m))$. Then $\min(E_{l-m}) = \min(E_l), |E_{l-m}| = |E_l| + |E_m|$.

$$|E_l| \cdot \min(E_l) + |E_m| \cdot \min(E_m) > \\ |E_{l-m}| \cdot \min(E_{l-m}) = \min(E_l) \cdot (|E_l| + |E_m|)$$

The loss σ is then:

$$|E_l| \cdot \min(E_l) + |E_m| \cdot \min(E_m) - \\ \min(E_l) \cdot (|E_l| + |E_m|) = \\ (\min(E_m) - \min(E_l) \cdot |E_m|)$$

The new maximum theoretical number of bins is $M' = {}^{(W-\sigma)}/_{t}$. Setting up exactly the same MILP model as in 4.1 it is possible to find the set of merged classes that minimizes the gain while guaranteeing a desired number of equivalence classes for the simplified problem.

Example of Chain Optimization

Consider the example shown in Section 4, and assume that the values and sizes of the equivalence classes are respectively $\min(E_1) = 13$, $|E_1| = 10$, $\min(E_2) = 15$, $|E_2| = 7$, $\min(E_3) = 20$, $|E_3| = 12$, $\min(E_4)$, $|E_4| = 3$. If the desired number of classes is $d = 2$ we can compute the loss for each chain based on (D.19)–(D.22):

$$C_1 = \{E_1\} \qquad \sigma(C_1) = 0$$
$$C_2 = \{E_2\}, \qquad \sigma(C_2) = 0$$
$$C_3 = \{E_3\}, \qquad \sigma(C_3) = 0$$
$$C_4 = \{E_4\}, \qquad \sigma(C_4) = 0$$
$$C_5 = \{E_1, E_2\}, \qquad \sigma(C_5) = 7 \cdot (15 - 13) = 14$$
$$C_6 = \{E_2, E_3\}, \qquad \sigma(C_6) = 12 \cdot (20 - 15) = 60$$
$$C_7 = \{E_3, E_4\}, \qquad \sigma(C_7) = 3 \cdot (25 - 20) = 15$$

$$C_8 = \{E_1, E_2, E_3\},$$
$$\sigma(C_8) = \sigma(C_5) + 12 \cdot (20 - 13) = 98$$
$$C_9 = \{E_2, E_3, E_4\},$$
$$\sigma(C_9) = \sigma(C_6) + 3 \cdot (25 - 15) = 90$$
$$C_{10} = \{E_1, E_2, E_3, E_4\},$$
$$\sigma(C_{10}) = \sigma(C_8) + 3 \cdot (25 - 13) = 134$$

According to constraint (D.20) only d chains can be selected:

$$C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_8 + C_9 + C_{10} = 2$$

According to constraint (D.21) some chains are mutually exclusive, so each equivalence class must be picked exactly once. As stated before, chains are sets, but with some abuse of notation, we here use them as binary variables to state whether a chain is selected ($C_i = 1$) or not ($C_i = 0$).

$$\begin{aligned} C_1 + C_5 + C_{10} &= 1 \\ C_2 + C_5 + C_6 + C_8 + C_9 + C_{10} &= 1 \\ C_3 + C_6 + C_7 + C_8 + C_9 + C_{10} &= 1 \\ C_4 + C_7 + C_9 + C_{10} &= 1 \end{aligned}$$

Finally, the objective function to minimize is:

$$\begin{aligned} & C_1 \cdot \sigma(C_1) + C_2 \cdot \sigma(C_2) + C_3 \cdot \sigma(C_3) \\ & + C_4 \cdot \sigma(C_4) + C_5 \cdot \sigma(C_5) + C_6 \cdot \sigma(C_6) + C_7 \cdot \sigma(C_7) \\ & + C_8 \cdot \sigma(C_8) + C_9 \cdot \sigma(C_9) + C_{10} \cdot \sigma(C_{10}) \end{aligned}$$

5 Computational Analysis

We ran an extensive analysis over 1500 generated problems, comparing both the standard and the equivalence class formulations for both the bin packing and bin covering problem; we also compare the standard and equivalence classes formulations for the bin packing problem over different benchmark sets from the literature:

Table 1: Comparison of the standard formulation and the equivalence class formulation over the benchmark instance sets, showing the number of solved instances within one minute and the average time calculated over the instances that did not time out. For each instance set it is reported the total number of instances and the number of instances that generate less than ten million packages. The '-' is used when no instance is solved, while the '*' means that the package generation algorithm run out of memory.

	Instances		STD		EQU	
	Complete Set	$\leq 1.0 \times 10^7$	solved	average	solved	average
Falkenauer U	80	80	16	34.11	80	1.64
Falkenauer T	80	80	10	30.07	80	0.76
Scholl 1	720	313	185	26.92	294	1.29
Scholl 2	480	54	50	6.86	48	7.69
Scholl 3	10	7	0	-	0	-
Schwerin 1	100	100	51	32.23	26	48.49
Schwerin 2	100	100	40	37.72	5	53.66
Wäscher	17	0	0	-	*	*
Schoenfield	28	0	0	-	*	*

- Falkenauer [14]: two sets with 80 instances each;
- Scholl [15] three sets with 720, 480 and 10 instances respectively;
- Schwerin [16] two sets with 100 instances each;
- Wäscher [17] a set with 17 instances;
- Schoenfield [18] a set with 28 instances.

All instances have been solved using the state-of-the-art MILP solver Gurobi 9 [19]. The time limit is 1200 seconds and the solver has been used with its default setting. All the experiments were performed on an *Intel Core i7 6700K, 4.0 GHZ, 32GB RAM* running *Ubuntu-16.04*. The implementation of all the models presented in this paper, as well as the benchmark instances. The instance generator is available on `https://github.com/sabinoroselli/bin_covering-packing.git`

To generate the instances we approximated a normal distribution. The parameters to generate instances are the number of items, the range of values, the average value of such range (which is the mean of the distribution) and the standard deviation. Since the results show a skewed distribution, we reported the median and upper and lower quartile for each category.

The first set of tests, reported in Table 2 has been run using the standard formulation to solve both the covering and the packing problem. The number of items ranges from 10 to 70 and the target value ranges from 300 to 900. The values of the items range from 130 to 170 and the value of the deviation ranges from 10 to 90; finally, five different instances are generated for each set of parameters by changing the random generation seed. Results show that different values for the deviation do not affect the running time significantly so they are not shown explicitly in the table: for each size and target value, the average over 25 instances is shown (five different values of deviation times five different random seeds).

Results from this first simulation show that, as expected, an increasing number of items makes the problem harder to solve. It is interesting though, to notice that even for relatively high number of items, there are still many instances that can be solved almost instantly which means that also large problems can have trivial solutions. What does affect the complexity of the instance, according to the results, is the target value t; though there are some outliers, most of the unsolved instances for the covering problem have a target value of 500, while for the packing problem it is 700.

In the second set of tests the number of items ranges from 60 to 500, while the other parameters are the same as in the previous tests. For the instance classes counting 200 to 500 items and with a target value of 800 and 900 it was only possible to solve one instance, therefore it was not possible to calculate the quartiles. The results, summarized in Table 3 show, as for the tests run for the standard formulation, an increasing amount of time required to find the optimal as the number of items and the target value increases (given a certain distribution and, thus, a certain number of equivalence classes). Unlike the standard approach though, the equivalence classes formulation seems to have a more steady trend: the former's performance is heavily affected by the intrinsic hardness of a specific instance, being able sometimes to immediately solve large size instances with a large target value, while getting stuck on relatively small sized instances; on the other hand, the latter's behaviour is more predictable and steadily increases with the instances parameters, as shown in Figure 1. Also, the comparison of Table 2 and Table 3 show tighter values of the quartiles for the equivalence classes formulation; this mean that the solving time for a given class of instances (in terms of generated packages for example) is more predictable.

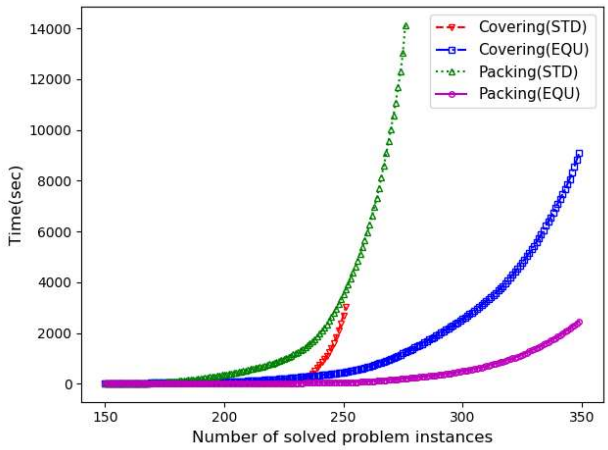

Figure 1: Comparison of model formulations (STD stands for *standard* and EQU for *equivalence classes*) and sorting type over generated problems.

Another conclusion we can draw from the data is the strong correlation between the solving time and the number of packages, which in turn is affected by the target value and, to a lesser extent, by the number of objects. A larger target value means an exponentially higher number of possible combinations to form skinny (or fit) packages; also having a higher number of items in each equivalence class means that it is possible to form more combinations of them that make valid packages, though this is true only up to a certain value. For instance, if the target value is 300, having two or 200 items of value 200 does not make any difference.

For this reason, an instance with 60 items and a target value of 300 only yields a few thousand packages, while the same instance with a target value of 900 can count millions of them. As mentioned before, the increase in the number of items also causes an increase in the number of packages, which seems nonetheless to happen within the same order of magnitude.

Though the number of packages has a direct impact on the solving time, the equivalence class formulation still proves to be efficient to solve very large sized problems, being scalable in terms of number of items (which is usually the limitation for the standard approach). Even so, the package generation time via a greedy algorithm and the model generation time are directly proportional

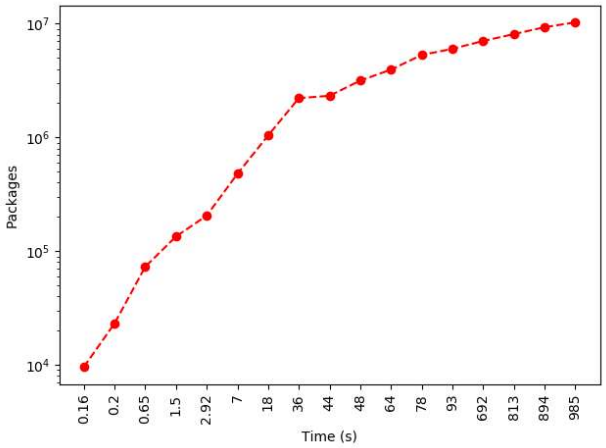

Figure 2: Evaluation of package generation time (in seconds) against number of generated packages (log scale).

to the number of generated packages and for very hard instances they might not be negligible. Figure 2 shows how the package generation time increases with respect to the number of packages (benchmark instances from the *Scholl* set have been employed to evaluate the package generation efficiency): it is still close to one minute for instances leading to four million packages but it goes up to about fifteen minutes for instances of around ten million packages. Though we have not investigated the subject for this work, we believe that there is quite some room for improvement in the implementation of the package generation algorithm (changing the programming language to begin with, since now it is written in Python). While solving the benchmark instances, the largest instances we could solve before running out of memory counted circa thirty million packages; memory overflow is another matter we plan to address in our future work.

The third set of instances, summarized in Table 1, shows the performance of the equivalence classes method over different sets of benchmark instances. In order to compare our formulation to the other relevant algorithms we found in the literature, we refer to Table I and Table II of [3], where such algorithms are evaluated using the same benchmark sets listed above, and setting the time limit to one minute (the implementation for such algorithms is available at the

authors web page [20]. Though the computers used are different, we believe that the comparison still gives a hint of the methods potentials. Nonetheless, we decided to run the benchmark instances again for the standard formulation (called *Basic ILP* in [3], since the authors used a different solver). Moreover, the algorithms listed in those tables, are specifically tailored to solve the bin packing problem, while our formulation provides a linear program that can be fed to any solver (or extended with additional constraints) and so can be done with the standard formulation. Therefore we decided to make a more rigorous comparison between the standard formulation and the equivalence classes formulation.

As already discussed when commenting on Table 3, the number of generated packages directly affects the solving time when using the equivalence classes approach; therefore we only considered instances counting less than ten millions packages. Table 1 shows the number of instances in each set and, next to it, the number of instances that lead to generate less than ten million packages. The equivalence class formulation is much faster than the standard formulation for the *Falkenauer* sets and can deal with all instances in less than 2 seconds each. When it comes to the *Scholl* instances, in the first set the equivalence classes formulation is still much faster and can deal with almost twice as many instances before the time limit while it performs very similarly, though slightly worse in the second set. Neither method can deal with any of the instances in the third set. The standard formulation performs considerably better than the equivalence classes one in both *Schwerin* sets, both in terms of instances solved and average time. Finally, the standard formulation cannot solve any of the instances from either the *Wäscher* or the *Schoenfield* set within the time limit, while the equivalence class formulation cannot even get started, since the computer ran out of memory while generating the packages.

A possible remedy to handle such hard instances is to reduce the number of classes by merging them into chains as explained in sections 4.2 and 4.1. The heuristic does not guarantee an optimal value to the original problem, but it can drastically reduce the number of packages, thus speeding up the model generation tremendously, while producing very close-to-optimal results. Table 4 shows an example for both the covering and the packing problem where the number of equivalence classes is progressively reduced from the original one, shown on the first line (the one that would lead to the true optimal

solution). The instance is evaluated considering two different target values: 450 is an exact multiple of 150 (the mean value of the items) while 525 is as far as possible from being an exact multiple. Also, two different values for the deviation are selected: 10 (corresponding to the data shown in the upper part of Table 4) and 90 (corresponding to the remaining data); neither of the two parameters seems to largely affect the accuracy. What we can see though, is a dramatic reduction in the number of generated packages as the number of classes decreases, while the optimal value for the simplified instances is still very close to the optimum for the original one.

Acknowledgements

The authors thank Folkmar Frederik Ramcke for implementing the algorithms for package generation.

6 Conclusions

In this paper we have presented alternative formulations to solve both the bin covering and the bin packing problem; we have shown that these formulations perform particularly well when the number of different values in the problem instance is limited. In such cases, our formulations allow to to solve problem counting hundreds of items in a considerably short time. This feature can prove useful for industrial applications where the number of items is high but the range of values is limited, such as battery recycling (for bin packing) or fixed tray weight sorting in food processing (bin covering). When this is not the case, our formulations allow for problem simplification by means of merging equivalence classes that still provides a close to optimal solution, while dramatically reducing the computation time. Moreover, the concept of skinny/fit packages can constitute a solid base to improve existing specific-purpose algorithms such as those given by [3], which we intend investigate further.

Table 2: Set of generated instances solved using the standard formulation for both the packing and the covering problem. The size of the instances ranges from 10 to 70 items and the target value from 300 to 900. The median time is reported (calculated over the solved instances) as well as the lower and upper quartile and the number of instances that timed out. The timeout is set to 1200 seconds. If no instance for a given category is solved, the cell is marked with the symbol '-'.

COVERING STANDARD

	300				400				500				600				700				800				900			
	med.	low	upp	TO	med.	low	upp	TO	med.	low	upp	TO	med.	low	upp	TO	med.	low	upp	TO	med.	low	upp	TO	med.	low	upp	TO
10	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0
20	0.00	0.00	0.02	0	0.00	0.00	0.00	0	0.00	0.00	0.02	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0
30	0.01	0.01	1.16	4	0.01	0.01	0.01	0	8.59	0.09	9.58	4	0.01	0.01	0.01	0	0.01	0.01	0.01	0	0.01	0.01	0.01	0	0.01	0.01	0.02	0
40	0.06	0.02	0.16	7	0.01	0.01	0.01	0	18.57	13.95	22.74	12	0.01	0.01	0.02	0	0.01	0.01	0.01	0	0.04	0.01	0.08	0	0.01	0.01	0.01	0
50	0.07	0.03	0.15	3	0.46	0.35	0.54	0	35.50	26.22	54.81	11	0.02	0.02	0.02	0	0.02	0.01	0.02	0	0.42	0.12	21.89	6	0.02	0.01	0.02	0
60	0.17	0.12	0.27	15	0.03	0.03	0.03	0	90.25	32.75	116.26	13	0.03	0.02	0.03	0	0.03	0.03	0.10	0	-	-	-	25	0.02	0.02	0.03	0
70	0.17	0.15	0.25	6	0.90	0.04	2.74	0	65.83	12.89	236.56	11	0.04	0.03	0.04	0	0.03	0.03	0.07	4	-	-	-	25	0.03	0.03	0.03	0

PACKING STANDARD

	300				400				500				600				700				800				900			
	med.	low	upp	TO	med.	low	upp	TO	med.	low	upp	TO	med.	low	upp	TO	med.	low	upp	TO	med.	low	upp	TO	med.	low	upp	TO
10	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0
20	0.00	0.01	0.10	0	0.05	0.03	0.08	0	0.02	0.00	0.03	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0	0.00	0.00	0.00	0
30	0.01	0.01	0.23	0	0.18	0.12	0.26	0	0.08	0.00	0.14	0	0.00	0.00	0.01	0	0.01	0.01	0.01	0	0.00	0.00	0.00	0	0.00	0.00	0.02	0
40	3.85	0.40	30.68	0	0.76	0.49	1.98	0	1.13	0.33	2.09	0	0.01	0.01	0.06	0	1.95	0.12	5.14	15	0.01	0.01	0.01	0	0.01	0.01	0.01	0
50	18.33	8.97	82.04	6	6.76	1.15	21.46	0	3.75	1.00	4.95	0	0.02	0.02	0.02	4	4.56	3.06	9.69	16	0.01	0.01	0.01	0	0.01	0.01	0.01	0
60	39.58	0.10	170.01	11	21.77	3.08	40.42	7	7.61	5.14	13.96	0	0.07	0.04	0.22	0	79.61	18.25	124.53	15	0.01	0.01	0.01	0	0.05	0.03	0.16	0
70	1.02	0.10	44.92	13	473.65	53.34	560.20	8	19.92	16.06	27.06	0	0.04	0.04	0.04	3	153.92	20.90	343.86	13	0.02	0.02	0.02	0	0.02	0.02	0.03	4

Table 3: Set of generated instances solved using the equivalence classes formulation for both the packing and the covering problem. The size of the instances ranges form 60 to 500 items and the target value ranges from 300 to 900. The median time is reported as well as the lower and upper quartile and average number of packages generated (up to tens of thousands, numbers are shown in regular notation, afterwards the scientific notation is employed. The timeout is set to 1200 seconds. The symbol '-' means that only one instance for that class was solved, therefore it was not possible to calculate the upper or lower quartile.

COVERING EQUIVALENCE CLASSES

	300			400			500			600			700			800			900		
	med.	low	upp	# pack	med.	low	upp	# pack	med.	low	upp	# pack	med.	low	upp	# pack	med.	low	upp	# pack	med.	low	upp	# pack
60	0.01	0.01	0.02	1727	0.06	0.05	0.06	5534	0.38	0.29	0.44	44566	1.02	0.79	1.38	1×10^5	7.39	3.91	8.73	3×10^5	39.30	29.96	56.11	1×10^6	106.04	65.85	144.22	3×10^6
70	0.02	0.01	0.02	1948	0.06	0.05	0.07	6321	0.46	0.37	0.52	53643	1.34	0.97	1.96	1×10^5	8.51	6.39	10.85	4×10^5	53.79	41.79	68.32	2×10^6	158.50	110.38	210.63	5×10^6
80	0.02	0.02	0.02	2210	0.07	0.05	0.07	7291	0.57	0.47	0.64	65301	1.94	1.75	2.31	2×10^5	12.48	7.52	13.51	5×10^5	82.53	69.75	98.70	3×10^6	266.01	221.57	359.33	6×10^6
100	0.02	0.02	0.03	2655	0.07	0.06	0.07	8652	0.80	0.66	0.86	82794	3.03	2.37	3.56	2×10^5	18.54	14.43	23.35	7×10^5	160.79	127.88	178.73	4×10^6	462.48	376.44	517.59	1×10^7
150	0.03	0.02	0.03	3128	0.09	0.08	0.10	10261	0.97	0.78	1.11	1×10^5	5.28	4.17	5.55	3×10^5	26.48	19.24	33.50	9×10^5	215.04	121.05	248.88	5×10^6	679.03	-	-	2×10^7
200	0.02	0.02	0.03	3340	0.10	0.09	0.10	10936	0.99	0.91	1.01	1×10^5	5.30	4.73	5.77	4×10^5	27.39	24.41	29.01	1×10^6	324.04	-	-	7×10^6	1115.65	-	-	2×10^7
300	0.01	0.01	0.02	3432	0.09	0.08	0.10	11330	0.96	0.93	1.01	1×10^5	6.04	5.63	6.96	4×10^5	27.71	27.07	29.18	1×10^6	301.28	-	-	8×10^6	1084.21	-	-	2×10^7
500	0.01	0.01	0.01	3477	0.06	0.05	0.08	11469	0.80	0.77	0.90	1×10^5	5.66	5.15	6.02	4×10^5	27.48	25.90	31.20	1×10^6	356.01	-	-	8×10^6	1109.46	-	-	2×10^7

PACKING EQUIVALENCE CLASSES

	300			400			500			600			700			800			900		
	med.	low	upp	# pack	med.	low	upp	# pack	med.	low	upp	# pack	med.	low	upp	# pack	med.	low	upp	# pack	med.	low	upp	# pack
60	0.00	0.00	0.00	281	0.00	0.00	0.00	524	0.03	0.02	0.03	5555	0.33	0.25	0.38	25853	0.64	0.47	0.81	50518	8.01	4.83	15.86	3×10^5	32.80	17.17	48.84	1×10^6
70	0.00	0.00	0.00	303	0.00	0.00	0.01	569	0.04	0.03	0.04	6327	0.37	0.29	0.45	30505	0.73	0.67	0.86	69849	9.56	8.01	10.94	4×10^5	44.81	25.19	64.88	1×10^6
80	0.00	0.00	0.00	329	0.00	0.00	0.01	625	0.04	0.04	0.04	7299	0.48	0.36	0.56	36765	0.96	0.89	1.10	74909	13.38	10.32	15.22	5×10^5	73.35	45.26	81.95	2×10^6
100	0.00	0.00	0.00	370	0.01	0.01	0.01	699	0.05	0.04	0.06	8644	0.49	0.38	0.61	46854	1.19	1.10	1.80	1×10^6	18.61	15.15	33.61	6×10^5	102.39	86.19	140.68	3×10^6
150	0.00	0.00	0.00	412	0.01	0.01	0.01	786	0.04	0.04	0.05	10237	0.56	0.49	0.62	58950	1.47	0.96	3.07	1×10^6	21.64	15.06	26.16	9×10^5	145.84	106.05	182.35	3×10^6
200	0.00	0.00	0.00	429	0.01	0.01	0.01	821	0.07	0.06	0.08	10904	0.60	0.53	0.79	64416	1.82	1.56	3.59	1×10^6	26.50	-	-	1×10^6	192.50	-	-	5×10^6
300	0.00	0.00	0.00	436	0.01	0.01	0.01	841	0.05	0.04	0.05	11286	0.71	0.59	0.81	67163	2.59	1.64	3.16	1×10^6	304.40	-	-	1×10^6	192.70	-	-	5×10^6
500	0.00	0.00	0.00	440	0.01	0.01	0.01	849	0.09	0.08	0.10	11417	0.73	0.61	0.89	68213	2.81	2.17	4.46	2×10^6	673.20	-	-	1×10^6	154.10	-	-	5×10^6

Table 4: Comparison of the optimal solution and solving time (in seconds) with respect to the number of generated packages for an instance of 200 items solved with both packing and covering method when varying the number of classes (Cl.) by simplifying the original instance. The instance is evaluated for target values of 450 and 525 and deviation values of 10 (top part of the table) and 90 (bottom part).

Cl.	COVERING						PACKING					
	Standard Deviation: 10											
	450			525			450			525		
	Packages	Optimum	Time	Packages	Optimum	Time	Packages	Optimum	Time	Packages	Optimum	Time
45	62999	65	0.47	181800	50	2.10	8964	67	0.08	15969	65	0.26
40	44616	65	0.35	118894	50	1.43	6768	67	0.07	11525	65	0.20
35	26057	65	0.19	72368	50	0.50	4137	67	0.04	7814	65	0.12
30	15931	65	0.10	39935	50	0.28	2486	67	0.03	4955	66	0.03
25	8154	65	0.08	19788	50	0.13	1566	67	0.02	2925	66	0.02
20	3742	65	0.03	8621	50	0.05	807	67	0.00	1540	66	0.01
15	1512	65	0.01	3061	50	0.02	411	67	0.00	681	66	0.00
10	445	64	0.00	715	50	0.00	106	68	0.00	220	66	0.00
	Standard Deviation: 90											
	450			525			450			525		
	Packages	Optimum	Time	Packages	Optimum	Time	Packages	Optimum	Time	Packages	Optimum	Time
80	540384	66	9.09	2151942	56	53.90	47068	67	0.52	219327	57	3.77
75	438155	66	6.40	1683682	56	41.53	38926	67	0.37	173769	57	5.44
70	315242	65	3.77	1222000	56	33.51	33502	67	0.25	143324	57	2.24
65	214423	65	1.89	825952	56	15.09	25168	67	0.18	99023	57	6.27
60	181567	65	1.90	647241	56	13.17	18370	67	0.09	67044	57	2.02
55	130228	65	1.26	450641	56	8.11	14857	67	0.08	52659	57	1.19
50	82868	65	0.69	293594	56	2.56	10848	67	0.07	36831	57	0.73
45	67720	65	0.44	216105	56	3.28	7912	67	0.07	25138	57	0.47
40	40267	65	0.23	128418	56	1.95	5988	67	0.04	18789	57	0.16
35	25224	65	0.19	77416	56	0.76	3583	67	0.02	10421	57	0.21
30	14716	65	0.09	42137	56	0.46	2579	67	0.02	7217	57	0.13
25	7903	65	0.05	21239	56	0.26	1491	67	0.02	3650	57	0.03
20	3635	65	0.03	8834	56	0.05	826	67	0.01	1904	58	0.01
15	1437	65	0.01	3149	55	0.02	310	68	0.00	674	58	0.01
10	344	64	0.00	677	54	0.00	112	69	0.00	213	59	0.00

References

[1] E. G. Coffman, J. Csirik, G. Galambos, S. Martello, and D. Vigo, "Bin packing approximation algorithms: Survey and classification", in *Handbook of combinatorial optimization*, Springer New York, 2013, pp. 455–531.

[2] S. F. Assmann, D. S. Johnson, D. J. Kleitman, and J.-T. Leung, "On a dual version of the one-dimensional bin packing problem", *Journal of algorithms*, vol. 5, no. 4, pp. 502–525, 1984.

[3] M. Delorme, M. Iori, and S. Martello, "Bin packing and cutting stock problems: Mathematical models and exact algorithms", *European Journal of Operational Research*, vol. 255, no. 1, pp. 1–20, 2016.

[4] G. Belov and G. Scheithauer, "A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting", *European Journal of Operational Research*, vol. 171, no. 1, pp. 85–106, 2006.

[5] F. Brandão and J. P. Pedroso, "Bin packing and related problems: General arc-flow formulation with graph compression", *Computers & Operations Research*, vol. 69, pp. 56–67, 2016.

[6] F. Clautiaux, S. Hanafi, R. Macedo, M.-É. Voge, and C. Alves, "Iterative aggregation and disaggregation algorithm for pseudo-polynomial network flow models with side constraints", *European Journal of Operational Research*, vol. 258, no. 2, pp. 467–477, 2017, ISSN: 0377-2217.

[7] M. Delorme and M. Iori, "Enhanced pseudo-polynomial formulations for bin packing and cutting stock problems", *INFORMS Journal on Computing*, vol. 32, no. 1, pp. 101–119, 2020.

[8] M. Dell'Amico, F. Furini, and M. Iori, "A branch-and-price algorithm for the temporal bin packing problem", *Computers & Operations Research*, vol. 114, pp. 1–16, 2020, ISSN: 0305-0548.

[9] S. Roselli, F. Hagebring, S. Riazi, M. Fabian, and K. Åkesson, "On the use of equivalence classes for optimal and sub-optimal bin covering", in *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, IEEE, 2019, pp. 1004–1009.

[10] L. V. Kantorovich, "Mathematical methods of organizing and planning production", *Management science*, vol. 6, no. 4, pp. 366–422, 1960.

[11] M. Labbé, G. Laporte, and S. Martello, "An exact algorithm for the dual bin packing problem", *Operations Research Letters*, vol. 17, no. 1, pp. 9–18, 1995.

[12] M. Peeters and Z. Degraeve, "Branch-and-price algorithms for the dual bin packing and maximum cardinality bin packing problem", *European journal of operational research*, vol. 170, no. 2, pp. 416–439, 2006.

[13] M. J. Brusco, H. F. Köhn, and D. Steinley, "Exact and approximate methods for a one-dimensional minimax bin-packing problem", *Annals of Operations Research*, vol. 206, no. 1, pp. 611–626, Jul. 2013.

[14] E. Falkenauer, "A hybrid grouping genetic algorithm for bin packing", *Journal of heuristics*, vol. 2, no. 1, pp. 5–30, 1996.

[15] A. Scholl, R. Klein, and C. Jürgens, "Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem", *Computers & Operations Research*, vol. 24, no. 7, pp. 627–645, 1997.

[16] P. Schwerin and G. Wäscher, "The bin-packing problem: A problem generator and some numerical experiments with FFD packing and MTP", *International Transactions in Operational Research*, vol. 4, no. 5-6, pp. 377–389, 1997.

[17] G. Wäscher and T. Gau, "Heuristics for the integer one-dimensional cutting stock problem: A computational study", *Operations-Research-Spektrum*, vol. 18, no. 3, pp. 131–144, 1996.

[18] J. E. Schoenfield, "Fast, exact solution of open bin packing problems without linear programming. draft", *US Army Space & Missile Defence Command, Huntsville*, vol. 20, p. 72, 2002.

[19] Gurobi Optimization, LLC, *Gurobi optimizer reference manual, version 9*, 2020.

[20] M. Delorme, M. Iori, and S. Martello, "BPPLIB: A library for bin packing and cutting stock problems", *Optimization Letters*, vol. 12, no. 2, pp. 235–250, 2018.