Solving Resource-Constrained Scheduling Problems with Exact Methods

Jordi Coll Caballero

Advisor: Enric Rodríguez Carbonell
Department of Computer Science

Co-Advisor: Josep Suy Franch
Departament d'Informàtica, Matemàtica Aplicada i Estadística
Universitat de Girona

Defense date: 4th of July of 2016

Master in Innovation and Research in Informatics
Advanced Computing

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Abstract

Scheduling problems mainly consist in finding an assignment of execution times (a schedule) to a set of activities of a project that optimizes an objective function. There are many constraints imposed over the activities that any schedule must satisfy. The most usual constraints establish precedence relations between activities, or limit the amount of some resources that the activities can consume. There are many scheduling problems in the literature that have been and are currently still being studied. A paradigmatic example is the Resource-Constraint Project Scheduling Problem (RCPSP). It consists in finding a start time for each one of the activities of a project, respecting pre-defined precedence relations between activities and without exceeding the capacity of a set of resources that the activities consume. The goal is to find a schedule with the minimum makespan (total execution time of the project). The RCPSP has many generalizations, one of which is the Multimode Resource-Constrained Project Scheduling Problem (MRCPSP). In this variation, each activity has several available execution modes that differ in the duration of the activity or the demand of resources. A solution for the MRCPSP determines the start times of the activities and also an execution mode for each one. These problems are NP-hard, and are known in the literature to be especially hard, with moderately small instances of 50 activities that are still open.

There are many approaches to solving RCPSP and MRCPSP in the literature. They are often tackled with metaheuristics due to their high complexity, but there are also some exact approaches, including Mixed Integer Linear Programming (MILP), Branch-and-Bound algorithms or Boolean Satisfiability (SAT), which have shown to be competitive and in many cases even better than metaheuristics. One of the exact methods that is growing in use in the field of constrained optimization is SAT Modulo Theories (SMT). This thesis is the continuation of previous works carried out in the Logic and Programming ($L \wedge P$) group of Universitat de Girona, which used SMT to tackle RCPSP and MRCPSP. Excluding these, there have not been any other attempts to use SMT to solve the MRCPSP. SMT solvers (like other generic methods such as SAT or MILP) do not know which is the problem they are dealing with. It is the work of the modeler to provide a representation of the problem (i.e. an encoding) in the language that the solver admits.

The main goal of this thesis is to use SMT to solve the Multimode Resource-Constraint Project Scheduling Problem. We focus on two already existing encodings for the MRCPSP, namely the *time* encoding and the *task* encoding. We use some existing preprocessing methods that contribute to the formulation of *time* and *task*, and present new preprocessings. Most of them are based on the idea of incompatibility between two activities, i.e., the impossibility that two activities run at the same time instant. These incompatibilities let us discharge some configurations of the solutions prior to encode the problem. Consequently, the use of preprocessings helps to reduce the size of the encodings in terms of variables and clauses. Another contribution of this work is the study of the *time* and *task* encodings and the differences that they present. We refine these encodings to provide more compact versions. Moreover, two new versions of these encodings are presented, which mainly differ in the codification of the constraints over the use of resources. One of them is based on Linear Integer Arithmetic expressions, and the other one in Pseudo-Boolean constraints and Integer Difference Logic. Another contribution of this work is the presentation of an ad-hoc optimization algorithm based on a linear search that mainly consists in three steps. First of all it simplifies the problem to efficiently ensure or discharge the feasibility of the instance, then it finds a first non-optimal solution by using a quick heuristic method, and finally it optimizes the problem making use of the knowledge acquired with the

preprocessings to boost the search. We also present an initial work on a more intrusive approach consisting in modifying the internal heuristic of the SMT solver for the decision of literals. This work involves the study of a state-of-the-art implementation of an SMT solver, and its modification to include a framework to specify heuristics related with the encoding of the problem. We give some initial results on custom heuristics for the *time* and *task* encodings of the MRCPSP.

Finally, we test our system with the benchmark sets of instances for the MRCPSP available in the literature, and compare our performance with a state-of-the-art exact solver for the MRCPSP. The results show that we are able to solve the major part of the benchmark sets. Moreover, we show to be competitive with the state-of-the-art solver of Vílim et. al. for the MRCPSP, being our system slower in solving the easiest benchmark instances, but outperforming the solver of Vílim et. al. in solving the hardest instances.

Contents

Part I

Introduction

Chapter 1

Introduction and Motivation

Scheduling problems mainly consist in finding an assignment of execution times (a schedule) to a set of activities of a project that optimizes an objective function. There are many constraints imposed over the activities that any schedule must satisfy. The most usual constraints establish precedence relations between activities, or limit the amount of some resources that the activities can consume. Therefore, *scheduling problem* is a generic term that includes a whole family of problems that fit the former definition. It has been and it is still a hot research topic, existing very diverse recent publications on solving scheduling problems with different approaches as we show in the state of the art in Section 4.

There are many well defined kinds of scheduling problems in the literature that have been and are currently still being studied. A paradigmatic example is the Resource-Constraint Project Scheduling Problem (RCPSP). It consists in finding a start time for each one of the activities of a project, respecting pre-defined precedence relations between activities and without exceeding the capacity of a set of resources that the activities consume. The activities are non-preemptive, what means that once they start they cannot be paused, and will be running all their duration. The resources are renewable, what means that they have a fixed capacity that is occupied in some units for an activity while it is being executed, and these units are released when the activity finishes its execution. Some examples of renewable resources are workers (an activity requires many workers), or memory for a CPU. The goal is to find a schedule with the minimum makespan (total execution time of the project). The RCPSP has many generalizations, one of which is the Multimode Resource-Constrained Project Scheduling Problem (MRCPSP). In this variation, each activity has several available execution modes. Every mode can differ in the duration of the activity and the demand over the resources. A solution for the MRCPSP determines the start times of the activities and also an execution mode for each one. Moreover, there may be non-renewable resources as well as the renewable resources. A non-renewable resource has a capacity that decreases as the activities use it, and cannot be recovered. Hence, it is needed to ensure that the overall use of these resources during the whole project is not bigger than their capacity. A budget or raw material are some examples of non-renewable resources.

These scheduling problems are NP-hard, and are known in the literature to be especially hard. There are instances of problems with projects of 50 or less activities that are still open. For this reason, there are many approaches to solving these problems in the literature. They are often tackled with metaheuristics due to their high complexity, but these approaches do not guarantee the optimality of the solutions that they find. Nevertheless, there are also some exact approaches, which have shown to be competitive and are in many cases even better than metaheuristics. Moreover, in the case of exact methods, the optimality of the solutions is mathematically proven.

Most of them are generic solving methods that offer a language to model constraint satisfaction and optimization problems. We can find approaches in the literature that tackle the RCPSP and the MRCPSP with, among others, Mixed Integer Linear Programming (MILP), branch-and-bound algorithms or Boolean satisfiability (SAT). One of the exact methods that is growing in use in the field of constrained optimization is SAT Modulo Theories (SMT), which is a generalization of SAT (satisfiability of Boolean propositional formulas), which allows to include expressions of a background theory in the formulas. Some of the most common theories are Linear Arithmetic or Uninterpreted Functions.

In this thesis we use SMT to tackle the MRCPSP problem. It has been done in collaboration with the Logic and Programming ($L \wedge P$) research group of Universitat de Girona, which have some previous work on solving scheduling problems with SMT ([3], [37]). Excluding this, and as far as we know, there have not been any other attempts to solve the RCPSP family of problems using SMT. We focus on the study and refinement of two different already existing SMT formulations of the MRCPSP, namely *time* and *task*. The contributions of this work can be summarized as follows:

1. We introduce new preprocessings that let us know some properties of the projects to schedule, and simplify the encodings by using this knowledge. By doing it, we are able to substantially reduce the sizes of the encodings, both in number of variables and number of constraints, and also truncating the search space.

2. We provide two new alternative versions of the *task* and *time* SMT encodings, one of them based on Linear Integer Arithmetic and the other one in Pseudo-Boolean constraints.

3. We propose algorithms to guide the optimization process by using the knowledge acquired with the preprocessings.

4. Finally, we explore a more intrusive approach consisting on the modification of the internal solving process of the SMT solver, concretely the heuristic of the decision of variables, to follow a user-defined criterion related with the given encoding.

The remaining of this document is structured as follows. In the second part, *II Antecedents*, we present all the basic knowledge related with this thesis. In Chapter 2, we state the formal definition of one of the paradigmatic scheduling problems, which is the RCPSP, and we also introduce the problem that we tackle in this thesis, which is the MRCPSP. We also present there the different benchmark sets that are currently being used for the MRCPSP. Chapter 3 contains an insight on Satisfiability Modulo Theories, presenting the basics on modelling language and solving methods. In Chapter 4 we present the state of the art on solving the MRCPSP, with special emphasis on a previous system based on SMT, whose techniques are reused in this thesis. The third part, *III Development and evaluation of the proposal*, exposes all the new contributions of this thesis and its results. First of all, and having presented the basics on scheduling problems and SMT solving, we expose in Chapter 5 the detailed goals of the thesis. In Chapter 6 we describe the settings of the different experiments contained in this document. In Chapter 7 we present new preprocessings and evaluate the impact that they have in solving different instances. Chapter 8 contains new variations for the *time* and *task* encodings for the MRCPSP. In Chapter 9 we present a generic ad-hoc optimization algorithm, and study how the use of preprocessings can be used to provide information to the SMT solver as we get close to the optimum makespan, and speedup this process. In Chapter 10 we present a framework to provide user-defined heuristics for the decision of variables to the SMT solver, and use it to study the performance of *time* and *task* encodings with several heuristics. Chapter 11 contains the time results of our system on different benchmark sets, and the comparison with the state-of-the-art solver for the MRCPSP. We finally present the conclusions of this thesis in Chapter 12.

Part II

Antecedents

Chapter 2

Problem Definition

2.1 RCPSP

The RCPSP is defined by a tuple (V, p, E, R, B, b) where:

- $V = \{A_0, A_1, \ldots, A_n, A_{n+1}\}$ is a set of *activities*. A_0 and A_{n+1} are dummy activities representing by convention, the *starting* and the *finishing* activities respectively. The set of non-dummy activities is defined by $A = \{A_1, \ldots, A_n\}$.
- $p \in \mathbb{N}^{n+2}$ is a vector of durations. p_i denotes the duration of activity i, with $p_0 = p_{n+1} = 0$ and $p_i > 0, \forall i \in \{1, \ldots, n\}$.
- E is a set of pairs representing precedence relations. Thus $(A_i, A_j) \in E$ means that the execution of activity A_i must precede that of activity A_j, i.e., activity A_j must start after activity A_i has finished. We assume that we are given a precedence activity-on-node graph $G = (V, E)$ that contains no cycles; otherwise the precedence relation is inconsistent. Since precedence is a transitive binary relation, the existence of a path in G from the node i to node j means that activity i must precede activity j. We assume that E is such that A_0 is a predecessor of all other activities and A_{n+1} is a successor of all other activities.
- $R = \{R_1, \ldots, R_m\}$ is a set of m *renewable resources*.
- $B \in \mathbb{N}^m$ is a vector of *resource availabilities*. B_k denotes the available amount of each resource R_k.
- $b \in \mathbb{N}^{(n+2)\times m}$ is a matrix of *demands* of activities for resources. The value $b_{i,k}$ represents the amount of resource R_k used during the execution of A_i. Note that $b_{0,k} = 0$, $b_{n+1,k} = 0$ and $b_{i,k} \geq 0$, $\forall i \in \{1, \ldots, n\}$, $\forall k \in \{1, \ldots, m\}$.

A *schedule* is a vector $S = (S_0, S_1, \ldots, S_n, S_{n+1})$ where S_i denotes the start time of each activity $A_i \in V$. We assume that $S_0 = 0$. A solution to an RCPSP instance is a non-preemptive (an activity cannot be interrupted once it is started) schedule S of minimal makespan S_{n+1} subject to the precedence and resource constraints:

$$\text{minimize } S_{n+1} \tag{2.1}$$

subject to:

$$S_j - S_i \geq p_i \qquad \forall (A_i, A_j) \in E \tag{2.2}$$

$$\sum_{A_i \in \mathcal{A}_t} b_{i,k} \leq B_k \qquad \forall B_k \in B, \forall t \in H \tag{2.3}$$

where $\mathcal{A}_t = \{A_i \in A \mid S_i \leq t < S_i + p_i\}$ represents the set of non-dummy activities in process at time t, the set $H = \{0, \ldots, T\}$ is the scheduling horizon, and T (the length of the scheduling horizon) is an upper bound for the makespan. A schedule S is feasible if it satisfies the generalized precedence constraints (2.2) and the resource constraints (2.3).

2.2 MRCPSP

The MRCPSP is a generalization of the RCPSP. It is defined by a tuple (V, M, p, E, R, B, b) where :

- $V = \{A_0, A_1, \ldots, A_n, A_{n+1}\}$ is a set of *activities*. Activities A_0 and A_{n+1} are dummy activities representing, by convention, the start and the end of the schedule, respectively. The set of non-dummy activities is defined by $A = \{A_1, \ldots, A_n\}$.
- $M \in \mathbb{N}^{n+2}$ is a vector of naturals, being M_i the number of *mode*s that activity i can execute, with $M_0 = M_{n+1} = 1$ and $M_i \geq 1, \forall A_i \in A$.
- p is a vector of vectors of naturals, being $p_{i,o}$ the *duration* of activity i using mode o, with $1 \leq o \leq M_i$. For the dummy activities, $p_{0,1} = p_{n+1,1} = 0$, and $p_{i,o} > 0, \forall A_i \in A, 1 \leq o \leq M_i$.
- E is a set of pairs of activities representing precedence relations. Concretely, $(A_i, A_j) \in E$ iff the execution of activity A_i must precede that of activity A_j, i.e., activity A_j must start after activity A_i has finished.

  We assume that we are given a precedence activity-on-node graph $G = (V, E)$ that contains no cycles, since otherwise the precedence relation is inconsistent. We assume that E is such that A_0 is a predecessor of all other activities and A_{n+1} is a successor of all other activities.
- $R = \{R_1, \ldots, R_{v-1}, R_v, R_{v+1}, \ldots, R_q\}$ is a set of resources. The first v resources are renewable, and the last $q - v$ resources are non-renewable.
- $B \in \mathbb{N}^q$ is a vector of naturals, being B_k the available amount of each resource R_k. The first v resource availabilities correspond to the renewable resources, while the last $q - v$ ones correspond to the non-renewable resources.
- b is a matrix of naturals corresponding to the *resource demands* of activities per mode. The value $b_{i,k,o}$ represents the amount of resource R_k used during the execution of activity A_i in mode o. Note that $b_{0,k,1} = 0$ and $b_{n+1,k,1} = 0, \forall k \in \{1, \ldots, q\}$.

A *schedule* is a vector of naturals $S = (S_0, S_1, \ldots, S_n, S_{n+1})$ where S_i denotes the start time of activity A_i. We assume that $S_0 = 0$. A schedule of modes is a vector of naturals $\mathbb{SM} = (\mathbb{SM}_0, \mathbb{SM}_1, \ldots, \mathbb{SM}_n, \mathbb{SM}_{n+1})$ where $\mathbb{SM}_i$, satisfying $1 \leq \mathbb{SM}_i \leq M_i$, denotes the mode of each activity A_i. A solution to an MRCPSP instance is a *schedule of modes* $\mathbb{SM}$ and a *schedule* S of minimal makespan S_{n+1}. The MRCPSP can hence be formulated as

$$\text{Minimize } S_{n+1} \tag{2.4}$$

subject to the following precedence and resource constraints:

$$\begin{aligned}(\$\mathbb{M}_i = o) \rightarrow (S_j - S_i \geq p_{i,o}) \\ \forall (A_i, A_j) \in E, \forall o \in \{1, \ldots, M_i\}\end{aligned} \tag{2.5}$$

$$\left(\sum_{A_i \in A} \sum_{o \in \{1,\ldots,M_i\}} ite(\$\mathbb{M}_i = o;\, b_{i,k,o};\, 0) \right) \leq B_k \quad \forall R_k \in \{R_{v+1}, \ldots, R_q\} \tag{2.6}$$

$$\left(\sum_{A_i \in A} \sum_{o \in \{1,\ldots,M_i\}} ite\left((\$\mathbb{M}_i = o) \wedge (S_i \leq t) \wedge (t < S_i + p_{i,o});\, b_{i,k,o};\, 0\right) \right) \leq B_k \quad \forall R_k \in \{R_1, \ldots, R_v\}, \forall t \in H \tag{2.7}$$

where $ite(c;\, e_1;\, e_2)$ is an *if-then-else* expression denoting e_1 if c is true and e_2 otherwise, $H = \{0, \ldots, T\}$ is the scheduling horizon, and T (the length of the scheduling horizon) is an upper bound for the makespan.

We also have to force the execution mode to be correct:

$$\$\mathbb{M}_i \geq 1 \qquad \forall A_i \in A \tag{2.8}$$

$$\$\mathbb{M}_i \leq M_i \qquad \forall A_i \in A \tag{2.9}$$

A schedule S is feasible if it satisfies the precedence constraints (2.5), the non-renewable resource constraints (2.6), the renewable resource constraints (2.7) and the execution mode correctness constraints (2.8) and (2.9). Figure 2.1 shows an example of an MRCPSP instance and solution that we will use as a running example to introduce some concepts.

Hence, the main differences with respect to the RCPSP are that MRCPSP include non-renewable resources, and that each activity has several execution modes.

An MRCPSP instance has a feasible schedule if and only if the following conditions hold:

- There not exist a cycle in the precedence graph (i.e., an activity is forced to start after itself finishes).
- There not exist any activity whose demand over a resource in all execution modes is greater than its capacity.
- There exist a schedule of modes such that all the non-renewable resource constraints (2.6) are satisfied.

The first two conditions are typically satisfied in the different benchmark set of instances available in the literature because they can be easily verified and therefore instances with these sources of infeasibility are not worth to study. This is not the case for the third condition, which introduces a combinatorial component to the problem. For this reason, in contrast with the RCPSP, there are benchmark sets for the MRCPSP containing infeasible instances; see Section 2.3 for more details.

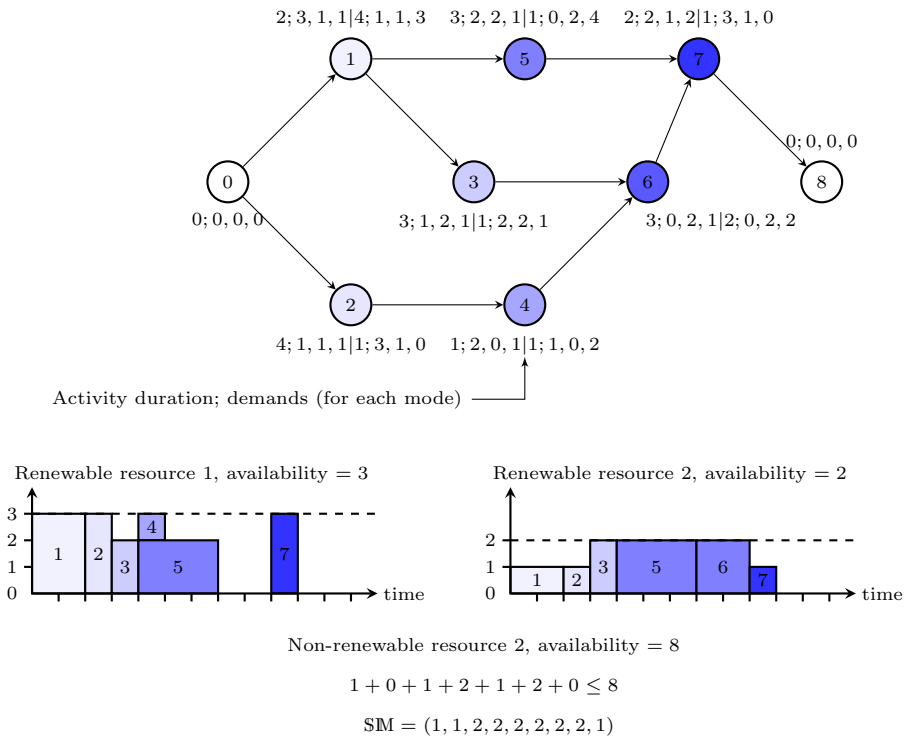

Figure 2.1: An MRCPSP instance of 7 non-dummy activities, two renewable resources and one non-renewable resource. The graph represents the activity precedences. Each node is an activity, and the numbers near the node express, for each execution mode (separated by |), its duration and the demand of the resources. Under the graph there is represented a solution, where the Gantt diagrams show the use of the renewable resources at all times, as well as the execution times of the activities.

2.3 Benchmark Instances of Scheduling Problems

Most of the works on MRCPSP in the literature [41, 40, 10] evaluate the performance of their systems using the instances available of PSPLib [25]. PSPLib is a library of benchmark sets of instances for scheduling problems which contains, among others, sets for the RCPSP and the MRCPSP. The datasets are publicly available at its web site `www.om-db.wi.tum.de/psplib/`. Regarding the MRCPSP, it contains the sets of instances described in Table 2.1.

The set j30 set will be used as a training set in this thesis, because it contains variety of instances in what regards to their hardness (it has many soft instances and at the same time is the only one in PSPLib with open instances), and also contains both feasible and infeasible instances. Will treat feasible and infeasible instances independently in many cases. From now on, we will refer to *j30SAT* as the subset of j30 that contains all the feasible instances, and to *j30UNSAT* as the subset containing all the infeasible instances. j30SAT contains 552 instances, and j30UNSAT contains 88 instances.

There is also MMLIB [39], a more recent repository of benchmark datasets for scheduling problems, including MRCPSP. They are publicly available at `http://www.projectmanagement.ugent.be/research/data/RanGen`. There are the following three sets:

set	instances	activities	modes	renewable res.	non-ren. res.
j10	537	10	3	2	2
j12	547	12	3	2	2
j14	551	14	3	2	2
j16	550	16	3	2	2
j18	552	18	3	2	2
j20	554	20	3	2	2
j30	640	30	3	2	2
m1	640	16	1	2	2
m2	481	16	2	2	2
m4	555	16	4	2	2
m5	558	16	5	2	2
r1	553	16	3	1	2
r3	552	16	3	3	2
r4	557	16	3	4	2
r5	546	16	3	5	2
n0	470	[10-20]	3	2	0
n1	637	16	3	2	1
n3	600	16	3	2	3
c15	551	16	3	2	2
c21	552	16	3	2	2

Table 2.1: Benchmark sets of PSPLib

MMLIB50 540 instances with 50 activities, each one with 3 execution modes. There are 2 renewable resources and 2 non-renewable resources.

MMLIB100 540 instances with 100 activities, each one with 3 execution modes. There are 2 renewable resources and 2 non-renewable resources.

MMLIB+ 3240 instances with 50 or 100 activities, each one with 3, 6 or 9 execution modes. The number of renewable and non-renewable resources are 2 or 4.

These instances are harder than the ones in PSPLib, and they are more recent. There are still not many works that solve them, but we can find some that use non-exact methods [20, 11].

Chapter 3

Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is a generalization of Boolean satisfiability. An SMT formula is a Boolean formula in which, in addition to Boolean variables, there can also occur predicates with predefined interpretations from background theories. The following is an example of SMT formula that includes the theory of Linear Integer Arithmetic:

$$(p \vee q) \wedge (\neg p \vee x \leq y) \wedge (x > 3 \vee y > 3)$$

where p and q are Boolean variables, and x and y are integer variables. The following is some basic nomenclature of SMT formulas:

- An *atom* is a predicate of the theory. In the previous formula, there appear the atoms $x \leq 3$, $x > 3$ and $y > 3$.
- A *literal* is an occurrence of a Boolean variable or an atom in a formula, or an occurrence of their negation. In our example, p, q, $\neg p$, $x \leq 3$, $x > 3$ and $y > 3$ are literals.
- A formula is in *Conjunctive Normal Form* (CNF) if it is a conjunction of *clauses*, which are disjunctions of literals. The previous formula is in CNF, and its clauses are $(p \vee q)$, $(\neg p \vee x \leq y)$ and $(x > 3 \vee y > 3)$.

The most common theories of the predicates appearing in SMT formulas are linear real or integer arithmetic, arrays, bit vectors, uninterpreted functions, or combinations of them. The expressibility of this language makes SMT a very good approach to model Constraint Satisfaction Problems. SMT is a good option as well taking into account efficiency, since current SMT solvers have shown to be very competitive with other *model-and-solve* exact approaches such as SAT or MILP. In Section 3.1 we make an overview on the basics on SAT / SMT solvers, and we introduce in Section 3.2 the theories of Linear Integer Arithmetic and Difference Logic, which are going to be used in this thesis.

3.1 SAT/SMT Solving

Current SMT solvers are based on the DPLL [13, 12] procedure for SAT solving. It is a procedure that can be modelled by a transition relation over states [29]. A state is either *FailState* or a pair $M \parallel \phi$, where ϕ is a finite set of clauses and M is a *partial assignment* (in the form of a sequence of literals). Some literals l in M will be annotated as being decision literals; these are

the ones added to M by the *Decide* rule, and are written l^d. The transition relation is defined by means of rules.

The classical DPLL transition system consists of the following five rules:

UnitPropagate :

$$M \parallel \phi, C \vee l \implies Ml \parallel \phi, C \vee l \quad \text{if} \quad \begin{cases} M \models \neg C \text{ and} \\ l \text{ is undefined in } M. \end{cases}$$

PureLiteral :

$$M \parallel \phi \implies Ml \parallel \phi \quad \text{if} \quad \begin{cases} l \text{ occurs in some clause of } \phi, \\ \neg l \text{ occurs in no clause of } \phi \text{ and} \\ l \text{ is undefined in } M. \end{cases}$$

Dedide :

$$M \parallel \phi \implies Ml^d \parallel \phi \quad \text{if} \quad \begin{cases} l \text{ or } \neg l \text{ occurs in some clause of } \phi \text{ and} \\ l \text{ is undefined in } M. \end{cases}$$

Fail :

$$M \parallel \phi, C \implies \mathit{FailState} \quad \text{if} \quad \begin{cases} M \models \neg C \text{ and} \\ M \text{ contains no decision literals.} \end{cases}$$

Backtrack :

$$Ml^dN \parallel \phi, C \implies M\neg l \parallel \phi, C \quad \text{if} \quad \begin{cases} Ml^dN \models \neg C \text{ and} \\ N \text{ contains no decision literals.} \end{cases}$$

The *PureLiteral* rule is usually used as a preprocessing step. Then, the evolution of the transition system follows these basic steps:

1. Apply *UnitPropagate* while possible.
2. If M contains all the variables, it is a complete assignment (or *model*) of the formula, i.e. a solution, and the procedure halts.
3. If we can apply $Fail$, the formula is unsatisfiable, and the procedure halts.
4. If we can apply *Backtrack*, we have found a *conflict*. We apply the *Backtrack* rule and return to step 1.
5. We apply the *Decide* rule, and return to step 1.

Most of modern DPLL algorithms replace the *Backtrack* rule by the *Backjump* rule and add three new rules:

- the *Learn* rule which implements the so-called Conflict-Driven Clause-Learning (CDCL). After a conflict is encountered, an explanation for it (a *lemma*) is learnt, in the form of a new clause.
- the *Forget* rule that is used to forget learned clauses (usually for reasons of space)
- the *Restart* rule that restarts the procedure but remembering what has been learned

Backjump :

$$Ml^d N \parallel \phi, C \implies Ml' \parallel \phi, C \quad \text{if} \quad \begin{cases} Ml^d N \models \neg C \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \phi, C \models C' \vee l' \text{ and } M \models \neg C', \\ l' \text{ is undefined in } M, \text{ and} \\ l' \text{ or } \neg l' \text{ occurs in } \phi \text{ or in } Ml^d N. \end{cases}$$

Learn :

$$M \parallel \phi \implies M \parallel \phi, C \quad \text{if} \quad \begin{cases} \text{all atoms of } C \text{ occur in } \phi \text{ or in } M \text{ and} \\ \phi \models C. \end{cases}$$

Forget :

$$M \parallel \phi, C \implies M \parallel \phi \quad \text{if} \quad \phi \models C.$$

Restart :

$$M \parallel \phi \implies \emptyset \parallel \phi.$$

Solvers using CDCL are often referred to as CDCL solvers. The following is an example of application of the DPLL rules on the formula $(\neg x_1 \vee x_2) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_5 \vee \neg x_6) \wedge (x_6 \vee \neg x_5 \vee \neg x_2)$, were we underline the clause causing the *Backjump* or *UnitPropagation*:

$\emptyset$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*Decide*
x_1^d	$\parallel$	$\underline{\neg x_1 \vee x_2}, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*UnitPropagate*
$x_1^d\ x_2$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*Decide*
$x_1^d\ x_2\ x_3^d$	$\parallel$	$\neg x_1 \vee x_2, \underline{\neg x_3 \vee x_4}, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*UnitPropagate*
$x_1^d\ x_2\ x_3^d\ x_4$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*Decide*
$x_1^d\ x_2\ x_3^d\ x_4\ x_5^d$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \underline{\neg x_5 \vee \neg x_6}, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*UnitPropagate*
$x_1^d\ x_2\ x_3^d\ x_4\ x_5^d\ \neg x_6$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, \underline{x_6 \vee \neg x_5 \vee \neg x_2}$	$\Rightarrow$	*Backjump & Learn*
$x_1^d\ x_2\ \neg x_5$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2, \neg x_2 \vee \neg x_5$	$\Rightarrow$	*Decide*
$x_1^d\ x_2\ \neg x_5\ x_3^d$	$\parallel$	$\neg x_1 \vee x_2, \underline{\neg x_3 \vee x_4}, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2, \neg x_2 \vee \neg x_5$	$\Rightarrow$	*UnitPropagate*
$x_1^d\ x_2\ \neg x_5\ x_3^d\ x_4$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2, \neg x_2 \vee \neg x_5$	$\Rightarrow$	*Decide*
$x_1^d\ x_2\ \neg x_5\ x_3^d\ x_4\ \neg x_6^d$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2, \neg x_2 \vee \neg x_5$	$\Rightarrow$	**SOLUTION**

In [29], we can find an adaptation to SMT of the DPLL procedure, called DPLL(T), where T stands for the parametrization on a theory. Similarly, SMT solvers using CDCL are referred to as CDCL(T) solvers.

We will not enter in detail into CDCL(T), but basically it follows the following mechanism. There is a previous step that converts an SMT formula to a SAT formula, introducing new Boolean variables in substitution of the atoms. For instance, the SMT formula:

$$(p \vee q) \wedge (\neg p \vee x \leq y) \wedge (x > 3 \vee y > 3)$$

would be substituted by the Boolean formula:

$$(p \vee q) \wedge (\neg p \vee b_1) \wedge (b_2 \vee b_3)$$

where b_1, b_2 and b_3 are fresh Boolean variables. Then, a mapping is constructed relating atoms and their corresponding Boolean variables:

$$b_1 \leftrightarrow x \leq y$$

$$b_2 \leftrightarrow x > 3$$

$$b_3 \leftrightarrow y > 3$$

In order to solve this system, a modified version of the DPLL procedure is applied to solve the Boolean formula, while interacting with a *T-solver* (theory solver), which is a module that handles the theory expressions, and is able to check the consistency of Boolean assignments with respect to the theory, detect conflicts, and produce corresponding explanations. We refer the reader to [29] for further details of the CDCL(T) procedure.

The *Decide* rule is flexible in the sense that it does not require any particular literal to be decided, the only condition is that it is not in the partial assignment. State-of-the-art SAT / SMT solvers use the *Variable State Independent Decay* (VSID) heuristic to choose the literal to decide [28]), which was engineered for conflict driven solvers. In Chapter 10 we enter in more detail in the decision of literals, and study the behaviour of alternative heuristics for the MRCPSP.

3.2 Linear Integer Arithmetic

The theory of Linear Integer Arithmetic (LIA) includes expressions of the form:

$$a_1 \cdot x_1 + \cdots + a_n \cdot x_n \# b$$

where $x_1, \ldots, x_n$ are integer variables, $a_1, \ldots, a_n$ are their coefficients, b is a constant term, and $\# \in \{<, \leq, =, >, \geq\}$. The interpretations of these expressions follow the arithmetic rules. This theory suits very well to our problem, as we can see in the *time* and *task* formulations of Section 4.3. Current theory solvers handling this theory are based on the Simplex method, concretely on the solver introduced in [19].

There is a specialization of LIA called Difference Logic (DL). In DL, the expressions are restricted to have the form:

$$x - y \leq c$$

where x and y are variables (integer variables in Integer DL), and c is a constant. Most of SMT solvers offer specific theory solvers for DL, based on the Bellman-Ford algorithm or the Floyd-Warshall algorithm, which are often more efficient than the Simplex method of general LIA theory solvers.

Chapter 4

State of the Art

The RCPSP and its variants have been widely studied in the literature. An insight to the problems, their characteristics, and different solving methods can be found in [4]. These problems are usually tackled with meta-heuristics due to their hardness. For the MRCPSP, many approaches have been proposed, including simulated annealing [8, 36], genetic algorithms [21, 2, 42], biased random sampling approaches [16], or neighbourhood search [20]. Nevertheless, exact approaches have shown to be also competitive for scheduling problems. There are many works for the RCPSP based on Constraint Programming (CP) [27, 6], Boolean satisfiability(SAT) [22], Satisfiability Modulo Theories (SMT) [3], Mixed Integer Linear Programming [26], branch and bound algorithms [15] and Lazy Clause Generation [33, 32]. Also there have been recent exact approaches to solve the MRCPSP, based on MILP [10], SMT [37], and branch an bound [40]. The latter has shown to be the state-of-the art in exact solving for the MRCPSP, by implementing a conflict-driven branching heuristic that resembles the one used by SAT/SMT solvers.

In this thesis we specially revisit, reuse and extend some of the work that has been published in [37, 7]. The following sections in this chapter expose the already existing techniques in the literature which this thesis builds upon. Section 4.1 contains a set of preprocessing techniques for the MRCPSP. Section 4.2 introduces a heuristic quick method to find initial solutions that is going to be used to set an upper bound of the makespan. Section 4.3 exposes the mentioned *time* and *task* encodings for this problem. Section 4.5 briefly describes what are Pseudo-Boolean constraints and BDDs, which are going to be used in this thesis.

4.1 Preprocessings

There are some classical preprocessing steps that are used by most of the solvers for scheduling problems. A good insight on these techniques can be found in [4]. We introduce the ones that we are going to use in Sections 4.1.1, 4.1.2, 4.1.3 and 4.1.4. In Section 4.1.5 we explain a new preprocessing technique that was firstly introduced in [37].

4.1.1 Extended Precedence Set

Since a precedence is a transitive relation, we can compute a lower bound on the time between each pair of activities in E. For this calculation it can be used the Floyd-Warshall algorithm on the graph defined by the precedence relation E, where each arc (A_i, A_j) is labelled with the duration $\min_{o \in \{1,\dots,M_i\}}(p_{i,o})$. This extended precedence set is named E^* and contains, for each pair of activities A_i and A_j such that A_i precedes A_j, a tuple of the form $(A_i, A_j, l_{i,j})$ where $l_{i,j}$

is the length of the longest path from A_i to A_j. Note that this longest path length is minimal with respect to the different activity modes. Note also that, if $(A_i, A_i, l_{i,i}) \in E^*$ for some A_i and $l_{i,i} > 0$, then there is a cycle in the precedence relation and therefore the problem instance is inconsistent.

4.1.2 Lower Bound

A lower bound LB for the makespan is a lower bound for the start time of activity A_{n+1}. The critical path (i.e. the maximum length path) between the initial activity A_0 and the final activity A_{n+1} in the graph gives such a lower bound. Note that we can easily know the length of this path if we have already computed the extended precedence set, since it corresponds to the value $l_{0,n+1}$ in the tuple $(A_0, A_{n+1}, l_{0,n+1}) \in E^*$.

For instance, in the example of Figure 2.1 the critical path is $[A_0, A_1, A_3, A_6, A_7, A_8]$ with modes $1, 1, 2, 2, 2, 1$, respectively, and its length is 6. Hence, we have $LB = 6$.

4.1.3 Upper Bound

An upper bound UB for the makespan is an upper bound for the start time of activity A_{n+1}. There is a *trivial upper bound* equal to the sum of the maximum duration of all activities:

$$UB = \sum_{A_i \in A} \max_{o \in \{1,\ldots,M_i\}} (p_{i,o})$$

In the example of Figure 2.1 this upper bound is 20.

4.1.4 Time Windows

We can reduce the domain of each variable S_i (start time of activity A_i), which initially is $\{0\,..\,UB - \min_{o \in \{1,\ldots,M_i\}}(p_{i,o})\}$, by computing its *time window*. The time window of activity A_i will be $[ES_i, LS_i]$, being ES_i its *earliest start time* and LS_i its *latest start time*. To compute the time window we use the lower and upper bound and the extended precedence set, as follows. For activities $A_i, 0 \leq i \leq n$,

$$\begin{aligned} ES_i &= l_{0,i} && \text{if} \quad (A_0, A_i, l_{0,i}) \in E^* \\ LS_i &= UB - l_{i,n+1} && \text{if} \quad (A_i, A_{n+1}, l_{i,n+1}) \in E^* \end{aligned}$$

and, for activity A_{n+1},

$$ES_{n+1} = LB \qquad LS_{n+1} = UB$$

Notice that the size of the time windows depends on a given UB. For instance, in the example of Figure 2.1, activity A_4 has time window $[1, 16]$ with the trivial $UB = 20$.

Based on the time window, we can define for an activity A_i its *earliest completion time* and its *latest completion time*, which consider the minimum and maximum durations respectively among the different execution modes:

$$\begin{aligned} EC_i &= ES_i + \min_{1 \leq o \leq M_i} p_{i,o} && \forall A_i \in V \\ LC_i &= LS_i + \max_{1 \leq o \leq M_i} p_{i,o} && \forall A_i \in V \end{aligned}$$

4.1.5 Non-Renewable Resource Demand Reduction

This preprocessing step consists in reducing the demand of non-renewable resources in a sound way. As we will see, this will allow us to save SMT literals in the constraints related to those kind of resources.

Let us introduce it through an example. In the example of Figure 2.1, the non-renewable resource R_3 has 8 units available and activity A_6 has two modes: mode 1 requires 1 unit of resource R_3, while mode 2 requires 2 units of the same resource. This problem can be transformed into an equivalent one, where the availability of resource R_3 is 7, and activity A_6 has a demand of 0 units of resource R_3 in mode 1, and of 1 unit in mode 2. Since in mode 1 the demand is of 0 units, it is not necessary to add any literal considering this mode in the constraints on non-renewable resources. Roughly, following the example, what could be done is to subtract from the availability of resource R_3, and from the different demands of activity A_6 for resource R_3 in each mode, the minimum amount of resource R_3 that activity A_6 needs. However, one could go one step further and, instead of subtracting the minimum demand value, subtract the demand value which most frequently occurs. This of course will lead to negative availabilities and demands. But, interestingly, this allows reducing the size of the constraints even more (since more demands become zero), while keeping soundness. Details are given below.

For this preprocessing, we construct a new vector B' of resource availabilities and a new matrix b' of resource demands, where:

- $B'_k = B_k$ and $b'_{i,k,o} = b_{i,k,o}$, $\forall k \in \{1, \ldots, v\}$, $\forall A_i \in V, \forall o \in \{1, \ldots, M_i\}$.
- For each non-renewable resource R_k and activity A_i, let $max_{k,i}$ denote the demand value for resource R_k with more occurrences in the different modes of activity A_i (and, in case of a tie, the smallest one). Then we state:

$$\begin{aligned} b'_{i,k,o} &= b_{i,k,o} - max_{k,i} && \forall A_i \in A, \forall o \in \{1, \ldots, M_i\} \\ B'_k &= B_k - \sum_{A_i \in A} max_{k,i} && \forall R_k \in \{R_{v+1}, \ldots, R_q\} \end{aligned}$$

Note that, as said, vector B' and matrix b' range now over integers instead of over naturals, i.e., they can contain some negative values. The zero $b'_{i,k,o}$ values (whose number is maximal thanks to the fact that we subtract the demand value with most occurrences) allow us to simplify the constraints on non-renewable constraints (see Equation 4.10 below).

4.2 PSS Heuristic

In [3], the authors proposed to use a fast heuristic method to find a schedule for the RCPSP, whose makespan serve as an upper bound for the optimum makespan. This upper bound will be in most of the cases better than the trivial upper bound. This heuristic is the *parallel scheduling generation scheme* (PSS) proposed in [23] and described in [24].

Given a project of n activities, this method requires at most n stages to find an schedule, and at each stage a subset of the activities are scheduled. Each stage s has associated a schedule time t_s (where $t_{s'} \leq t_s$, for $s' \leq s$). There are three activity sets:

- Complete set C: activities already scheduled and completed up to the schedule time t_s.
- Active set A: activities already scheduled, but still active at the schedule time t_s.

- Decision set D: activities not yet scheduled which are available for scheduling with start time t_s, w.r.t. precedence and resource constraints.

Each stage consists of two steps:

- Determining the new t_s: the earliest completion time of activities in the active set A. The activities with a finish time equal to the new t_s are removed from A and put into C. This may place new activities into D.
- One activity from D is selected with a priority rule (in [3] the activity with the smallest number) and scheduled to start at t_s, being removed from D and added to A. Then the set D is recomputed. This step is repeated until D becomes empty.

The method terminates when all activities are scheduled.

This procedure is not suitable for the MRCPSP, since it does not deal with the selection of execution modes, neither with constraints over non-renewable resources. However it has the advantage of being very fast, and serves very well to the purpose of finding a first upper bound. In Section 9.2 we will see how this method can be used for this purpose in the MRCPSP.

4.3 Encodings

In [3], two different SMT encodings for the RCPSP were presented, namely the *time* encoding and the *task* encoding. They are similar to the *time* formulation of [31] and [26] and to the *task* formulation of [30] and [32], but conveniently adapted to SMT.

In [37] the *time* and *task* encodings were adapted to the MRCPSP. The authors use the theory of Linear Integer Arithmetic, which allow to easily encode MRCPSP instances as logical combinations of arithmetic constraints. With SMT, Boolean variables and integer variables can occur together in the resulting formula, which is very interesting in terms of modeling. Some refinements were introduced considering the preprocessing steps described in Section 4.1 (extended precedences, time windows, etc.)

The set of integer variables $\{S_0, S_1, \ldots, S_n, S_{n+1}\}$ denote the start time of each activity, and S' is defined as the set $\{S_1, \ldots, S_n\}$. The schedule of modes with the set of Boolean variables $\{sm_{i,o} \mid 0 \leq i \leq n+1, 1 \leq o \leq M_i\}$, being $sm_{i,o}$ true if and only if activity A_i is executed in mode o.

The objective function is always (2.4), and there are the following constraints in both encodings:

$$S_0 = 0 \tag{4.1}$$

$$S_i \geq ES_i \qquad \forall A_i \in \{A_1, \ldots, A_{n+1}\} \tag{4.2}$$

$$S_i \leq LS_i \qquad \forall A_i \in \{A_1, \ldots, A_{n+1}\} \tag{4.3}$$

$$sm_{i,o} \rightarrow S_j - S_i \geq p_{i,o} \qquad \forall (A_i, A_j) \in E, \forall o \in \{1, \ldots, M_i\} \tag{4.4}$$

$$S_j - S_i \geq l_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{4.5}$$

$$sm_{0,1} = true \tag{4.6}$$

$$sm_{n+1,1} = true \tag{4.7}$$

$$\bigvee_{1 \leq o \leq M_i} sm_{i,o} \qquad \forall A_i \in A \tag{4.8}$$

$$\neg sm_{i,o} \vee \neg sm_{i,o'} \qquad \forall A_i \in A, 1 \leq o \leq M_i, o < o' \leq M_i \tag{4.9}$$

where (4.2) and (4.3) encode the time windows, (4.4) encodes the precedences, (4.5) encodes the extended precedences and (4.6), (4.7), (4.8) and (4.9) ensure that each activity runs in exactly one mode.

For non-renewable resources, the following constraints replace (2.6), which use the values resulting from the *non-renewable resource demand reduction* preprocessing step:

$$\left(\sum_{A_i \in A} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ b'_{i,k,o} \neq 0}} ite\left(sm_{i,o};\, b'_{i,k,o};\, 0\right) \right) \leq B'_k \qquad \forall R_k \in \{R_{v+1}, \ldots, R_q\} \tag{4.10}$$

Notice that the *ite* expression is removed in the cases where $b'_{i,k,o} = 0$.

Constraints (2.7) on renewable resources are reformulated differently in each of the two encodings, as described below.

4.3.1 Time Formulation

This is the most obvious formulation. It basically consists in stating, for every time unit t and renewable resource R_k, that the sum of demands for this resource from the different activities cannot exceed the availability of the resource.

$$\begin{aligned} sm_{i,o} \rightarrow (y_{i,t} \leftrightarrow (S_i \leq t) \wedge (t < S_i + p_{i,o})) \\ \forall A_i \in A, \forall o \in \{1,\ldots,M_i\}, \forall t \in \{ES_i, \ldots, LS_i + p_{i,o}\} \end{aligned} \tag{4.11}$$

$$\left(\sum_{A_i \in A} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ ES_i \leq t \leq LS_i + p_{i,o} - 1 \\ b'_{i,k,o} \neq 0}} ite\left(sm_{i,o} \wedge y_{i,t};\, b'_{i,k,o};\, 0\right) \right) \leq B'_k$$

$$\forall R_k \in \{R_1, \ldots, R_v\}, \forall t \in \{0, \ldots, UB\} \tag{4.12}$$

Constraints (4.11) give value to the Boolean variables $y_{i,t}$, which are true if and only if activity A_i is running at time t. Constraints (4.12) replace the (2.7) of the problem definition. The number of theses constraints is proportional to $UB + 1$, and the size of the sums is directly related to the size of the time windows (which are also dependent on UB). Therefore, the size of this encoding is highly dependent on the value of UB.

4.3.2 Task Formulation

This formulation uses variables indexed by activity number, and not by time. The key idea is that, in the non-preemptive case, checking only that there is no overload at the beginning (or end) of each activity is sufficient to ensure that there is no overload at every time unit. Hence, in this formulation the number of variables and constraints is independent of the length of the scheduling horizon.

Boolean variables $z^1_{i,j}$ denote whether activity A_i does not start after activity A_j does, Boolean variables $z^2_{i,j}$ to denote whether activity A_j starts before the end of activity A_i. The conjunction

$z^1_{i,j} \wedge z^2_{i,j}$ denotes whether activity A_i is running when activity A_j starts. The constraints are the following:

$$z^1_{i,j} \leftrightarrow true \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{4.13}$$

$$z^1_{j,i} \leftrightarrow false \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{4.14}$$

$$z^1_{i,j} \leftrightarrow S_i \leq S_j \qquad \forall A_i, A_j \in A, i \neq j \tag{4.15}$$

$$z^2_{i,j} \leftrightarrow false \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{4.16}$$

$$z^2_{j,i} \leftrightarrow true \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{4.17}$$

$$sm_{i,o} \rightarrow (z^2_{i,j} \leftrightarrow S_j < S_i + p_{i,o}) \qquad \begin{array}{l} \forall A_i, A_j \in A, \\ i \neq j, \forall o \in \{1, \ldots, M_i\} \end{array} \tag{4.18}$$

$$sm_{j,o'} \rightarrow \left(\sum_{A_i \in A \setminus \{A_j\}} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ b'_{i,k,o} \neq 0}} ite(sm_{i,o} \wedge z^1_{i,j} \wedge z^2_{i,j}; b'_{i,k,o}; 0) \right) \leq B'_k - b'_{j,k,o'}$$
$$\forall A_j \in A, \forall o' \in \{1, \ldots, M_j\}, \forall R_k \in \{R_1, \ldots, R_v\} \tag{4.19}$$

The last constraints (4.19) state, for each activity A_j, mode o' and renewable resource R_k, that the sum of resource demands for R_k at the start time of A_j must not exceed the capacity B'_k of R_k.

4.4 Optimization of the Makespan

SMT has its roots in the field of hardware and software verification, typically dealing with decision problems. For this reason, in the early uses of SMT in the field of constrained problems optimization, many ad-hoc procedures were defined to deal with optimization. One of the basic approaches consists in successively bounding the value of the objective function, until the lower bound coincides with the lower bound and therefore the optimum is found. Nowadays, there are many SMT solvers that support optimization, as is the case of *Z3* [14] or *OptiMathSAT* [35]. The usual behaviour is to combine the computation of Boolean satisfiable models with the optimization of the theory expressions.

Nevertheless, one of the purposes of this thesis is to study the relation of the encoding sizes and computation times as UB evolves. For this reason we will focus on ad-hoc optimization procedures which make independent satisfiability checks to the SMT solver and that let us guide the search of the optimum makespan. Concretely, we will use a linear search optimization scheme (see Algorithm 1), which starts from a feasible UB for the makespan (if any), and reduces it until the instance becomes infeasible.

Algorithm 1 Linear search optimization schema

Input: An MRCPSP instance INS
Output: Optimum makespan, or INFEASIBLE

```
UB ← get_upper_bound(INS)
LB ← get_upper_bound(INS)
smt_assert_encoding(ENC)
SAT ← smt_check()
if not SAT then
  return INFEASIBLE
end if
UB ← UB − 1
while SAT and UB ≥ LB do
  smt_update_upper_bound(UB)
  SAT ← smt_check()
  if SAT then
    MODEL ← smt_get_model()
    MAKESPAN ← smt_get_makespan(MODEL)
    UB ← MAKESPAN − 1
  end if
end while
if SAT then
  return UB
else
  return UB + 1
end if
```

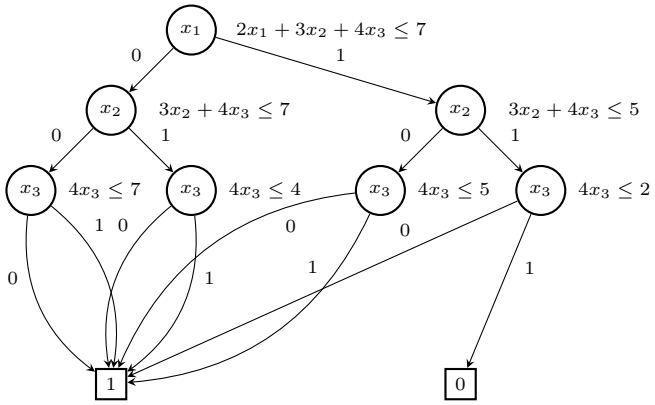

Figure 4.1: BDD for $2x_1 + 3x_2 + 4x_3 \leq 7$.

4.5 Pseudo-Boolean Constraints

As defined in [1], a Pseudo-Boolean constraint has the form:

$$a_1 x_1 + \cdots + a_n x_n \# K$$

where the a_i and K are integer coefficients, the x_i are Boolean variables, and the relation operator $\#$ belongs to $\{<, >, \leq, \geq, =\}$. A typical data structure to represent Boolean functions is a Binary Decision Diagrams (BDD) [9]. A BDD is a rooted, directed, acyclic graph, where each non-terminal (decision) node corresponds to a Boolean variable x and has two child nodes with edges representing a *true* and a *false* assignment to x. There are two terminal nodes that are the 0-terminal and the 1-terminal, representing the truth value of the formula for the assignment leading to them. A BDD is called *ordered* if the variables appear in the same order on all paths from the root. A BDD is said to be *reduced* if the following two rules have been applied to its graph until a fix point:

- Merge any isomorphic subgraphs
- Eliminate any node whose two children are isomorphic.

A *Reduced Ordered Decision Diagram* (ROBDD) is canonical (unique) for a particular function and variable order. As an example, we have in Figure 4.1 a BDD that represents the Pseudo-Boolean constraint $2x_1 + 3x_2 + 4x_3 \leq 7$, and Figure 4.2 illustrates the canonical ROBDD for the same constraint with the order $x_1 \prec x_2 \prec x_3$.

In [7], the $L \wedge P$ group implemented a framework based on the ideas presented in [1] that, given a Pseudo-Boolean Constraint, generates a SAT encoding using ROBDDs. We will use this framework in this thesis.

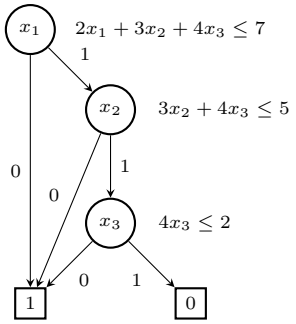

Figure 4.2: ROBDD for $2x_1 + 3x_2 + 4x_3 \leq 7$.

Part III

Development and Evaluation of the Proposal

Chapter 5

Goals of the Thesis

The main goal of this thesis is to investigate in depth the *time* and *task* SMT encodings for the MRCPSP. In order to do it, we will focus on the following specific goals:

- Design preprocessings that help us to anticipate properties of the possible schedules for an instance (Chapter 7). It means that prior to calling an SMT solver, we can anticipate the value of some variables, and therefore reduce the size of the encodings regarding number of variables, number of constraints and size of the constraints. We want to evaluate how the preprocessings help to solve each kind of instance (depending on its hardness), and also the differences presented between both encodings.

- Propose new alternatives of formulations for *time* and *task* (Chapter 8), to try to improve the performance of the state-of-the-art formulations presented in Section 4.3. We will use Linear Integer Arithmetic as theory, and we will also use Pseudo-Boolean constraints to formulate some expressions.

- Study the use of the preprocessed information, obtained either with the new preprocessings or the ones already existing in the literature, in ad-hoc optimization procedures (Chapter 9). The aim is to simplify the problem and to reduce the size of the encoding as we get close to the optimum makespan.

- Explore an approach more intrusive to the SMT solver by tuning its internal implementation (Chapter 10). This is a large field to explore, but in this thesis we have as a goal to make an initial work, consisting on the implementation of new heuristics for the selection of the variables to use in the *decide* operations of the SAT/SMT CDCL solving algorithm. We want to study the behaviour of the solver for different heuristics for both formulations. This will require a study of an state-of-the-art implementation of an SMT solver, and an extension of its functionality by adding the possibility of defining new heuristics.

Finally, in order to determine if the goals of the thesis have been achieved, all the new techniques will be collected and used to evaluate the performance of solving benchmark sets of instances, and will be compared with the performance of the state-of-the-art exact solver of the MRCPSP [40].

Chapter 6

Working Environment

In this chapter we present the working environment and experimental settings used in the thesis. Section 6.1 explains which SMT solver has been used and justifies the decision. Section 6.2 contains the experiment global settings. Finally, Section 6.3 describes the running environment that has been used.

6.1 SMT Solver

The first step prior to designing and evaluating the different proposals that appear in this work has been to decide which SMT solver is going to be used. It has been decided to use Yices 2.4.2 [18] for many reasons:

- It is an state-of-the-art SMT solver.
- The previous work of the L∧P research group in the field of scheduling problems has shown that the results obtained with this solver are competitive.
- The source code of the version 2 is, for research purposes, freely available and modification permissions are granted.
- After a first study of the code, it has been shown to be very well commented, self-documented, compact and with an intuitive structure.

The Z3 solver [14] was also considered, since it is also a state-of-the-art SMT solver and the source code is also available, but the previous work on Yices and the higher complexity of Z3's architecture lead to opt for Yices. The ease of modifying the implementation had an important weight on the decision since an important part of this thesis focuses on implementing and studying modifications on the heuristic of decision of variables.

Yices, as most of the SMT solvers, supports the SMT-LIB 2 standard for the interface of solving methods. However, since the modifications that are introduced to the solver affect the core components of the solver, in this thesis we work with Yices' own API.

6.2 Experimental Settings

The timeout for the experiments has been set to 3600 seconds to be able to evaluate the performance of the techniques in the hardest instances. In Chapters 7, 8, 9 and 10, we have used

as a training set of examples the j30 set from PSPLib, because it contains variety of instances in what regards to their hardness (it has many soft instances and at the same time is the only one in PSPLib with open instances), and also contains both feasible and infeasible instances. In fact, we will use the distinction of j30SAT and j30UNSAT made at Section 2.3.

Finally, in Chapter 11 we collect the best of our techniques, and obtain performance results for a larger collection of benchmark sets, not only limited to j30, and compare our system with the state-of-the-art solver on MRCPSP [40].

6.3 Running Environment

All the experiments contained in this thesis have been executed in an 8GB Intel® Xeon® E3-1220v2 machine at 3.10 GHz. All the chapters except for Chapter 11 focus on evaluating the impact of several proposals and their relative efficiency compared to other proposals. In order to save experimentation time and take advantage of the available resources, the experiments shown in these chapters do not use the whole capacity of the machine but run many different jobs at a time in the machine (always in different threads), and distribute the memory evenly between them. Obviously, all the experiments whose performance are compared will be assigned the same machine settings.

Limiting the capacity of the machines worsens the time performance, and at Chapter 11 we want to provide the status of our proposals with our available resources, and compare them with solvers of other authors without limiting the advantage that they may take of multiple threading (this is not the case for Yices, which does not use parallelism). For this reason, the experiments in Chapter 11 have been run with only one job at a time in each machine.

Chapter 7

Preprocessings

In Section 4.1 we presented some preprocessings that were used in previous work for solving the MRCPSP with SMT, and which will be employed in this thesis as well. In Section 7.1 we are going to introduce new preprocessings that will let us further refine the constraints and reduce the size of our encodings. Since the new preprocessings do not have the same impact on all the instances, we analyse in Section 7.2 the amount of information discovered for each preprocessing depending on the hardness of the instances. We include time improvements due to the preprocessings in Chapter 8, once we have explained how the encodings are modified to introduce these new preprocessings.

7.1 New Preprocessings

7.1.1 Extended Precedence Set: Energy Precedences

We have seen in Section 4.1.1 how to compute the extended precedence set E^*. Next we will show that the demands on renewable resources let us go one step further. Note that any activity A_k such that $(A_i, A_k, l_{i,k}) \in E^* \wedge (A_k, A_j, l_{k,j}) \in E^*$, will be completely executed in the time interval $[S_i + \min_{o \in \{1,\dots,M_i\}}(p_{i,o}), S_j]$. We can compute the set of such activities A_k for any pair of activities A_i, A_j such that $(A_i, A_j, l_{i,j}) \in E^*$. Let us define this set as $AB_{i,j}$, the set of activities between A_i and A_j:

$$AB_{i,j} = \{A_k \mid (A_i, A_k, l_{i,k}) \in E^*, (A_k, A_j, l_{k,j}) \in E^*\}$$

Returning to the running example of Figure 2.1, $AB_{1,7} = \{A_3, A_5, A_6\}$. Note that the time interval between the end of A_1 and the start of A_7 must be wide enough to run all $A_k \in AB_{1,7}$ without exceeding the availability of any renewable resource. Looking at the Gantt diagram of the solution for resource R_2, we can see that it is needed 6 units of time using the whole capacity of R_2 to run all the activities in $AB_{1,7}$, which are making the minimum demand on R_2 among their execution modes.

This necessity of having wide enough time intervals gives, for every resource $r \in \{1, \dots, v\}$, a lower bound ($RLB_{i,j,r}$) of the time difference between the end of A_i and the start of A_j:

$$RLB_{i,j,r} = \lceil \frac{1}{B_r} \cdot \sum_{A_k \in AB_{i,j}} \min_{o \in \{1,\dots,M_k\}}(p_{k,o} \cdot b_{k,r,o}) \rceil$$

$RLB_{i,j,r}$ is the minimum time difference between A_i and A_j so that the activities in $AB_{i,j}$ will not surpass the capacity of resource r at any instant. This clearly establishes a precedence relation between A_i and A_j, so we can update every time lag $l_{i,j}$ of the extended precedence set as:

$$l'_{i,j} = \max(l_{i,j}, \min_{o \in \{1,\dots,M_i\}}(p_{i,o}) + \max_{r \in \{1,\dots,v\}}(RLB_{i,j,r})) \qquad \forall (A_i, A_j, l_{i,j}) \in E^*$$

where $l'_{i,j}$ is the new value for the time lag and $l_{i,j}$ is the previous value. In the example of Figure 2.1, the original value of $l_{1,7}$ was 5, and it can be updated with this mechanism to 8.

Note that an increase of a single extended precedence can be propagated to other extended precedences in E^* for transitivity. This preprocessing resembles the energy-based reasoning used by some constraint propagators (see [4]). For this reason, we will refer to these new precedences as *energy precedences*.

As a particular case, it is worth noting that $l_{0,n+1}$, i.e., the minimum time lag between the starting activity and the finishing activity, might be increased thanks to the energy precedences, and hence giving a better lower bound LB for the optimum makespan.

7.1.2 Start Time Window Incompatibilities

Time windows can provide information regarding precedences between activities, even if there is not an (extended) precedence between them. For instance, if $LC_i \leq ES_j$, for some A_i and A_j, we know for sure that A_j will start after the completion of A_i, even if $(A_i, A_j, l_{i,j} \notin E^*$. As exposed in Section 4.3.2, the *task* encoding checks if an activity A_i is running when activity A_j starts. That encoding gives constant values to variables $z^1_{i,j}$ and $z^2_{i,j}$ if there exists an extended precedence between A_i and A_j. We can discard still more overlaps by computing an incompatibility Boolean matrix STI as:

$$STI_{i,j} \iff ES_j \geq LC_i \vee LS_j < ES_i \qquad \forall A_i, A_j \in A, i \neq j$$

where $STI_{i,j}$ equal to true means that we can ensure that activity A_i will never be running when activity A_j starts in any schedule. Notice that this matrix is not necessarily symmetric. Also, the number of start time window incompatibilities found depends on the size of the time windows, and therefore of the value of B. Looking at Figure 2.1, there is not any start time window incompatibility if we take into account the trivial $UB = 20$. However, if we consider the UB of the solution, which is 10, we have that $ES_7 = 5$, and $LS_1 = 4$, so $STI_{1,7} = true$.

7.1.3 Resource Incompatibilities

In instances with activities that have a high demand of renewable resources, it may be the case that there are pairs of activities (A_i, A_j) whose added minimum demand on a renewable resource is higher than its capacity. If that is the case, we know for sure that these activities will never be running at a same time in a feasible schedule. In this situation we will say that they are *resource incompatible*. We define the symmetric Boolean matrix RI as:

$$RI_{i,j} \iff \bigvee_{1 \leq r \leq v} \min_{1 \leq o \leq M_i} b_{i,r,o} + \min_{1 \leq o \leq M_j} b_{j,r,o} > B_r \qquad \forall A_i, A_j \in A, i \neq j$$

where $RI_{i,j}$ equal to true means that activities A_i and A_j are resource incompatible. In the example of Figure 2.1, $RI_{3,5} = true$, because activities A_3 and A_5 can never run at the same

time because they both have a minimum demand of 2 on R_2, and the capacity $B_2 = 2$ would be exceeded.

7.1.4 Disjoint Use of Renewable Resources

Remember that *task* encoding checks, for every renewable resource r, that at the start time of every activity A_j, the capacity of r is not exceeded. It does it not by checking which activities are running at a time t but which ones overlap with A_j at its start. Note that, therefore, if two activities never use the same renewable resource regardless the execution modes, we do not need to care about whether they run at a same time. We will say that the pairs of activities that satisfy this property are *resource disjoint*. This fact will let us omit some constraints and variables $z^1_{i,j}$ and $z^2_{i,j}$ related with the overlapping of resource disjoint activities. We compute the Boolean matrix of resource disjunctions D as:

$$D_{i,j} \iff \{r \mid 1 \leq r \leq v, \bigvee_{1 \leq o \leq M_i} b_{i,r,o} > 0\} \cap \{r \mid 1 \leq r \leq v, \bigvee_{1 \leq o \leq M_j} b_{j,r,o} > 0\} = \emptyset$$

where $D_{i,j}$ equal to true means that A_i and A_j are resource disjoint. In Figure 2.1, $D_{4,6} = true$, because A_4 do not demand R_2 in any mode and A_6 do not demand R_1 in any mode.

7.2 Impacts of the Preprocessings

Now we are presenting results that show how each preprocessing serves to discover information of the instances. We show in a scatter plot for each new preprocessing the amount of information discovered for each instance in the j30SAT set. In the plots, each point corresponds to an instance. Axis y contains a different measure for each preprocessing:

- Figure 7.1 shows the number of extended precedences that have been increased in E^* due to energy precedences. We count both the extended precedences that have been replaced by an energy precedence, and the extended precedences that have been increased by transitivity due to a new energy precedence.
- In Figure 7.2 we show the number of start time window incompatibilities detected. The time window incompatibilities depend on the time windows, and therefore on the UB of the encoding. We are counting them in the encoding for the optimum makespan (or the best known makespan) of each instance.
- Figure 7.3 shows the number of resource incompatibilities found, breaking the symmetries $RI_{i,j} = RI_{j,i}$.
- Figure 7.4 shows the number of resource disjunctions found, breaking the symmetries $D_{i,j} = D_{j,i}$.

Axis x, which is in logarithmic scale, represents the computation time required to solve the instances with the *time* encoding presented in Section 4.3.1 with a linear search optimization algorithm. We use this computation time as an estimation of the hardness of the instance. On the one hand, identifying the hardness ranges where the preprocessings have the greatest influence will serve us to better understand the reason of the computation time improvements in any case in Section 8.5.1. On the other hand, the amount of information obtained with a particular preprocessing is a metric that could serve as an indicator of the hardness of the instances.

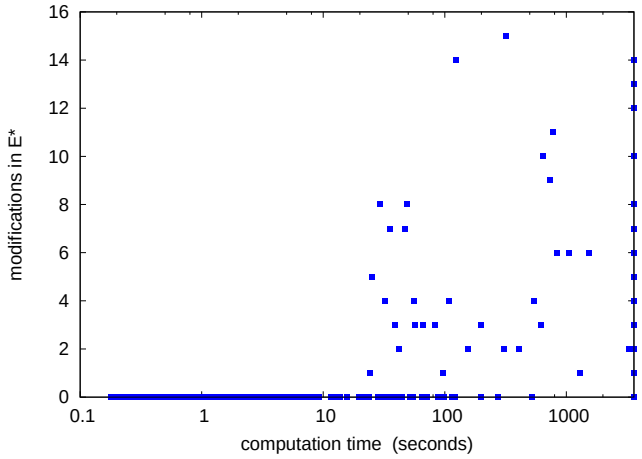

Figure 7.1: For each instance in j30SAT, number of extended precedences added or modified in the extended precedence set due to energy precedences (axis y), compared to the solving time of the instance (axis x). We count both the energy precedences and the precedences increased due to transitivity from an energy precedence. The points in the right vertical edge of the plot correspond to instances that have been not solved within the timeout of 3600 seconds.

The results show that:

- Resource incompatibilities and energy precedences are more present in hardest instances. This is specially the case for energy precedences, detected only in instances with solving time higher than 20, and detected almost in all instances of computation time higher than 500. Notice that the energy precedences are prone to occur in instances with high demands on renewable resources and narrow precedences graph.
- Resource disjunctions and start time window incompatibilities tend to appear less when the instance is harder, having zero resource disjunctions in instances with solving time greater than approximately 20. Contrarily to the energy precedences, resource disjunctions are prone to occur in instances with low demands on renewable resources and start time window incompatibilities are prone to occur in instances with wide precedences graph.

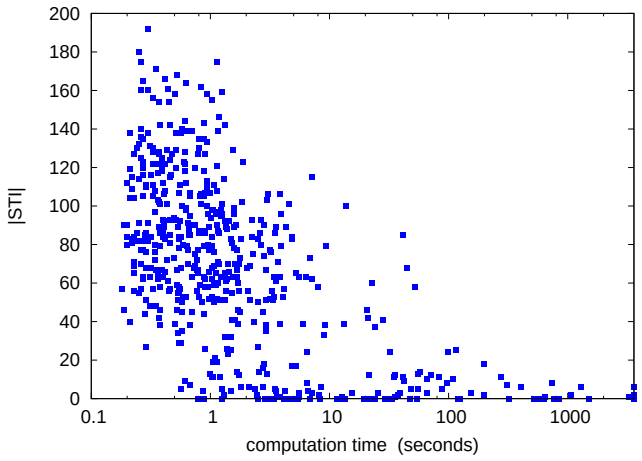

Figure 7.2: For each instance in j30SAT, number of start time window incompatibilities detected (axis y), denoted $|STI|$ in the plot, compared to the solving time of the instance (axis x).

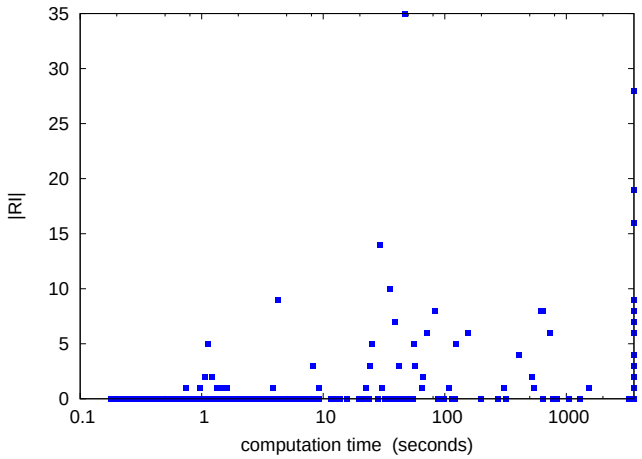

Figure 7.3: For each instance in j30SAT, number of resource incompatibilities detected (axis y), denoted $|RI|$ in the plot, compared to the solving time of the instance (axis x). The points in the right vertical edge of the plot correspond to instances that have been not solved within the timeout of 3600 seconds.

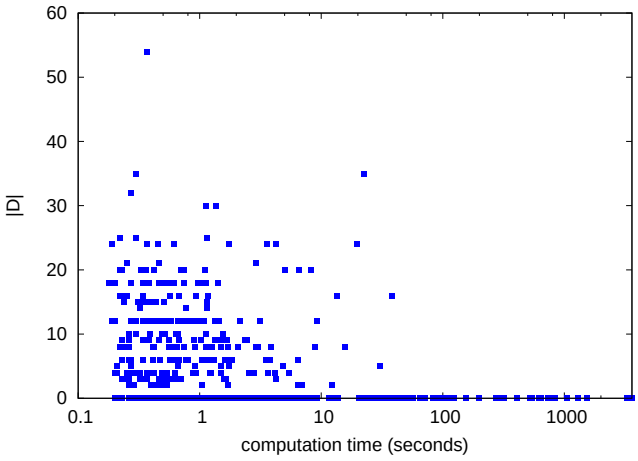

Figure 7.4: For each instance in j30SAT, number of pairs of resource disjoint activities detected (axis y), denoted $|D|$ in the plot, compared to the solving time of the instance (axis x).

Chapter 8

Encoding Study and Refinement

In this chapter we are going study and refine the *time* and *task* encodings for the MRCPSP that we have seen in Section 4.3. In Section 8.1 we analyse the properties of these encodings, making special emphasis on their dependence on the UB. The following sections incrementally introduce refinements on the original encodings: Section 8.2 explains how the new preprocessings can be used to modify the formulations of the MRCPSP, in Section 8.3 we comment how to reformulate the encodings in CNF formulas, and in Section 8.4 we formulate three different versions of *time* and *task*. Finally, Section 8.5 shows the improvements gained with each refinement.

8.1 Study of the Encodings

The main difference that the *time* and *task* encodings present is that the former encodes the constraints over the renewable resources for every time instant from 0 to a time horizon, and the later checks that these constraints are satisfied at the beginning of every activity. The former depends on an upper bound UB for the makespan, and introduces $\mathcal{O}((v + nM_{max})\,UB)$ constraints, $\mathcal{O}(n(M_{max} + UB))$ Boolean variables and $\mathcal{O}(nM_{max}\,UBv)$ integer variables[1], where $M_{max} = \max_{A_i \in A}(M_i)$. On the other hand, *task* encoding depends on the number of activities of the project, introducing $\mathcal{O}(nM_{max}(n+v))$ constraints, $\mathcal{O}(n(M_{max}+n)$ Boolean variables and $\mathcal{O}(n^2 M_{max} v)$ integer variables.

Therefore, the size of the encoding is proportional to UB in *time* and independent of it in *task*, in terms of number of constraints, number of Boolean variables and number of integer variables. Nevertheless, the domains of the integer variables $\{S_1, \ldots, S_n\}$ are proportional to the time windows, which at their turn are proportional to UB. For this reason, stating whether the performance of the *task* encoding is totally independent on the UB needs a deeper analysis.

We have run two experiments which show that *task* is indeed independent on the UB, and also serve to have an insight of the impacts that the dependency on UB has for the *time* encoding. The first one consists in, given an MRCPSP instance, multiply the durations of all the activities in all modes by a same constant factor c. Notice than the optimum makespan will increase by the same constant factor c. Then, we encode the instance as described in Section 4.3, but enforcing $UB = LB = \textit{optimum makespan}$, i.e. we do not search the optimum makespan, but having it as an input we compute a schedule for this makespan. We then make a single call to the SMT solver to compute an optimum solution, and record the computation time. Notice that in some sense

[1] In the formulation presented in 4.3, there only explicitly appear $n + 2$ integer variables, that are $\{S_0, \ldots, S_{n+1}\}$. Nevertheless, each *ite* expression introduces one fresh integer variable implicitly.

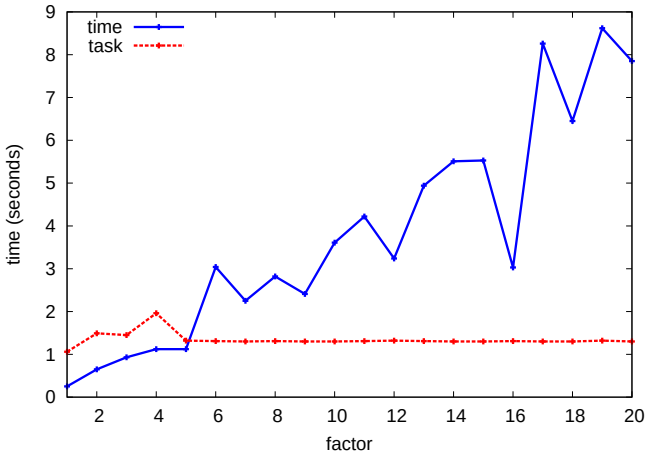

Figure 8.1: Each point corresponds to the computation of an optimum schedule for the *j3022_5* instance. Axis y contains the computation time, and axis x the factor multiplying the durations of the activities. This result is shown for both *task* and *time* encodings.

we are not modifying the nature of the instances. We could for example multiply the durations by 60, and think of it as expressing the time units in seconds instead of minutes. For this reason, the changes on the performance will mainly be due to the impact of the UB in the encoding. We do it for both *time* and *task* encodings, and multiplying the durations by the constant factors in the range $\{1, 2, \ldots, 20\}$. We have performed this experiment with many instances, and all the results go in the line of what Figure 8.1 illustrates for the instance *j3022_5*. We can see that indeed the *task* encoding has always approximately the same performance regardless the coefficient, while the time encoding gives increasing computation times.

Our second experiment analyses the evolution of the computation time as UB decreases in a linear search optimization. We have encoded the same example instance as before for all the values of UB in the range $[OPT, \ldots, 1.5 \cdot UB_T]$, and for each UB we have called the SMT solver. UB_T denotes the trivial upper bound for the makespan, and OPT denotes the optimum makespan for this instance. The results can be seen in Figure 8.2. In this case $UB_T = 252$. Notice that if the instance is feasible we can find a schedule that fits the trivial upper bound such that:

- it runs only one activity at a time, respecting the precedence constraints, and therefore the constraints over renewable resources are satisfied (assuming that there are not infeasible modes[1]).
- its schedule of modes has to respect the non-renewable resource constraints.

Moreover, any such a schedule would fit for any upper bound greater than the trivial. So, the intuition says that it should be equally easy to construct a schedule for any $UB \geq UB_T$, but the plot shows that the *time* encoding suffers from having its encoding size depending on the upper bound. Contrarily, the computation time required by *task* is always the same for these trivial

[1] A mode o of an activity A_i is said to be infeasible if the demand $b_{i,k,o}$ on some resource k is greater than its capacity B_k. Notice that there is not any feasible solution in which an activity has assigned an infeasible mode. We can search for infeasible modes and remove them in polynomial time.

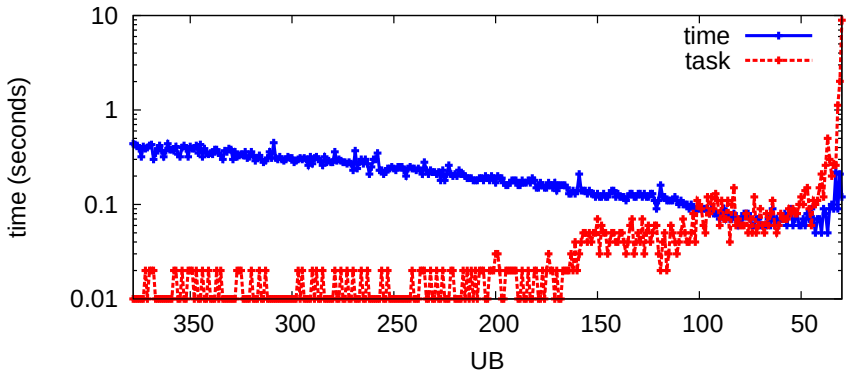

Figure 8.2: Computation time of the satisfiability check of the SMT solver for each value of UB in a range, for the instance j3022_5. Axis y is in log scale.

schedules. On the other hand, this example shows that having the performance depending on the UB can also be good, because the *time* encoding outperforms *task* in the final iterations, which are the ones with smallest UB. Finally, Figure 8.3 shows the normalized accumulated time through the evolution of a linear search optimization, from the trivial UB to the optimum makespan. It can be seen that the *time* encoding suffers from having to compute the easiest schedules due to the size of the encoding, whereas this is not the case for the *task* encoding.

8.2 Application of the New Preprocessings

The new preprocessings presented in Chapter 7 will serve us to reduce the size of the encodings. Although they do not have any impact on the formal definition of the *time* encoding, the energy precedences let us compress the time windows, and therefore the size of the constraints and the number of variables $y_{i,t}$ is reduced[1].

On the other hand, we have the new incompatibility matrices STI, RI and D that, applied to the *task* formulation, let us discharge overlaps between activities and hence reduce the size of the encoding. Let us define the Boolean function $incomp(i, j)$ as:

$$incomp(i,j) = STI_{i,j} \vee RI_{i,j} \vee D_{i,j} \vee (A_i, A_j, l_{i,j}) \in E^* \vee (A_j, A_i, l_{j,i}) \in E^* \tag{8.1}$$

which is true only if there exists no feasible schedule in which A_i is active when activity A_j starts, or A_i and A_j never consume the same renewable resource. The constraints for the *task* encoding are reformulated as:

[1]Recall that the variables $y_{i,t}$ denote whether A_i is running at time t, and are only defined for the possible time instances according to the time windows.

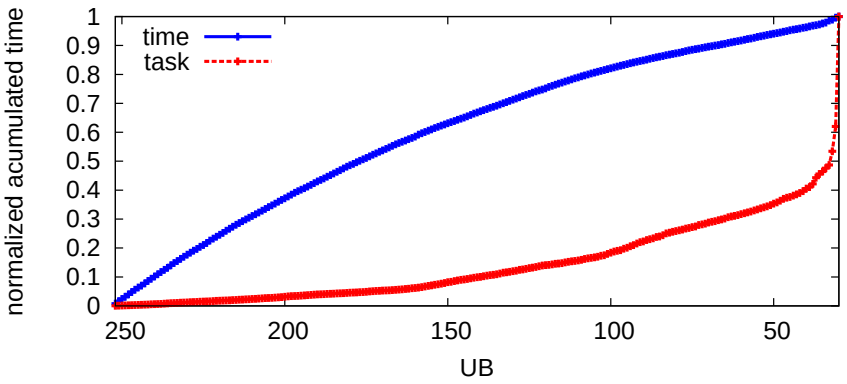

Figure 8.3: Normalized accumulated computation time of a linear search optimization for the instance j3022_5.

$$z^1_{i,j} \leftrightarrow S_i \leq S_j \qquad \forall A_i, A_j \in A, \neg incomp(i,j), i \neq j \tag{8.2}$$

$$sm_{i,o} \rightarrow (z^2_{i,j} \leftrightarrow S_j < S_i + p_{i,o}) \qquad \begin{array}{l} \forall A_i, A_j \in A, \neg incomp(i,j), \\ i \neq j, \forall o \in \{1,\ldots,M_i\} \end{array} \tag{8.3}$$

$$(sm_{i,o} \wedge sm_{j,o'}) \rightarrow (S_j \geq S_i + p_{i,o} \vee S_i \geq S_j + p_{j,o'}) \qquad \begin{array}{l} \forall A_i, A_j \in A, \\ RI_{i,j} = true, \\ STI_{i,j} = false, \\ \forall o \in \{1,\ldots,M_i\}, \\ \forall o' \in \{1,\ldots,M_j\} \end{array} \tag{8.4}$$

$$sm_{j,o'} \rightarrow \left(\sum_{\substack{A_i \in A \setminus \{A_j\} \\ \neg incomp(i,j)}} \; \sum_{\substack{o \in \{1,\ldots,M_i\} \\ b'_{i,k,o} \neq 0}} ite(sm_{i,o} \wedge z^1_{i,j} \wedge z^2_{i,j}; b'_{i,k,o}; 0) \right) \leq B'_k - b'_{j,k,o'}$$
$$\forall A_j \in A, \forall o' \in \{1,\ldots,M_j\}, \forall R_k \in \{R_1,\ldots,R_v\} \tag{8.5}$$

First of all, notice that in (8.2) the variable $z^1_{i,j}$ is defined to be equivalent to an atom. Therefore, the atom can be directly used in replacement of $z^1_{i,j}$. We keep $z^1_{i,j}$ as an alias in the formulation for simplicity and for the sake of resemblance to the original *task* encoding, but it will be replaced by the corresponding atom when passed to the solver. Then, the use of the notion of incompatibility defined by *incomp* let us omit some variables and constraints: constraints (4.13), (4.14), (4.15), (4.17), (4.16) and (4.18) have been replaced by (8.2) and (8.3), and constraints (4.19) have been replaced by (8.5). Finally, there are the new constraints (8.4) which enforce that resource incompatibilities are respected[1].

8.3 CNF Conversion

As we have seen in Section 3, SAT (SMT) expressions are basically propositional (modulo theories) formulas expressed in Conjunctive Normal Form. Nevertheless, most SMT solvers allow specifying any generic Boolean formula, not restricted to be a conjunction of disjunctions. Neither the *time* formulation presented in Section 4.3 nor the *task* formulation refined and presented in Section 8.2 are in CNF. For instance, recall the formulation for constraints (4.11) of the *time* formulation, which is not in CNF:

$$sm_{i,o} \rightarrow (y_{i,t} \leftrightarrow (S_i \leq t \wedge t < S_i + p_{i,o})) \qquad \text{(4.11 revisited)}$$

Any propositional Boolean formula can be converted into an equivalent formula that is in CNF, by applying Boolean algebra transformations. This process may lead to an exponential growth of the size of the formula. For this reason, one normally uses the *Tseytin transformation* [38] or variations to obtain, given a Boolean formula F, an equisatisfiable[2] formula F', which only requires a linear growth, but introduces a linear number of new auxiliary Boolean variables. However, in our case we can reformulate our constraints applying Boolean algebra to provide CNF formulas, and avoiding the solver the internal transformation to CNF that uses new auxiliary Boolean variables. Basically, we decompose double implications to sets of implications, and transform implications to disjunctions. As an example, the expression in (4.11):

$$sm_{i,o} \rightarrow (y_{i,t} \leftrightarrow (S_i \leq t) \wedge (t < S_i + p_{i,o}))$$

is transformed into:

$$\{\neg sm_{i,o} \vee \neg y_{i,t} \vee S_i \leq t\}$$

$$\{\neg sm_{i,o} \vee \neg y_{i,t} \vee t < S_i + p_{i,o}\}$$

$$\{\neg sm_{i,o} \vee y_{i,t} \vee \neg(S_i \leq t) \vee \neg(t < S_i + p_{i,o})\}$$

Although these transformations are trivial, in Section 8.5.2 we show that they have an impact on the size of the encoding and the time performance.

[1] Start time window incompatibilities are enforced by the time window definitions. The disjunctions of renewable resource usages are intrinsic of the instance, and do not need to be constrained. However, the resource incompatibilities must be enforced if we remove the incompatible pairs from the constraints (8.5).

[2] Two Boolean formulas F and F' are equivalent if any model of F is a model of F', and any model of F' is a model of F. The equisatisfiability guarantees that F' is satisfiable if and only if F is satisfiable. Moreover, with the Tseytin transformation we can obtain a model for F from a model for F'.

8.4 Three Versions of the Encodings

In this section we are going to collect the refinements presented in Sections 8.2 and 8.3 to express the new *time* and *task* formulations. We firstly skip the constraints over resource demands (both renewable and non-renewable), and afterwards we propose three different encodings for these constraints, each one adaptable both to *time* and *task* formulations. The first one (Section 8.4.1) is the formulation that we have seen so far, which uses *if-then-else* expressions. In Section 8.4.2 we use 0/1 integer variables to model the sum of demands on a resource as a LIA expression. In Section 8.4.3 we replace the 0/1 integer variables by Boolean variables and express the constraints using Pseudo-Boolean functions.

The following constraints, which are common in both *time* and *task* are preserved from the previous formulation:

$$S_0 = 0 \tag{4.1 revisited}$$

$$S_i \geq ES_i \qquad \forall A_i \in \{A_1, \ldots, A_{n+1}\} \tag{4.2 revisited}$$

$$S_i \leq LS_i \qquad \forall A_i \in \{A_1, \ldots, A_{n+1}\} \tag{4.3 revisited}$$

$$S_j - S_i \geq l_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{4.5 revisited}$$

$$sm_{0,1} = true \tag{4.6 revisited}$$

$$sm_{n+1,1} = true \tag{4.7 revisited}$$

$$\bigvee_{1 \leq o \leq M_i} sm_{i,o} \qquad \forall A_i \in A \tag{4.8 revisited}$$

$$\neg sm_{i,o} \vee \neg sm_{i,o'} \qquad \forall A_i \in A, 1 \leq o \leq M_i, 1 \leq o' \leq M_i, o \neq o' \tag{4.9 revisited}$$

Constraints (4.4) are reformulated as disjunctive clauses:

$$\neg sm_{i,o} \vee S_j - S_i \geq p_{i,o} \qquad \forall (A_i, A_j) \in E, \forall o \in \{1, \ldots, M_i\} \tag{8.6}$$

Constraints for the *time* Formulation

The following constraints substitute (4.11), and give a value to the variables $y_{i,t}$, which is true if an only if A_i is running at time t:

$$\begin{gathered} \neg sm_{i,o} \vee \neg y_{i,t} \vee S_i \leq t \\ \wedge \\ \neg sm_{i,o} \vee \neg y_{i,t} \vee t < S_i + p_{i,o} \\ \wedge \\ \neg sm_{i,o} \vee y_{i,t} \vee \neg (S_i \leq t) \vee \neg (t < S_i + p_{i,o}) \end{gathered} \tag{8.7}$$

$$\forall A_i \in A, \forall o \in \{1, \ldots, M_i\}, \forall t \in \{ES_i, \ldots, LS_i + p_{i,o}\}$$

Constraints for the *task* Formulation

The definition of $z^1_{i,j}$ remains the same, and is true if and only if activity A_i does not start after activity A_j does. Remember that we treat it as an alias (i.e. we don't create the variables nor the following constraints):

$$z^1_{i,j} \leftrightarrow S_i \leq S_j \qquad \forall A_i, A_j \in A, \neg incomp(i,j), i \neq j \qquad \text{(8.2 revisited)}$$

The following constraints substitute (8.3) in the same definition of $z^2_{i,j}$: it denotes whether activity A_j starts before the end of activity A_i.

$$\begin{gathered} \neg sm_{i,o} \vee \neg z^2_{i,j} \vee S_j < S_i + p_{i,o} \\ \wedge \\ \neg sm_{i,o} \vee \neg (S_j < S_i + p_{i,o}) \vee z^2_{i,j} \end{gathered} \qquad (8.8)$$

$$\forall A_i, A_j \in A, \neg incomp(i,j), i \neq j, \forall o \in \{1, \ldots, M_i\}$$

Finally, the following constraints substitute (8.4) and enforce that two activities A_i and A_j that are incompatible due to resource demands (i.e. $RI_{i,j} = true$) never overlap:

$$\begin{aligned} &\neg sm_{i,o} \vee \neg sm_{j,o'} \vee S_j \geq S_i + p_{i,o} \vee S_i \geq S_j + p_{j,o'} \qquad && \forall A_i, A_j \in A, \\ & && RI_{i,j} = true, \\ & && STI_{i,j} = false, \\ & && \forall o \in \{1, \ldots, M_i\}, \\ & && \forall o' \in \{1, \ldots, M_j\} \end{aligned} \qquad (8.9)$$

8.4.1 *Ite*: Use of if-then-else Expressions

Here we consider the definition of the constraints over resources (both renewable and non-renewable) that we have seen so far. From now on, we will denote it as the *ite* encoding, and we will have both the *time ite* and the *task ite* encodings.

The constraint over non-renewable resources is the same for both *time* and *task*:

$$\left(\sum_{A_i \in A} \sum_{\substack{o \in \{1, \ldots, M_i\} \\ b'_{i,k,o} \neq 0}} ite\left(sm_{i,o}; b'_{i,k,o}; 0\right) \right) \leq B'_k \qquad \forall R_k \in \{R_{v+1}, \ldots, R_q\} \qquad (8.10)$$

As before, the differences arise in the encoding of the constraints over renewable resources.

***Time ite* Encoding**

The following constraints enforce that the capacities of renewable resources are not exceeded at any time:

$$\left(\sum_{A_i \in A} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ ES_i \leq t \leq LS_i + p_{i,o} - 1 \\ b'_{i,k,o} \neq 0}} ite\left(sm_{i,o} \wedge y_{i,t};\, b'_{i,k,o};\, 0\right) \right) \leq B'_k$$

$$\forall R_k \in \{R_1, \ldots, R_v\}, \forall t \in \{0, \ldots, UB\} \quad (8.11)$$

***Task ite* Encoding**

The equivalent for *task* ensures that the capacities of renewable resources are not exceeded at the start time of any activity:

$$\neg sm_{j,o'} \vee \left(\sum_{\substack{A_i \in A \setminus \{A_j\} \\ \neg incomp(i,j)}} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ b'_{i,k,o} \neq 0}} ite(sm_{i,o} \wedge z^1_{i,j} \wedge z^2_{i,j};\, b'_{i,k,o};\, 0) \right) \leq B'_k - b'_{j,k,o'}$$

$$\forall A_j \in A, \forall o' \in \{1, \ldots, M_j\}, \forall R_k \in \{R_1, \ldots, R_v\} \quad (8.12)$$

8.4.2 *Mult*: Use of 0/1 Integer Variables

This formulation expresses the constraints over resources using Linear Integer Arithmetic expressions. The resulting encodings are in CNF, so they avoid the internal transformations of the SMT solvers (cf. *ite*). From now on we will denote it as the *mult* encoding. We introduce a new 0/1 integer variable $m_{i,o}$ that takes value 1 if A_i runs in mode o, and 0 otherwise. The following constraints define its value:

$$(m_{i,o} \geq 0) \ \wedge (m_{i,o} \leq 1) \ \wedge \ (\neg(m_{i,o} \geq 1) \vee sm_{i,o}) \ \wedge \ (m_{i,o} \geq 1 \vee \neg sm_{i,o}) \quad (8.13)$$

$$\forall A_i \in A, \forall o \in \{1, \ldots, M_i\}$$

The constraints over the non-renewable resources are defined as:

$$\left(\sum_{A_i \in A} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ b'_{i,k,o} \neq 0}} m_{i,o} \cdot b'_{i,k,o} \right) \leq B'_k \qquad \forall R_k \in \{R_{v+1}, \ldots, R_q\} \quad (8.14)$$

***Time mult* Formulation**

We introduce a new 0/1 integer variable $x_{i,o,t}$ that takes value 1 if A_i runs in mode o at time t, and 0 otherwise:

$$
\begin{gathered}
(x_{i,o,t} \geq 0) \wedge (x_{i,o,t} \leq 1) \\
\wedge \\
(\neg(x_{i,o,t} \geq 1) \vee sm_{i,o}) \wedge (\neg(x_{i,o,t} \geq 1) \vee y_{i,t}) \wedge (x_{i,o,t} \geq 1 \vee \neg sm_{i,o} \vee \neg y_{i,t})
\end{gathered} \tag{8.15}
$$

$$\forall A_i \in A, \forall o \in \{1, \ldots, M_i\}$$

The constraint over the renewable resources is:

$$
\left(\sum_{A_i \in A} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ ES_i \leq t \leq LS_i + p_{i,o} - 1 \\ b'_{i,k,o} \neq 0}} x_{i,o,t} \cdot b'_{i,k,o}) \right) \leq B'_k
$$

$$\forall R_k \in \{R_1, \ldots, R_v\}, \forall t \in \{0, \ldots, UB\} \tag{8.16}$$

***Task mult* Formulation**

We introduce a new 0/1 integer variable $z_{i,o,j}$ that takes value 1 if A_i is running in mode o when A_j starts, and 0 otherwise:

$$
\begin{gathered}
z_{i,o,j} \geq 0 \wedge z_{i,o,j} \leq 1 \\
\wedge \\
\neg(z_{i,o,j} \geq 1) \vee sm_{i,o} \\
\wedge \\
\neg(z_{i,o,j} \geq 1) \vee z^1_{i,j} \\
\wedge \\
\neg(z_{i,o,j} \geq 1) \vee z^2_{i,j} \\
\wedge \\
z_{i,o,j} \geq 1 \vee \neg sm_{i,o} \vee \neg z^1_{i,j} \vee \neg z^2_{i,j}
\end{gathered} \tag{8.17}
$$

$$\forall A_i, A_j \in A, i \neq j, \neg incomp(i,j), \forall o \in \{1, \ldots, M_i\}$$

The constraint over the renewable resources is:

$$
\neg sm_{j,o'} \vee \left(\sum_{\substack{A_i \in A \setminus \{A_j\} \\ \neg incomp(i,j)}} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ b'_{i,k,o} \neq 0}} z_{i,o,j} \cdot b'_{i,k,o}) \right) \leq B'_k - b'_{j,k,o'} \tag{8.18}
$$

$$\forall A_j \in A, \forall o' \in \{1, \ldots, M_j\}, \forall R_k \in \{R_1, \ldots, R_v\}$$

8.4.3 *BDD*: Pseudo-Boolean Constraints

This formulation expresses the constraints over resources using Pseudo-Boolean constraints. As we have seen in Section 4.5, a Pseudo-Boolean constraint has the form:

$$a_1 x_1 + \cdots + a_n x_n \# K$$

where the a_i and K are integer coefficients, the x_i are Boolean variables, and the relation operator $\#$ belongs to $\{<, >, \leq, \geq, =\}$. Our definitions of the constraints over resources perfectly fit this template, and therefore can be modeled with Pseudo-Boolean constraints. We can define the constraints over non-renewable resources as:

$$\left(\sum_{A_i \in A} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ b'_{i,k,o} \neq 0}} sm_{i,o} \cdot b'_{i,k,o} \right) \leq B'_k \qquad \forall R_k \in \{R_{v+1}, \ldots, R_q\} \tag{8.19}$$

Notice that there is a difference with respect to the *mult* encoding (8.14), that is that we do not introduce new integer 0/1 variables $m_{i,o}$ but we directly use the Boolean variables $sm_{i,o}$. Hence, constraints (8.14) contain LIA expressions that are handled by the LIA theory solver, whereas this is not the case for constraints (8.19). The latter is in fact a formal definition of the constraints, but we need to use some implementation of Pseudo-Boolean constraints that translates them into Boolean formulas. For this purpose, we are going to use Binary Decision Diagrams (BDDs), concretely the framework developed in [7] that we mentioned in Section 4.5.

Encoding the constraints over resources with Pseudo-Boolean constraints has an impact in the whole encoding nature, because all the remaining constraints containing theory (LIA) expressions are (with trivial arithmetic transformations) Difference Logic expressions. As we have seen in Section 3.2, some SMT solvers (Yices included) use specialized theory solvers for Difference Logic.

Time BDD Formulation

To encode the constraints over renewable resources, we are going to define a Boolean variable that has the same meaning that the integer 0/1 variable $x_{i,o,t}$ from the *time mult* encoding. It is $x'_{i,o,t}$:

$$\begin{gathered} \neg x'_{i,o,t} \vee sm_{i,o} \\ \wedge \\ \neg x'_{i,o,t} \vee y_{i,t} \\ \wedge \\ x'_{i,o,t} \vee \neg sm_{i,o} \vee \neg y_{i,t} \end{gathered} \tag{8.20}$$

$$\forall t \in \{ES_i, \ldots, LS_i + p_{i,o} - 1\}$$

The constraint over the renewable resources is:

$$\left(\sum_{A_i \in A} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ ES_i \leq t \leq LS_i + p_{i,o} - 1 \\ b'_{i,k,o} \neq 0}} x'_{i,o,t} \cdot b'_{i,k,o}) \right) \leq B'_k$$

$$\forall R_k \in \{R_1, \ldots, R_v\}, \forall t \in \{0, \ldots, UB\} \quad (8.21)$$

***Task* Formulation**

Similarly to the case of the *time BDD* encoding, we are defining a Boolean variable $z'_{i,o,j}$ in substitution of the integer 0/1 variable $z_{i,o,j}$:

$$\begin{gathered} \neg z'_{i,o,j} \vee sm_{i,o} \\ \wedge \\ \neg z'_{i,o,j} \vee z^1_{i,j} \\ \wedge \\ \neg z'_{i,o,j} \vee z^2_{i,j} \\ \wedge \\ z'_{i,o,j} \vee \neg sm_{i,o} \vee \neg z^1_{i,j} \vee \neg z^2_{i,j} \end{gathered} \quad (8.22)$$

$$\forall A_i, A_j \in A, i \neq j, \neg incomp(i,j), \forall o \in \{1, \ldots, M_i\}$$

The constraint over the renewable resources is:

$$\neg sm_{j,o'} \vee \left(\sum_{\substack{A_i \in A \setminus \{A_j\} \\ \neg incomp(i,j)}} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ b'_{i,k,o} \neq 0}} z'_{i,o,j} \cdot b'_{i,k,o}) \right) \leq B'_k - b'_{j,k,o'} \quad (8.23)$$

$$\forall A_j \in A, \forall o' \in \{1, \ldots, M_j\}, \forall R_k \in \{R_1, \ldots, R_v\}$$

8.5 Results

In this section we are going to evaluate the impact on the encoding size and the time performance of the encoding refinements presented in Sections 8.2, 8.3 and 8.4. We evaluate each improvement progressively: first the impact of the preprocessings, then the improvement achieved by expressing the constraints of the *ite* encoding in CNF, and finally the performance comparison of the *ite*, *mult* and *BDD* encodings.

8.5.1 Application of the New Preprocessings

We present in this section the impact of the preprocessings in reducing the size of the encodings, and the improvement that they suppose on the time performance. Figures 8.4 and 8.5 show, for every instance in j30SAT, the percentage of reduction of the number of Boolean variables in the *time* and *task* encodings respectively. Each point corresponds to an instance. A point with a percentage value of 5% (axis y), means that there is an instance whose encoding using the new preprocessings has decreased a 5% in number of Boolean variables with respect to the encoding for the same instance without using the new preprocessings. In axis x, we put again the solving time for the *time* encoding of Section 4.3.1, to make this results complementary to the results of Section 7.2. The same results on the number of clauses and the number of atoms have shown to be proportional to the reduction of Boolean variables, so we do not include the plots to avoid redundancy. On the other hand, Figure 8.6 compares the performance of the *time* encoding without using the new preprocessings and using them, and Figure 8.7 shows the same result for *task*[1].

We can see that the *time* encoding only reduces its size in some of the hardest instances, which are the ones that let us discover some energy precedences (recall Figure 7.1), and less than a 7% in all cases. We cannot conclude that this preprocessing clearly benefits the time encoding, although the hardest instances experienced an improvement. On the other hand, the task encoding has the most important reduction of the size of the encoding in the easiest instances, thanks to the start time window incompatibilities and the resource disjunctions (recall Figures 7.2 and 7.4). However, the results on the easiest instances do not suggest that the SMT solver is clearly benefited by this encoding reduction. This is different for the hardest instances, which are benefited from the resource incompatibilities (recall Figure 7.3) and show generally a time speedup.

8.5.2 CNF Conversion

Figures 8.8 and 8.9 compare the solving times with and without translating the formulas into CNF for *time* and *task* respectively. It can be seen that for *time* the solving time is clearly reduced in the easiest instances (up to 100 seconds of solving time), and this is also the case for the majority of the hardest instances. In contrast, the translation to CNF has a very different effect with *task*, since there is no clear victor between the two alternatives and the plot presents a lot of variance. We can see in Table 8.1 that making the translation into CNF reduces the number of Boolean variables and clauses in an important proportion, thanks to avoiding the internal translation of the SMT solver.

[1]The scatter plots like the ones in Figure 8.6 or Figure 8.7 show, for each instance of a set, which is the best of the two approaches that are being compared. That is, every point corresponds to a different instance, and each axis means the execution time of a different approach. If a point is placed in the upper triangle of the plot, the approach in the y axis is the one that requires most time for that instance, and the opposite happens when the point is placed in the lower triangle. Recall that we use a timeout of 3600 seconds. An instance that only timed out in the approach of axis x will be placed in the right vertical edge of the plot. If the timed out approach is the one in axis y, the point will be placed in the top horizontal edge. Finally, the instances timed out in both approaches appear in the top right corner of the plot. We will frequently use this kind of plots to compare results.

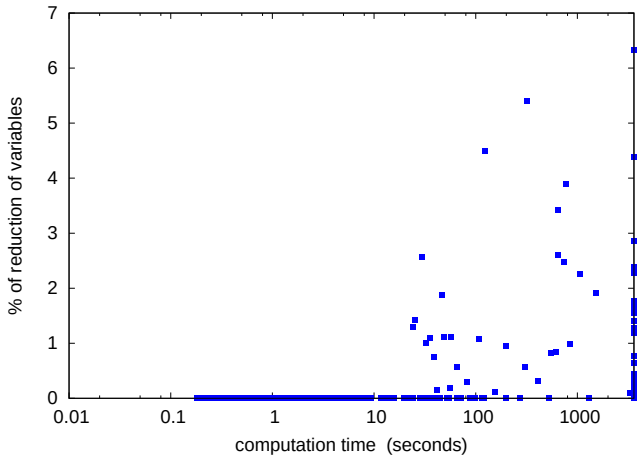

Figure 8.4: Percentage of reduction of the number of Boolean variables in the *time* encoding due to the new preprocessings. The points in the right vertical edge of the plot correspond to instances that have been not solved within the timeout of 3600 seconds.

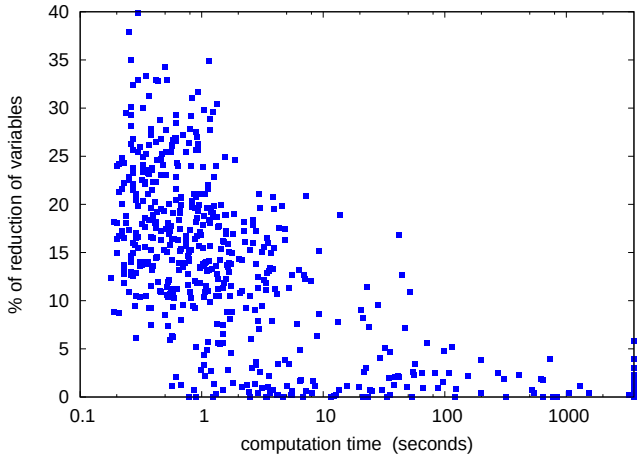

Figure 8.5: Percentage of reduction of the number of Boolean variables in the *task* encoding due to the new preprocessings. The points in the right vertical edge of the plot correspond to instances that have been not solved within the timeout of 3600 seconds.

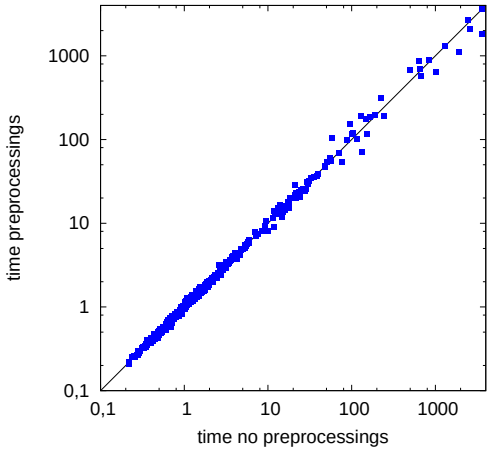

Figure 8.6: Solving times of the j30SAT set with *time* using the new preprocessings compared to the solving time without using the preprocessings.

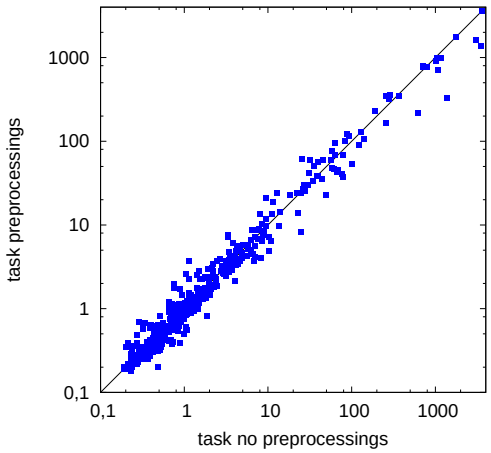

Figure 8.7: Solving times of the j30SAT set with *task* using the new preprocessings compared to the solving time without using the preprocessings.

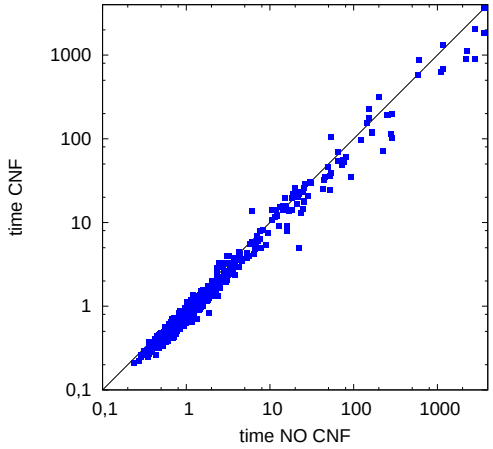

Figure 8.8: Comparison of the solving times of *time* transformed into CNF (axis y) and without transforming into CNF (axis x).

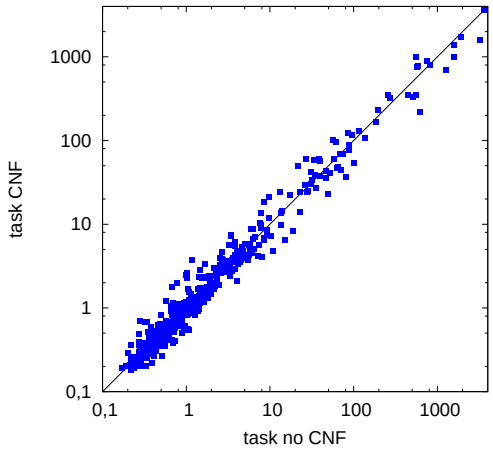

Figure 8.9: Comparison of the solving times of *task* transformed into CNF (axis y) and without transforming into CNF (axis x).

	time			task		
	no CNF	**CNF**	**% of reduction**	**no CNF**	**CNF**	**% of reduction**
clauses	59741	45665	23,56	23487	18949	19,32
Boolean vars	46417	38153	17,8	13115	11841	9,72

Table 8.1: Average number of clauses and Boolean variables of the instances in j30SAT for *time* and *task*, and percentage of the reduction, with and without making the transformation into CNF. The sizes are counted with the first UB of the linear search.

	time			task		
	ite	**mult**	**BDD**	**ite**	**mult**	**BDD**
solved	535	537	544	526	525	531
clauses	45665	14523	178441	18949	3857	140514
atoms	31180	7102	1648	9938	3298	1710
Boolean vars	38153	8986	140376	11841	9374	77184

Table 8.2: Number of instances solved, and average number of clauses, atoms and Boolean variables of the instances in j30SAT. The total number of instances in j30SAT is 552. The sizes are counted with the first UB of the linear search.

8.5.3 *Ite*, *Mult* and *BDD*

In this section we compare the performance of the three versions of the encodings: *ite*, *mult* and *BDD*. Figures 8.10 and 8.11 compare the performance of *time ite* with *time mult* and *time BDD* respectively. Figures 8.12 and 8.13 compare the performance of *task ite* with *task mult* and *task BDD* respectively. Finally, Table 8.2 contains, for each encoding, the number of instances solved without timing out, and the size in terms of number of Boolean variables, atoms and clauses. The main results are the following:

- The *time mult* encoding is slower than the *time ite* encoding in the easiest instances. For the hardest instances there is no a clear victor although *mult* solves two instances more than *ite* .
- There is no a clear victor between *task ite* and *task mult* in the easiest instances, but the tendency is that *mult* becomes slightly faster as the hardness of the instance increases.
- The *BDD* encodings are in clear disadvantage in the easiest instances. This is due to the time required to compute the ROBDD for the Boolean implementation of the Pseudo-Boolean constraints. However, there is a very important speedup in the hardest instances, solving more instances than the other approaches, with 531 instances solved with *task* and 544 with *time*.
- The *mult* encoding is the most compact regarding the number of clauses and Boolean variables, with a significant improvement with respect to *ite* also in number of atoms. *BDD* is the one that has the least number of atoms, and all of them corresponding to precedences or constraints on the domain of the variables S_i, and hence belonging to the theory of Integer Difference Logic. On the other hand, the numbers of Boolean variables and clauses increase a lot, due to the Boolean implementation of the constraints over resources.

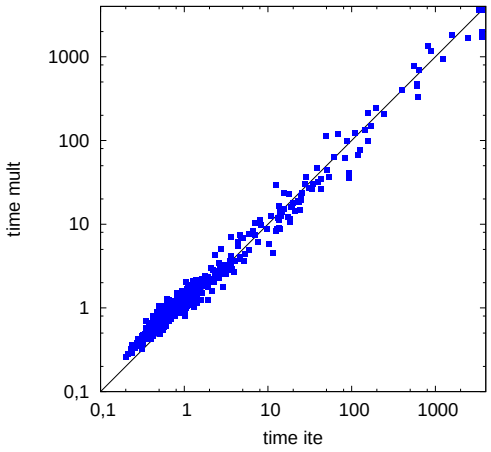

Figure 8.10: *time ite* compared to *time mult*.

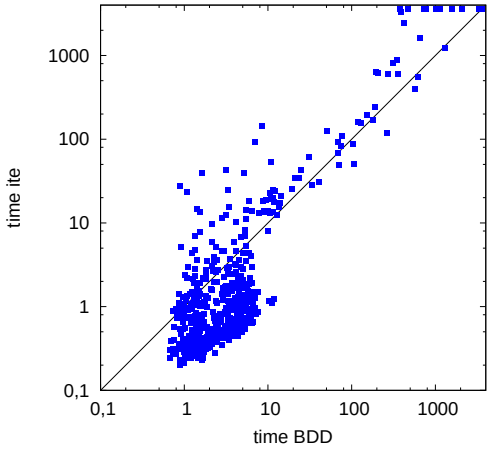

Figure 8.11: *time BDD* compared to *time ite*.

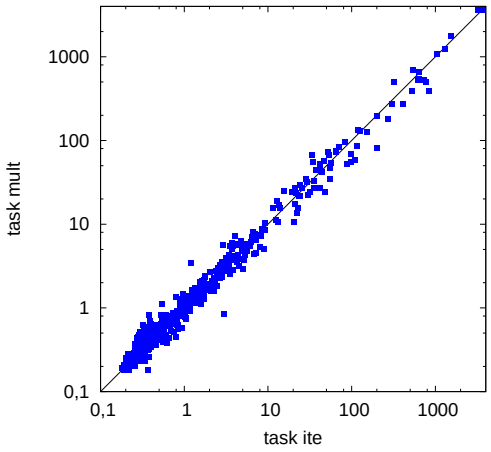

Figure 8.12: *task ite* compared to *task mult*.

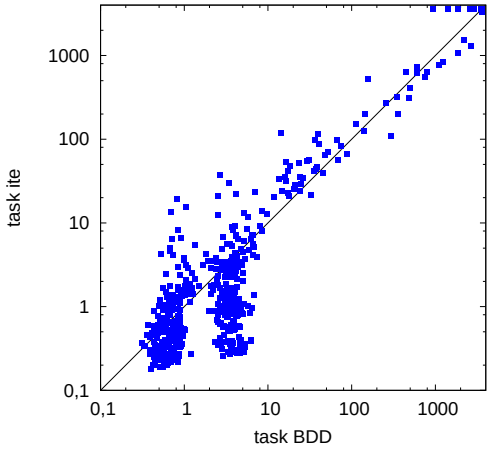

Figure 8.13: *task BDD* compared to *task ite*.

Chapter 9

Optimization Procedure

In this chapter we are going to introduce and study a modular optimization framework for the MRCPSP. Firstly, in Section 9.1 we explain how to efficiently check the feasibility of an instance. Section 9.2 shows how to find a significantly better upper bound for the makespan than the trivial upper bound. Finally, Section 9.3 presents a generic algorithm for solving the MRCPSP and different approaches to simplify the encodings as the optimization procedure evolves.

9.1 Detecting Infeasibility

In Section 2.2 we saw that any MRCPSP instance has a feasible schedule if and only if the following conditions hold:

- There not exist a cycle in the precedence graph (i.e., an activity is forced to start after itself finishes).
- There not exist any activity whose demand over a resource in all execution modes is greater than its capacity.
- There exist a schedule of modes such that all the non-renewable resource constraints (2.6) are satisfied.

For this reason, it is enough to check that the three of them hold to ensure that the instance is satisfiable. The first two are easily verifiable in polynomial time. The benchmark sets normally do not contain instances with such properties because they are not natural of real problem instances and do not enrich the set. However, the third condition holds in many of the instances of PSPLib, and it is by itself hard to check. Notice that this condition is independent of the execution times of the activities. Therefore, assuming that there are not infeasible modes, we could translate the problem of checking the feasibility of an instance of the MRCPSP as the problem of checking whether there is a schedule of modes such that the capacity of the non-renewable resources is never exceeded. Formally, it could be defined as the MRCPSP-SAT, consisting in a tuple (V, M, R, B, b) where:

- $A = \{A_1, \ldots, A_n\}$ is a set of activities.
- $M \in \mathbb{N}^n$ is a vector of naturals, being M_i the number of modes that activity i can execute, with $M_i \geq 1, \forall A_i \in A$.

- $R = \{R_1, \ldots, R_q\}$ is a set of q non-renewable resources.
- $B \in \mathbb{N}^q$ is a vector of naturals, being B_k the available amount of each resource R_k.
- b is a matrix of naturals corresponding to the resource demands of activities per mode. $b_{i,k,o}$ represents the amount of resource R_k used during the execution of activity A_i in mode o.

The problem consists in finding a schedule of modes $\mathbb{SM} = (\mathbb{SM}_1, \ldots, \mathbb{SM}_n)$ where $\mathbb{SM}_i$, $1 \leq \mathbb{SM}_i \leq M_i$, denotes the mode of each activity A_i, and the following constraint has to be satisfied:

$$\left(\sum_{A_i \in A} \sum_{o \in \{1,\ldots,M_i\}} ite(\mathbb{SM}_i = o;\, b_{i,k,o};\, 0) \right) \leq B_k \qquad \forall R_k \in \{R_1, \ldots, R_q\}$$

This new problem is equisatisfiable to the MRCPSP, but is totally independent of the start times of the activities. Hence, it is presumably easier to check the feasibility of the MRCPSP-SAT than of the MRCPSP.

The plot in Figure 9.1 indicates that it is indeed the case. It shows the comparison of execution time for the unsatisfiable instances of the j30 set, when encoding the MRCPSP-SAT and when encoding the MRCPSP in the *time* formulation. Figure 9.2 contains the same comparison for the *task* formulation. The two encodings of the MRCPSP have been done using the trivial *UB* of the instances. It can be seen that the conversion to the MRCPSP-SAT makes it much faster to detect the infeasibility of the instances. Between the two encodings for the MRCPSP, the *time* is the one that requires more time with big difference. Figure 9.3 shows these pronounced differences between the three approaches in a box plot[1]. There is one order of magnitude of improvement from the *time* encoding to the *task* encoding, and one order of magnitude from the *task* encoding to the MRCPSP-SAT. Notice also that the variance of the execution varies in the same fashion among the three approaches.

9.2 Adjusting the Upper Bound for the Optimum Makespan

In Section 4.1.3 we have seen that we can find a trivial upper bound *UB* for the makespan equal to the sum of the maximum duration of each activity. This is however an *UB* very distant from the optimum makespan in most of the instances. The makespan of any feasible schedule is an upper bound for the optimum makespan, so we could try to improve the trivial *UB* by finding a feasible schedule with a heuristic algorithm. The PSS algorithm presented in Section 4.2 would suit for this purpose. This heuristic algorithm works for the RCPSP problem, i.e., it does not deal with variable durations and resource demands depending on the execution mode. However, the quick feasibility check for the MRCPSP-SAT that we have seen in Section 9.1 returns a feasible schedule of modes whenever the instance is feasible. If that is the case, the MRCPSP instance can be adapted by fixing the durations and the renewable resource demands of each activity according to the schedule of modes. In other words, given a feasible schedule of modes, we can reduce the problem to an RCPSP, and then the PSS algorithm applies. We can ignore

[1] Contrarily to the scatter plots that we have seen so far, the box plots like the one in Figure 9.3 do not provide the information of which is the best approach for each instance, but which is the best approach for a whole set of instances, by showing the quartiles of the running times of the set with every approach. The individual points appearing in the boxplot are outliers.

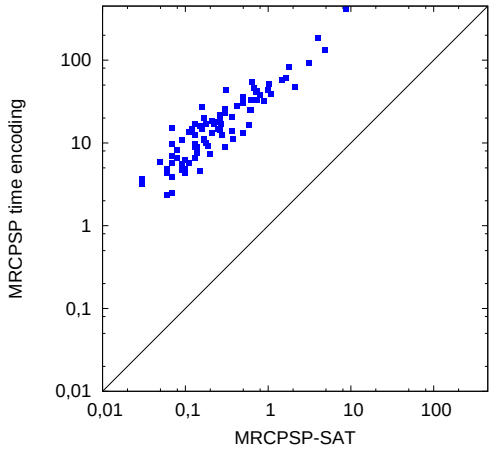

Figure 9.1: Time comparison of proof of infeasibility between the MRCPSP-SAT, and the MRCPSP with *time ite* encoding. The time is expressed in seconds, and the plot has logarithmic scale in both axes.

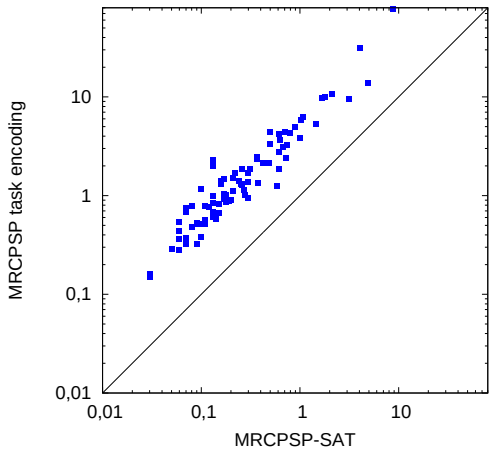

Figure 9.2: Time comparison of proof of infeasibility between the MRCPSP-SAT, and the MRCPSP with *task ite* encoding. The time is expressed in seconds, and the plot has logarithmic scale in both axes.

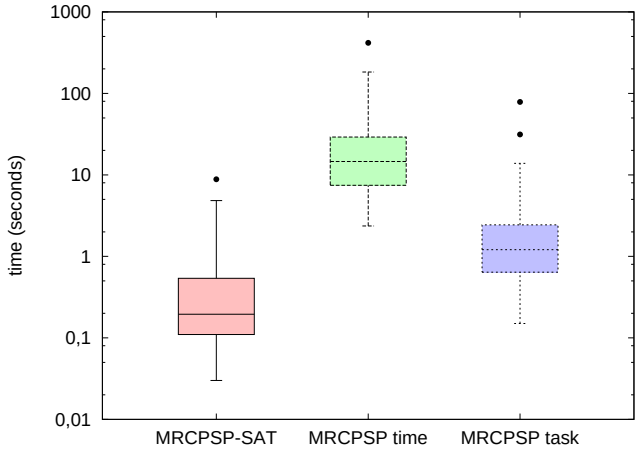

Figure 9.3: Time comparison of proof of infeasibility for the MRCPSP-SAT, and the MRCPSP with *time ite* and *task ite* encodings. Axis y contains the execution time expressed in seconds, and the plot has logarithmic scale in the y axis.

the constraints over the non-renewable resource usages, since we already know that the schedule of modes satisfies them. The advantage of using this heuristic is that it requires little time to find a solution with a makespan significantly better than the trivial UB. As seen in Section 8, it is important for the size of the encoding and for the performance to find a good starting upper bound. This improvement can be seen in Figure 9.4, which compares the values of the UB given by the heuristic and the values given by the trivial UB, for all the instances in the j30SAT set. In the first case, the mean is 76, while in the second case it is 239.

9.3 Optimizing the Makespan

The two previous sections explained how to check if an instance has feasible solutions, and if that is the case, how to obtain a first UB for the makespan. Together with a final phase of finding the optimum makespan, they describe the generic algorithm (see Algorithm 2) used to solve the instances. This algorithm contains three main steps that can be implemented independently: verify feasibility, bound the makespan, and optimize. We have already covered the two first steps, and in this section we are going to study algorithms for the last optimization step.

As we have seen in Section 8, the size of the *time* encoding is highly dependent on the time horizon, and a good refinement of it can also help to tighten the time windows. In the following sections we study how reducing the size of the encoding and guiding the solver through the optimization process can help to design efficient optimization methods.

Will gradually introduce different optimization approaches (Sections 9.3.1, 9.3.2, 9.3.3) while analysing their behaviour for the *time* encoding, which has shown to be the highly dependent on the encoding size. All these approaches are based on the linear search schema commented in Section 4.4. These methods take advantage of the fact that the time windows are compressed as the UB decreases. Although *task* detects more start time window incompatibilities as the time windows are compressed, and this suppose a reduction of the size of the encoding (as seen in Section 8.5.1), we have already seen that the hardest instances do not get benefit of this preprocessing, and some preliminary experimentation has shown that the optimization techniques

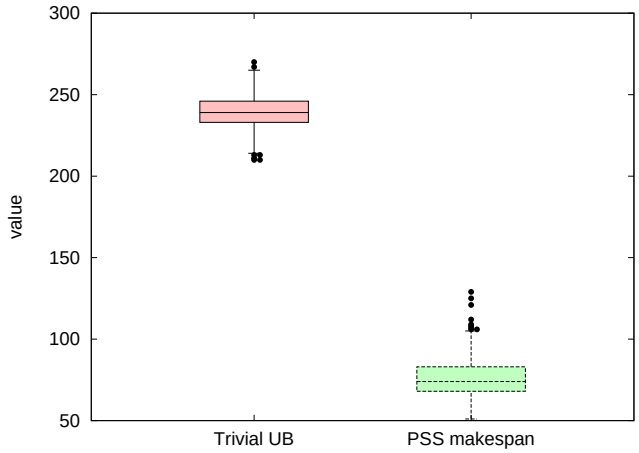

Figure 9.4: Values of the trivial upper bound and of the upper bound given by the PSS heuristic, on the j30SAT set.

Algorithm 2 Generic solving algorithm for the MRCPSP

Output: Optimum makespan if feasible. Otherwise return infeasible.

$INS \leftarrow read\_MRCPSP\_instance()$ //Contains: V, A, M, p, E, R, B, b
$PREP \leftarrow preprocessing()$ //Contains: $E^*, ES, LS, STI, RI, D, B', b'$
$ENC \leftarrow encode\_MRCPSP\_SAT(INS, PREP)$
$(SAT, MODEL) \leftarrow smt\_check(ENC)$ //Check feasibility, and give a model if any
if SAT **then**
  $LB \leftarrow l_{0,n+1}$ //Trivial lower bound
  $SM \leftarrow get\_schedule\_of\_modes(MODEL)$
  $UB \leftarrow pss(INS, SM)$
  $OPTIMUM \leftarrow find\_optimum(LB, UB, INS, PREP)$
  return $OPTIMUM$
else
  return $INFEASIBLE$
end if

presented below do not suppose an improvement for *task*. For this reason, in this section we will focus on the *time* encoding.

9.3.1 Encoding Size Reduction

In Section 3 we presented how some current SMT solvers are able to deal with optimization problems. They use optimization procedures on the theory solvers to find models with improved objective value, and restrict the search according to these new bounds. On the other hand, there are SMT solvers that do not deal with optimization but only with satisfiability checks, as is also the case for classical SAT solvers. The optimization applications using this kind of solvers usually implement custom optimization procedures based on successive satisfiability checks by bounding the objective function. Notice that the two former approaches simply bound the objective function as the search of the optimum proceeds, but the SMT solver ignores which problem it is dealing with. Therefore it is unable to use problem specific knowledge to infer partial models that are logical consequence of the formula given an upper bound of the objective function. However, this information can be obtained as a preprocessing previous to check the satisfiability for some bound.

Let us illustrate this idea with an example for the MRCPSP. Suppose a project in which we have found an upper bound for the makespan equal to 20. In other words, we know that there is a solution for this project in which the finishing activity starts at time 20[1]. Recall the constraint of not exceeding the capacity of renewable resources at any time, for the *time* formulation:

$$\left(\sum_{A_i \in A} \sum_{\substack{o \in \{1,\dots,M_i\} \\ ES_i \leq t \leq LS_i + p_{i,o} - 1}} ite\left(sm_{i,o} \wedge y_{i,t};\, b'_{i,k,o};\, 0\right) \right) \leq B'_k$$

$$\forall R_k \in \{R_1, \dots, R_v\}, \forall t \in \{0 \dots UB\}$$

Let us suppose that we have found by means of an SMT check a schedule with makespan 20. Then, we could check for a better makespan by bounding it to 19 and checking again. When we force UB to be less than or equal to 19, we gain new knowledge. On the one hand, the former constraint is trivially satisfied for time $t = 20$, because no activity will be running at time 20. This means that all the clauses and variables involved in this constraint can be removed. Moreover, we can refine the encoding for each activity due to time window compression. Let A_i be an activity with extended precedence to the finishing activity equal to 10, i.e. $(i, n+1, 10) \in E^*$, and $pmax_i = \max_{1 \leq o \leq M_i}(p_{i,o}) = 5$. Then, we know that:

$$LS_i = UB - l_{i,n+1} = UB - 10$$

$$LC_i = LS_i + pmax_i = UB - 5$$

For the starting $UB = 20$, it was possible for activity A_i to be running at time 14, because $LC_i = 15$. However, if we consider $UB = 19$, we have that $LC_i = 14$, or in other words, we know for sure that activity A_i will never be running at time 14. In this case, the variable $y_{i,14}$ can

[1]Recall that the finishing activity is dummy, it has duration 0 and is defined to be executed once all the other activities have finished. Therefore the makespan coincides both with the starting time and the ending time of this activity.

be assigned to false or removed. Moreover, S_i is an integer variable whose domain is defined as $\{ES_i, \ldots, LS_i\}$, and therefore a reduction of LS_i also reduces the domain of S_i.

We have seen how knowing an upper bound let us gain a-priori knowledge of the values of some variables, and constraints. Now we analyse how this information could be used to help the SMT solver in finding a solution. We will do it by using the API of Yices, which does not offer optimization functionalities, to interact with the SMT solver after every satisfiability check for a given bound and analyse some metrics.

Let us define some optimization strategies based on multiple checks to the SMT solver, reducing the upper bound until the optimum if found. Each one makes a different use of the knowledge obtained with the UB:

Bound makespan: It is the basic linear search schema, presented in Algorithm 3. With this strategy the encoding is computed and passed to the SMT solver only once, with the first known UB. Every time a new UB is found, we assert a new constraint of the form $S_{n+1} \leq UB$. With this strategy, no knowledge is given to the SMT solver.

Compress time windows: It corresponds to Algorithm 4. It behaves as *bound makespan*, and the time windows are compressed in each query according to the UB. It is done by asserting new clauses that are added to the original encoding. Concretely, we assert single atom clauses of the form $(S_i \leq LS_i)$ to truncate the domain of S_i.

Assign variables: It corresponds to Algorithm 5. It behaves as *compress time windows*, and we assert single variable clauses of the form $(\neg y_{i,t})$ in each iteration. These variables express the (im)possibility for an activity A_i to be running at a time t. By doing this, we are adjusting the range of time an activity can be running with the current UB.

Minimize encoding: It corresponds to Algorithm 6. This approach recomputes the encoding every time a new UB is discovered. The encoding is adjusted to only define variables and clauses that make sense for the given UB. This means treating each call to the SMT solver independently, each time beginning from scratch. In terms of knowledge provided to the solver, it is equivalent to the *assign variables* strategy.

The second and third strategies encode the problem only once, and achieve problem simplification by assigning Boolean variables and reducing domains of integer variables. They do it by extending the encoding, asserting new clauses in each iteration. Since they are single atom clauses, the atom must be satisfied to satisfy the clause. This means that the values for the corresponding Boolean variables can be propagated at base level without any decision. These strategies do not reduce the size of the encoding (i.e. the number of clauses and variables), but they do not need to encode the problem each time. For this reason, the internal status of the solver persists after each iteration. This lets us take advantage of the learning capabilities of the CDCL(T) SMT solvers by reusing variable activities and learnt lemmas from previous checks. These strategies require that the SMT solver support multiple satisfiability checks and extensions of the encoding between checks. This is the case of all the SMT solvers that support the SMT-LIB 2 standard, so these algorithms could be applied to most available SMT solvers.

On the other hand, the *minimize encoding* achieves problem simplification by omitting constraints and its associated clauses and variables. This strategy could fit to any SMT solver, because it does not require an interactive API but it initializes the solver from scratch at every iteration. By doing this, we can reduce the number of variables and clauses of the encoding according to the current UB and pass a smaller problem to the SMT solver. It has the drawback of beginning the satisfiability check from scratch every time, and hence losing the activity of variables and the database of learnt lemmas of the previous iteration. Moreover, if it is very

easy to find a schedule for the given UB, the time invested to provide the encoding to the solver could significantly penalize the overall execution time of the iteration.

Algorithm 3 Bound makespan algorithm

Require: The instance is feasible.
Input: LB, UB, instance data (INS), preprocessing data ($PREP$).
Output: Optimum makespan

```
ENC ← encode_MRCPSP(LB, UB, INS, PREP)
smt_assert_encoding(ENC)
SAT ← smt_check()
if SAT then
  MODEL ← smt_get_model()
  MAKESPAN ← smt_get_makespan(MODEL)
  UB ← MAKESPAN − 1
end if
while SAT and UB ≥ LB do
  smt_assert(S_{n+1} ≤ UB)
  SAT ← smt_check()
  if SAT then
    MODEL ← smt_get_model()
    MAKESPAN ← smt_get_makespan(MODEL)
    UB ← MAKESPAN − 1
  end if
end while
if SAT then
  return UB
else
  return UB + 1
end if
```

In order to evaluate the impact of the reduction of the problem size, we have run all the algorithms for all the instances in the j30SAT set. The time performances are compared by pairs of algorithms with scatter plots, which illustrate which of the two algorithms is better for each problem instance. First, we compare in Figure 9.5 the *bound makespan* algorithm, which is completely blind, and the *compress time windows* algorithm, which performs domain reduction. We can see that there is no clear dominance between one strategy and the other. This means that we do not provide relevant information to the SMT solver by explicitly reducing the time windows.

Let us analyse why it happens. What we are asserting in the *compress time windows* algorithm is:

$$S_i \leq LS_i = UB - l_{i,n+1} \qquad \forall A_i \in A \tag{9.1}$$

$$S_{n+1} \leq UB \tag{9.2}$$

whereas in the *bound makespan* algorithm we only assert (9.2). Recall the encoding of extended precedences of variables:

$$S_j - S_i \geq l_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^*$$

Algorithm 4 Compress time windows algorithm

Require: The instance is feasible.
Input: LB, UB, instance data (INS), preprocessing data ($PREP$).
Output: Optimum makespan

```
ENC ← encode_MRCPSP(LB, UB, INS, PREP)
smt_assert_encoding(ENC)
SAT ← smt_check()
if SAT then
  MODEL ← smt_get_model()
  MAKESPAN ← smt_get_makespan(MODEL)
  UB ← MAKESPAN − 1
end if
while SAT and UB ≥ LB do
  PREP ← compress_time_windows(UB, PREP)
  smt_assert(S_{n+1} ≤ UB)
  smt_assert_domain_reduction(INS, PREP) //Clauses (S_i ≤ LS_i)
  SAT ← smt_check()
  if SAT then
    MODEL ← smt_get_model()
    MAKESPAN ← smt_get_makespan(MODEL)
    UB ← MAKESPAN − 1
  end if
end while
if SAT then
  return UB
else
  return UB + 1
end if
```

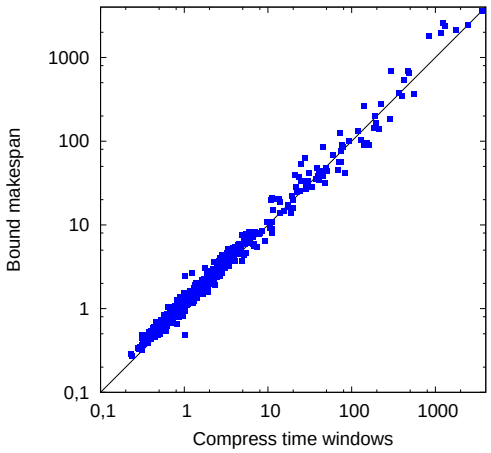

Figure 9.5: Comparison of time of *bound makespan* and *compress time windows* algorithms for the j30SAT set. The time is expressed in seconds and both axes are in logarithmic scale.

Algorithm 5 Assign variables algorithm

Require: The instance is feasible.
Input: LB, UB, instance data (INS), preprocessing data ($PREP$).
Output: Optimum makespan

```
ENC ← encode_MRCPSP(LB, UB, INS, PREP)
smt_assert_encoding(ENC)
SAT ← smt_check()
if SAT then
  MODEL ← smt_get_model()
  MAKESPAN ← smt_get_makespan(MODEL)
  UB ← MAKESPAN − 1
end if
while SAT and UB ≥ LB do
  PREP ← compress_time_windows(UB, PREP)
  smt_assert(S_{n+1} ≤ UB)
  smt_assert_domain_reduction(INS, PREP) //Clauses (S_i ≤ LS_i)
  smt_assert_trivial_variables(INS, PREP) //Clauses (¬y_{i,t})
  SAT ← smt_check()
  if SAT then
    MODEL ← smt_get_model()
    MAKESPAN ← smt_get_makespan(MODEL)
    UB ← MAKESPAN − 1
  end if
end while
if SAT then
  return UB
else
  return UB + 1
end if
```

Algorithm 6 Minimize encoding algorithm

Require: The instance is feasible.
Input: LB, UB, instance data (INS), preprocessing data ($PREP$).
Output: Optimum makespan

```
ENC ← encode_MRCPSP(LB, UB, INS, PREP)
smt_assert_encoding(ENC)
SAT ← smt_check()
if SAT then
  MODEL ← smt_get_model()
  MAKESPAN ← smt_get_makespan(MODEL)
  UB ← MAKESPAN − 1
end if
while SAT and UB ≥ LB do
  PREP ← compress_time_windows(UB, PREP)
  ENC ← encode_MRCPSP(LB, UB, INS, PREP)
  smt_assert_encoding(ENC)
  SAT ← smt_check()
  if SAT then
    MODEL ← smt_get_model()
    MAKESPAN ← smt_get_makespan(MODEL)
    UB ← MAKESPAN − 1
  end if
end while
if SAT then
  return UB
else
  return UB + 1
end if
```

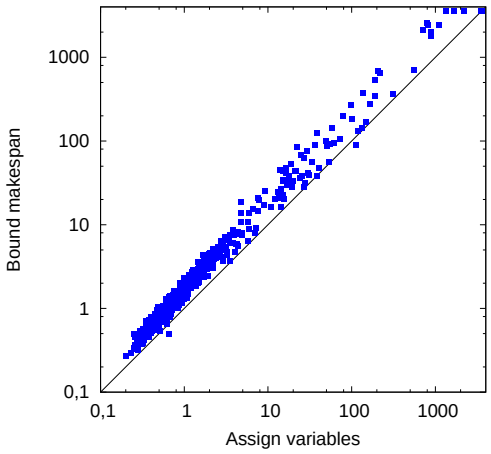

Figure 9.6: Comparison of time of *bound makespan* and *assign variables* algorithms for the j30SAT set. The time is expressed in seconds and both axes are in logarithmic scale.

Using only these constraints and (9.2), the theory solver is able to propagate the same information that we are asserting:

$$S_i \leq S_{n+1} - l_{i,n+1} \leq UB - l_{i,n+1} \qquad \forall A_i$$

We get a different result when comparing the time of *bound makespan* and *assign variables* algorithm. It can be seen in Figure 9.6 that the former requires significantly more time in almost all the instances than the latter.

Finally, Figure 9.7 shows the comparison between the *bound makespan* and the *minimize encoding* algorithms. For the easiest instances the latter is in clear disadvantage compared to the former, whereas the opposite happens with the instances requiring more than 100 seconds. This is happening because the computation of the encoding and the initialization of the SMT solver on each iteration penalize the total execution time of the iteration. However, for the hard instances this initialization time becomes negligible compared to the time needed to solve the last iterations, and the *minimize encoding* goes much faster in those cases.

9.3.2 Mixed Strategies

The time required for the easiest instances with the *minimize encoding* algorithm could be reasonable for many environments, especially if we compare it with the time needed to solve the hardest instances. Nevertheless, there is a time penalization for trying to simplify a query to the SMT solver that turns out to be very easy to compute. This leads to the conclusion that it is not worth to build new simplified encodings for easy to solve instances. An alternative that has shown good performance is to force the value of the variables as they become trivial. A more flexible approach are mixed algorithms, which could serve to exploit the best of each approach at each time. Let us analyse this idea. Figure 9.8 shows the computation time of the *smt_check()* operation in each iteration of the *assign variables* algorithm, for some of the hardest instances (with a total computation time to be solved higher than 1000 seconds). The leftmost point of each function corresponds to the starting UB, and the rightmost point to the last check that proves optimality. Both axes have been normalized to make all the instances comparable. Figure 9.9 contains the same results with y axis in logarithmic scale.

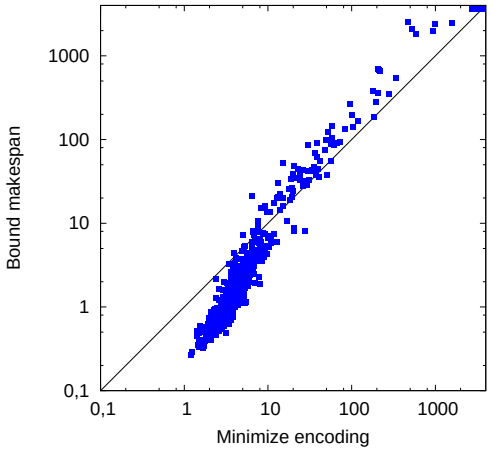

Figure 9.7: Comparison of time of *bound makespan* and *minimize encoding* algorithms for the j30SAT set. The time is expressed in seconds and both axes are in logarithmic scale.

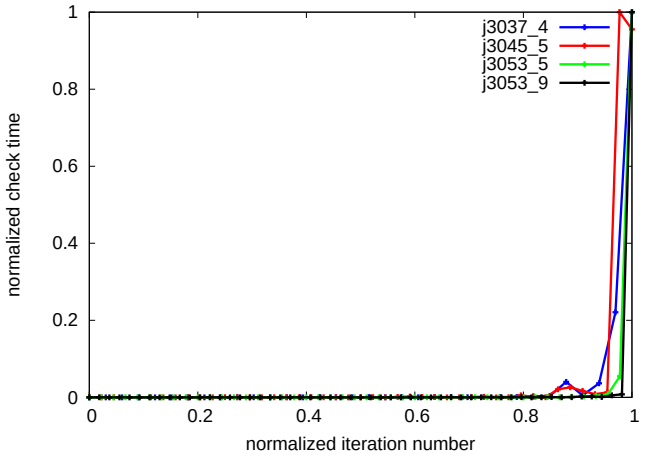

Figure 9.8: Execution times for the *assign variables* algorithm for some of the hard instances of j30SAT set. The time is expressed in seconds and both axes are normalized

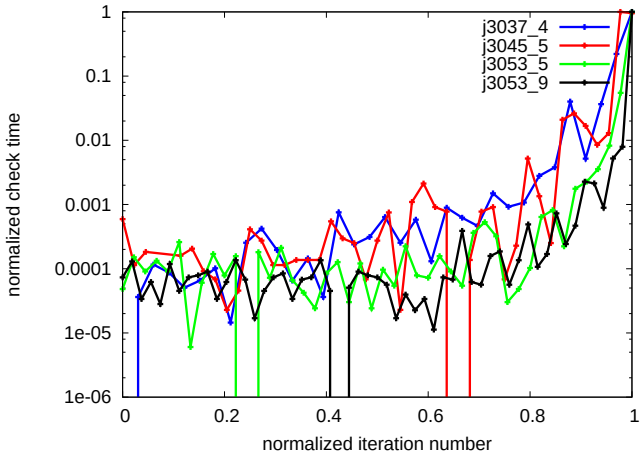

Figure 9.9: Execution times for the *assign variables* algorithm for some of the hard instances of j30SAT set. The time is expressed in seconds, both axes are normalized, and the y axis is in logarithmic scale.

These plots show that approximately the first 80% of the iterations require small computation time and there is almost no growth of it, which means that it is easy for the solver to find schedules that fit the given UB. On the other hand, in the last 5% of the iterations the computation time increases substantially. This evolution of the computation time through the iterations suggests that we could try to anticipate when the minimization of the encoding is worth. One effective way to do it would be recording the computation time needed for the check in every iteration, and once it becomes significantly bigger than the encoding and initialization time, change of strategy for the next iterations. For the easiest instances, the strategy would never be changed. We are going to study the following mixed algorithm:

Mixed strategy: It corresponds to Algorithm 7. The search starts behaving like the *assign variables* strategy. Once it estimates that it is becoming hard for the SMT solver to find a solution for the given UB, it changes to the *minimize encoding* strategy. By doing this, it avoids to encode the instance and initialize the SMT solver every time while it is not worth, but it encodes the problem again for every UB once the time of initializing is negligible compared to the solving time. We switch of strategy when the computation time of the last check operation is bigger than the time required to encode the problem and initialize the solver multiplied by a constant factor c.

The difference between this new strategy and *assign variables* is that the first one prioritises the minimization of the encoding and the second one the reuse of the learning of the previous iterations.

We have evaluated the performance of the mixed strategy on the j30 set, setting the parameter $c = 10$, thus requiring an order of magnitude of difference to switch of strategy. The first result that we can appreciate (see Figure 9.10) is that the *mixed strategy* effectively serves to distinguish the hardest instances from the easiest, and hence avoiding the initialization time when it is not worth (cf. Figure 9.7). Now, we have two different approaches, namely *assign variables* and *mixed strategy*, that improve the performance of the optimization process by reducing the complexity of the problem as it gets close to the optimum makespan, and work well both for easy and hard instances. Figure 9.11 contains the comparison of times between them. Obviously they have the

Algorithm 7 Mixed strategy

Require: The instance is feasible.
Input: LB, UB, instance data (INS), preprocessing data ($PREP$), factor of difference between encoding time and check time (c).
Output: Optimum makespan

```
CHANGED ← false
T_ENC ← Record the computation time of the following two operations
ENC ← encode_MRCPSP(LB, UB, INS, PREP)
smt_assert_encoding(ENC)
T_CHECK ← Record the computation time of the following operation
SAT ← smt_check()
if SAT then
  MODEL ← smt_get_model()
  MAKESPAN ← smt_get_makespan(MODEL)
  UB ← MAKESPAN − 1
end if
while SAT and UB ≥ LB do
  PREP ← compress_time_windows(UB, PREP)
  if CHANGED or T_CHECK ≥ c · T_ENC then
    CHANGED ← true
    ENC ← encode_MRCPSP(LB, UB, INS, PREP)
    smt_assert_encoding(ENC)
  else
    smt_assert(S_{n+1} ≤ UB)
    smt_assert_domain_reduction(INS, PREP) //Clauses (S_i ≤ LS_i)
    smt_assert_trivial_atoms(INS, PREP) //Clauses (¬y_{i,t})
    N_R ← smt_get_restarts()
    N_F ← smt_get_forgets()
  end if
  T_CHECK ← Record the computation time of the following operation
  SAT ← smt_check(ENC)
  if SAT then
    MODEL ← smt_get_model()
    MAKESPAN ← smt_get_makespan(MODEL)
    UB ← MAKESPAN − 1
  end if
end while
if SAT then
  return UB
else
  return UB + 1
end if
```

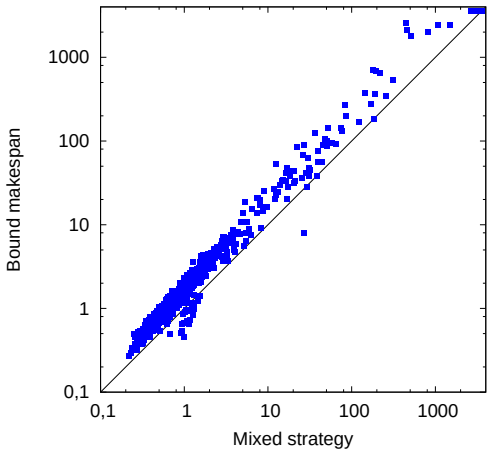

Figure 9.10: Comparison of time of *bound makespan* and *mixed strategy* algorithms for the j30SAT set. The time is expressed in seconds and both axes are in logarithmic scale.

same performance (with small variability due to the randomness of the solver) for the easiest instances, in which case the *mixed strategy* behaves exactly like *assign variables*. In the hardest instances there is no clear victor, what leads to the conclusion that both the simplification by variable assignment plus the reuse of the learning, and the minimization of the encoding, are equally good approaches to boost the optimization of the MRCPSP.

9.3.3 Quantification of the Simplification

It is important to notice that in this chapter we are simplifying the work to the SMT solver by using problem specific properties that the solver ignores. It does not mean that the SMT solvers are unable to reduce the problem size as they get close to the optimum solution, nor that they cannot discover trivial values for the variables. In fact, Yices performs clause database reduction and assignment of trivial Boolean variables before starting a satisfiability check. Nevertheless, it can only be done by reasoning over the clauses and the expressions of the theory, not by being aware of the problem itself. What we have shown is that can we can exploit this problem information to further help the SMT solver in the simplification of the problem, and save it the work of simplifying the problem. The following results help to explain the previous time results, by analysing how the problem size and complexity evolves for each of the studied strategies.

In Table 9.1 we can see how the problem size varies for each approach in terms of average number number of variables and average number of clauses among all the instances in the set. The first column contains the values for the first *UB* in the first iteration (that is the same in all algorithms). The other columns contain, for each algorithm, the values of the last iteration. In order to compute the averages with equality of conditions, we only consider the instances that have been solved without exceeding the time-out with all the algorithms.

We can see that the sizes are much small in the *minimize encoding* algorithm than in the other approaches. The *mixed strategy* has values similar to *assign variables* because the set of hard instances in which the switch of strategy is performed is small compared to the set of easy instances. It is also interesting to notice that the solver is able to reduce the number clauses with all the algorithms that do not do it explicitly, specially for the *assign variables* algorithm. This is not the case for the number of variables, but the results have shown that it is not a problem

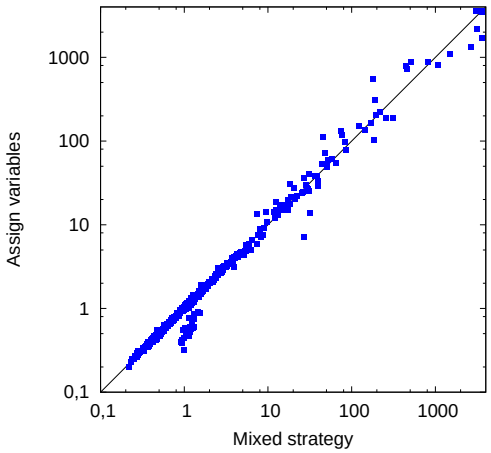

Figure 9.11: Comparison of time of *assign variables* and *mixed strategy* algorithms for the j30SAT set. The time is expressed in seconds and both axes are in logarithmic scale.

Algorithm	Start UB	Bound makespan	Compress time windows	Assign variables	Minimize encoding	Mixed startegy
Bool vars	39951,76	39951,76	39951,76	39951,76	9521,26	35587,33
Clauses	78702,05	67563,65	61984,62	51849,84	17517,33	51393,56

Table 9.1: The first column contains the average number of Boolean variables and the number of clauses of the *time ite* encoding for the j30SAT instances with the *UB* given by PSS. The other columns contain the number of Boolean variables and number of clauses at the last optimization step, for every optimization algorithm.

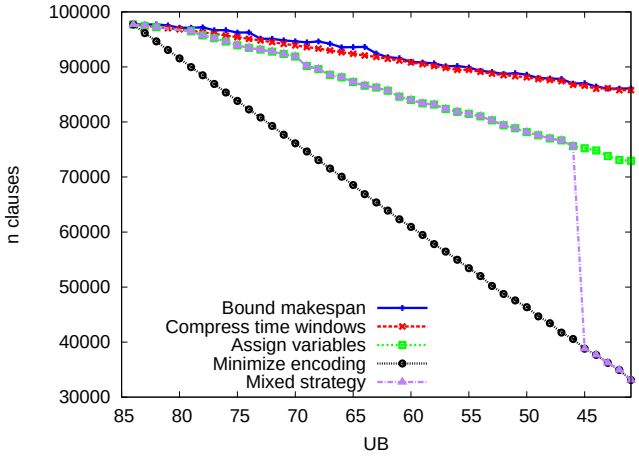

Figure 9.12: Evolution of the number of clauses with the different algorithms through the iterations for the instance *j3045_5*.

if we assign their value. Figure 9.12 shows the evolution through the iterations of the number of clauses respectively for instance *j3045_5*.

Chapter 10

Decision Heuristics

In the previous chapters we have presented several methods that treated the SMT solver as a black box, i.e. we have only used the API to pass encodings to the solver and to guide the optimization procedure. In this chapter we are going to study how to tune an SMT solver to adapt its internal algorithms to our purpose. In particular, we want to study the effects of defining ad-hoc heuristics in the selection of the variables of *decide* operations. VSID [28], the heuristic currently used by SAT/SMT solvers, has been well studied and refined for giving a good performance to the conflict driven solvers, and its nature is in fact to deal with the conflictive variables. This makes evident the difficulty of finding heuristics that could outperform the VSID heuristic, but the work in this chapter is a first step in this line that will help to better understand the behaviour of the solver for the given encodings. Section 10.1 explains the basic implementation details of Yices 2, putting special emphasis on the decision of variables. In Section 10.2 we present a modification of the implementation to support user-defined heuristics for the decision of variables. In Section 10.3 we present some heuristics for the decision of variables for the MRCPSP. Finally, in Section 10.4 we expose some initial performance results and analysis of the behaviour of the solver.

10.1 Study of Yices 2 Implementation

Yices 2's source code is free to use and modify for academic purposes. It is fully written in C language, and uses the GNU Multiple Precision Library (GMPLib) and the GNU gperf library of perfect hash function generators. It can be obtained at its web page (`http://yices.csl.sri.com/`), together with the manual and the API reference. A first step previous to define heuristics for the decision of variables has been to study the implementation of Yices.

As explained in the manual, the architecture of Yices 2 consists of three main modules: the *term database*, the *context management* and the *model management* (see Figure 10.1). The *term database* stores all the variables and expressions defined through the API, dealing with multiple or equivalent definitions of a same term. The *context management* module provides and manages contexts, which are central data structures to assert formulas and check their satisfiability. It contains internalization mechanisms to map the terms and formulas provided by the user to internal Boolean and theory variables, and CNF formulas, which are the data structures used by the CDCL(T) solver also included in this module. Finally, the *model management* module offers mechanisms to query the results of models obtained with satisfiability checks. The internal solver is composed of a CDCL SAT solver and some theory solvers that support the following theories: uninterpreted functions with equality, integer and real linear arithmetic, theory of bitvectors, and theory of arrays (see Figure 10.2).

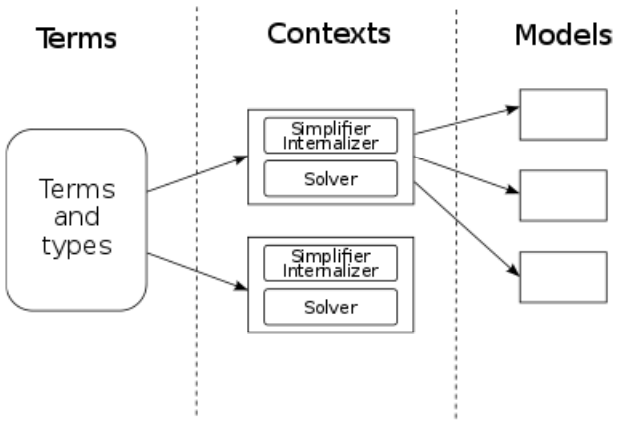

Figure 3.1: Top-level Yices 2 Architecture

Figure 10.1: Top-level Yices 2 Architecture. Figure obtained from [17].

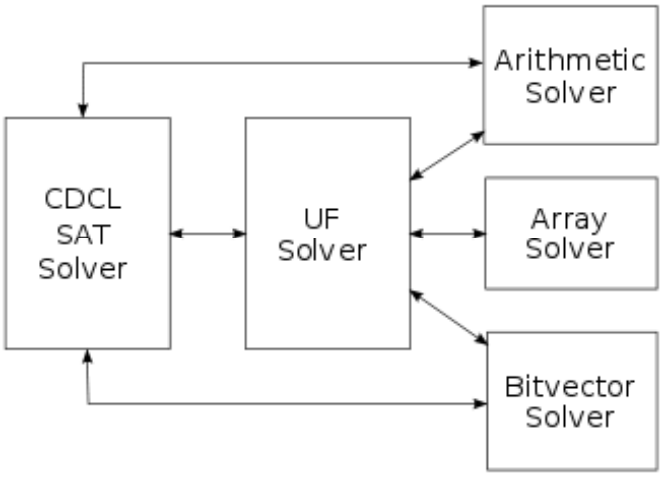

Figure 10.2: Yices 2 solver components. Figure obtained from [17].

Neither the manual nor the API are intended to describe the source code. Nevertheless the code is well self-documented and the different functions and data structures are well described in the comments. An important work during this thesis has consisted in studying and understanding this state-of-the-art implementation of an SMT solver. Now the most representative characteristics are described, specially the ones closely related with the heuristic of decision of variables.

10.1.1 SMT Core

The CDCL SAT Solver component is mainly composed by the *smt_core*, that is a data structure that contains all the clauses, variables and data used in the solving process. Also in the core, there are implemented the well-known CDCL methods presented in Chapter 3. The most relevant data contained in the *smt_core* are:

- Sizes of the data, such as number of variables or number of problem clauses, learned clauses and total clauses.
- Boolean values of the variables, antecedent for its propagation if that's the case (propagation of a certain clause, explanation of the theory solver), and its decision level.
- Current decision level.
- Activity of the lemmas, to be used when choosing which lemmas to forget.
- An assignment stack for the partial model under construction.
- A priority queue for the VSID implementation of the selection of variables to decide.

Every Boolean variable is identified by an integer in the range $[0, nvars - 1]$, where *nvars* is the number of variables. Most of the data that have an entry for every Boolean variable is stored in arrays of size *nvars*, that are indexed by the identifier of the corresponding variable. For instance, the Boolean value and the decision level of the variables are stored in arrays indexed by identifier.

For a variable x, its corresponding positive literal is identified by the integer $2x$, and the negative literal by $2x + 1$. The value of a Boolean variable is represented with two bits, that have the following meaning:

00: The variable is not assigned, and its preferred value is *false*.

01: The variable is not assigned, and its preferred value is *true*.

10: The variable is assigned to *false*.

11: The variable is assigned to *true*.

This representation of the values is intended to store the candidate value for an unassigned variable if it is used in a decide operation.

As VSID requires, the *smt_core* keeps track of an activity[1] measure for each Boolean variable. The activity of a variable quantifies the influence of the variable in finding conflicts, and the variable selected to make a decide operation is always the unassigned one with greatest activity at that moment. There is a data type used for the *smt_core*, named *var_heap_t*, defined as follows:

[1] In this chapter the term *activity* appears many times. The *activity of a variable* is a key metric for the VSID heuristic. This activity has nothing to do with the activities of an MRCPSP.

```
typedef struct var_heap_s {
 uint32_t size;
 double *activity;
 bvar_t *heap;
 int32_t *heap_index;
 uint32_t heap_last;
 double act_increment;
 double inv_act_decay;
} var_heap_t;
```

This data type contains all the data needed for the implementation of the VSID heuristic. The meaning of each attribute is the following:

size: number of variables of the problem.

activity: activity for every variable in the problem, independently of whether it is contained in *heap*.

heap: array implementation of a heap (priority queue).

heap_index: array containing the index of every variable x in the array *heap*. If x is not in the heap, $heap\_index[i] = -1$.

heap_last: index of the last element in the heap. It is needed for the implementation of the operations of the heap.

act_increment: increment of the activity of the variables. Whenever a variable v appears in the analysis of a conflict, its activity value is incremented as follows:

$$activity[v] = activity[v] * act\_increment$$

inv_act_decay: inverse of the decay factor of the activity of the variables. This value is greater or equal than 1. After a conflict, the *act_increment* parameter is updated as follows:

$$act\_increment = act\_increment * inv\_act\_decay$$

The policy to maintain the variables in the heap is the following:

- When a variable is assigned to the partial model, it is not removed from the heap.
- When a variable is needed to make a decision, the top of the heap is retrieved. While the top variable is an already assigned variable, it is removed from the heap and the next top is retrieved. This process will finish either obtaining the unassigned variable with greatest activity if any, or when the heap is empty, what means that all the Boolean variables are assigned and hence we have obtained a Boolean model. Having a Boolean model does not necessarily mean that the problem is satisfiable: a last consistency check to the theory solver might be needed.
- Every time a Boolean variable is deleted from the model, either by a backtracking operation after a conflict, or by a restart, it is inserted without repetition in the heap.

10.1.2 Internalization of the Terms

The internal data structures of the *smt_core* are totally hidden to the user of the API. Instead of it, the data types that the user handles are *terms*. The following are some kinds of terms that can be used:

- Boolean constants (*true* or *false*) and constants of a theory (integer, real, etc.)
- Boolean variables of variables from a theory
- Boolean operations: not, and, or, implication, etc.
- Arithmetic operations between terms: sum, multiplication, etc.
- Arithmetic expressions (atoms) of the form: $t_1 \leq t_2$, $t_1 = t_2$, etc.
- If-then-else operations: $ite(t_1; t_2; t_3)$
- Other kinds of terms

Each term has a type associated. For instance, the Boolean variables and the arithmetic atoms have Boolean type, whereas integer variables have integer type. Any Boolean term is a formula, and the user can assert as many formulas to a context as needed before making a satisfiability check. The whole problem formulation is the conjunction of all the asserted formulas in a context.

Once a formula is asserted, it is the turn for the internalizer to convert the Boolean formula expressed with terms to a CNF formula. At this moment, the theory expressions are passed to the theory solver, and all term Boolean variables and term atoms are associated with a literal of an internal Boolean variable[1]. Let us put this relation between terms and internal Boolean variables more clear with an example. Consider the atom $x \leq 4$, where x is an integer variable. This is indeed a Boolean term, and it will be associated with an internal Boolean variable, say b. Then, Yices will make one of the two following mappings:

$$x \leq 4 \leftrightarrow b$$
$$x \leq 4 \leftrightarrow \neg b$$

In other words, the mapping is done with an internal Boolean variable and a polarity (i.e. with a literal). Let us assume that the top mapping is the one performed. Then, the term $\neg(x \leq 4)$ will also be mapped to the internal variable b, but to the opposite literal:

$$\neg(x \leq 4) \leftrightarrow \neg b$$

Moreover, the internalization process deals with polymorphisms, so the following mappings are also being defined:

$$x > 4 \leftrightarrow \neg b$$
$$\neg(x > 4) \leftrightarrow b$$

The relation between a Boolean term and its internal Boolean variable (if any) does not exist until a formula containing this term is asserted. The formulas can be asserted gradually, so the internal CNF formula is also created gradually.

Another important aspect is that it is not required that the asserted formulas are in conjunctive normal form, but it can be any formula whose parsing tree has a Boolean term in the root. For instance, the following would be a valid formula to assert:

[1] When we say *term Boolean variable* or *term atom*, we refer to terms corresponding to Boolean variables or atoms. When we say *internal Boolean variable*, we refer to Boolean variables in the internal CNF representation of the solver.

$$(x \leq 3 \wedge y \geq 5z + 2) \rightarrow b \vee (x = z)$$

To deal with generic formulas, a flattening process is done in which new auxiliary Boolean variables may be created.

Although the mapping between variables and terms is hidden to the user, it persists during the life of the context, so that the user can obtain a model after a satisfiability check of a satisfiable formula, and query the value of each one of the terms.

10.2 An Extension to Support User-Defined Heuristics

The main goal that we are pursuing is to evaluate alternative heuristics to VSID that take into account the meaning of the variables in the given encoding. Since we are dealing with different encodings, and the intention is to further investigate in this area with different kind of scheduling problems, we have designed a framework flexible enough to handle with diverse problems. The way to define the desired decision heuristics is by means of an extension of Yices API.

10.2.1 Extension of the API

We have designed two generic heuristics that can be used to define new and more specific heuristics. Namely they are the *order* heuristic and the *decide allowed* heuristics:

Order heuristic An order of the variables of the problem is given. The selected variable to decide will be the first in the order that is unassigned. It is not imposed that all the variables are contained in the order.

Decide allowed heuristic A set of variables that are allowed to be used to decide is given. Inside this set, which variable to chose to decide will be determined following the VSID heuristic, i.e., it will be the unassigned one with greatest activity. It is only allowed to select variables from this set unless all of them have a value assigned.

We are not enforcing that all the variables are in the order for the *order* heuristic, nor that all the variables are allowed to be decided in the *decide allowed* heuristic. Then, there is the possibility that all the variables in the order or the decide allowed variables are assigned and the solver still requires to make decisions. In this case, the VSID heuristic will be used to chose one of the remaining variables.

Moreover, we provide the functionality of determining for each one of the variables which will be the polarity when making a decision, i.e., which of the *true* (non negated) literal and the *false* (negated) literal of the variable is added as a decision to the model. Yices provides many global settings for choosing the polarity, which are applied to the whole set of variables. Exactly one of the following can be selected:

- Decide always the *true* literal.
- Decide always the *false* literal.
- Decide the *true* literal if the variable is a pure Boolean (i.e. it does not have an atom attached). Delegate to the theory solver otherwise.
- Decide the *false* literal if the variable is a pure Boolean. Delegate to the theory solver otherwise.
- Use the last value assigned to the variable.

- Use the last value if the variable is a pure Boolean. Delegate to the theory solver otherwise.

We extended Yices so that the user can specify one of the following behaviours independently for each one of the variables:

- Decide always the *true* literal.
- Decide always the *false* literal.
- Use the global setting.

Finally, there is a last introduced functionality that is specifying the initial activity of a variable. It is thought to be able to guide the VSID heuristic at the first steps of the search.

10.2.2 Implementation of the Extension

The first thing to take into account when extending the API is that it only deals with terms and not with internal Boolean variables. Nevertheless, we know that once we have asserted the formulas, the mapping is complete. We add new functions to the API to be called after the formulas have been asserted, so that we can directly store in the *smt_core* the information related with the newly defined heuristics. The following are the newly defined functions:

```
//ctx: context with asserted formulas
//t: a Yices term
//sign: output parameter, polarization of the internalization (i.e. negated
    literal or non-negated literal)
//       if t is mapped to x, sign = true.
//       if t is mapped to ¬x, sign = false.
//return identifier of the internal Boolean variable of term t.
//   If t is not a Boolean term, return -2.
//   If t is not internalized as Boolean variable in the context ctx return -1.
// This function is used to discover the relation with Boolean terms and
    internal Boolean variables (literals)
bvar_t custom_get_boolvar_of_term(context_t *ctx, term_t t, bool * sign);


//ctx: context with asserted formulas
//x: an internal Boolean variable in ctx
//If and only if allowed = true, put x in the set of decide allowed variables
//This function is used to specify the decide allowed heuristic
void custom_bvar_set_decide_allowed(context_t *ctx, bvar_t x, bool allowed);


//ctx: context with asserted formulas
//order: list of internal Boolean variables, which order will be followed when
//       deciding. The first in the list will be the first to decide, and the
//       last one the last to decide
//size: number of variables in order
//This function is used to specity the order heuristic
void custom_set_decision_order(context_t *ctx, bvar_t *order, int size);


//ctx: context with asserted formulas
//x: an internal Boolean variable in ctx
//mode: DECIDE_TRUE, DECIDE_FALSE or GLOBAL_SETTING
//This function especifies which polarity will be used when variable 'x' is
    decided
void custom_bvar_set_polarity_decision_mode(context_t *ctx, bvar_t x,
```

```
    polarity_decision_mode_t mode);


//ctx: context with asserted formulas
//x: an internal Boolean variable in ctx
//a: activity value
//Set activity value of x
void custom_bvar_set_activity(context_t *ctx, bvar_t x, double a);
```

These methods provide the functionality to obtain the internal Boolean variable corresponding to a term (if any), and to specify the *decide allowed* and *order* heuristics. It is also needed to specify which heuristic is going to be used. This is done with the following already existing function of the API of Yices:

```
yices_set_param(param_t *param, const char *name, const char *value);
```

This is the function used to specify the parameters of the solving process. We have added a new parameter named *decide-heuristic* that can take as values *vsid*, *decide-allowed* and *order* to specify the corresponding heuristic method.

Moreover, some modifications have been done to the *smt_core* to implement these new functionalities. Most data of the core is stored using few bits to save memory usage. Following this same fashion, a new array indexed by variable identifier has been added, named *heuristic_params*, where the bits of each entry have the following meaning:

bit 0 (less weight): 1 if the variable is in *decide allowed* set, 0 otherwise (for *decide allowed* heuristic).

bits 2-1: polarity decision mode. 00: global setting. 10: decide true. 11: decide false.

bit 3: 1 if the variable is contained in the decision order, 0 otherwise (for *order* heuristic).

We will say that a variable is a *priority variable* if it is either part of the *decide allowed* set, or it is included in the order. Otherwise, we will say that it is a non-priority variable. Both in *decide allowed* and *order* heuristics, the non priority variables will not be decided until all the priority variables are in the model. The *heap_t* data type of Yices is modified so that it maintains two heaps: *heap* for the non priority variables, and *heap_priority* for the priority variables.

```
typedef struct var_heap_s {
 uint32_t size;
 double *activity;
 bvar_t *heap;
 bvar_t *heap_priority; //New attribute
 int32_t *heap_index;
 uint32_t heap_last;
 uint32_t heap_priority_last; //New attribute
 double act_increment;
 double inv_act_decay;
} var_heap_t;
```

Each variable will be added, retrieved or removed from its corresponding heap according to whether it is a priority variable or not. This modification is enough to manage the *decide allowed* heuristic. Regarding the *order* heuristic, the *heap_priority* is also used to manage the order of decision of the variables. This is done by initializing the activity of the variables in the order so that the lower position in the order, the higher the activity. Then, this order is preserved by not updating the activities of the variables in conflicts, which are frozen.

10.3 Heuristics for the MRCPSP

We can find in the literature a variety of heuristics for scheduling problems, both for exact methods and for meta-heuristics. There are several heuristics for branch-and-bound methods that incrementally construct schedules (constructive heuristics). An overview of different heuristics can be found in [4]. A recurrent strategy is to first schedule the activities with shortest critical path from the starting activity (i.e. the ones with lower earliest start time). In [34] and [40] conflict-driven branching approaches are proposed for the assignment of start times of the activities. The latter is in fact the current state-of-the-art solver for the MRCPSP. The authors of [34] go a step further and propose a hybrid approach for lazy clause generation that at the beginning uses a constructive heuristic and then turns to use a conflict-driven heuristic. We have also seen in Section 9.1 that we can treat a component of the problem, namely the satisfaction of the non-renewable resource constraints, independently to find feasible schedules of modes (although they do not guarantee optimal schedules). In [5] the authors tackle another multi-mode variant of scheduling a problem in two steps, first finding suitable modes and then dealing with the other components of the problem. The *decide* operation in the SAT/SMT solvers can be seen as the equivalent of the branching of other approaches.

What we will do in this chapter is to define several heuristics for our encodings of the MRCPSP problem and study how the SMT solver behaves with them. Following the line of the heuristics of the literature, we are going to define heuristics that prioritize some variables depending on their meaning in combination of the VSID heuristic, and also evaluate constructive heuristics prioritizing the activities with leftmost time windows. The extension of the API that we have presented will serve us to define these heuristics, based on the *decide allowed* (Section 10.3.1) and *order* heuristics (Section 10.3.2). Since the heuristics are defined over the variables of the encoding, there are some heuristics that are exclusive for the *time* encoding and some that are exclusive for the *task* encoding.

10.3.1 *Decide Allowed* Heuristics

We have seen the generic behaviour of this kind of heuristics in Section 10.2. To define a new heuristic, it is only needed to define the set of decide allowed variables, from now on denoted by W.

Assign modes

In this heuristic we will prioritize the assignment of the modes to the activities. This heuristic can be applied both in the *time* and *task* encodings. The decide allowed set is defined as:

$$W = \{m_{i,o} \mid \forall A_i \in A, o \in \{1, \ldots, M_i\}\}$$

Bound start times

This heuristic will prioritize the atoms that bound the start time of the activities, i.e., the atoms with a form similar to $(S_i \leq k)$ or $(k < S_i)$, being k a constant. In particular we can find such atoms in the *time* formulation:

$$\left(\sum_{A_i \in A} \sum_{\substack{o \in \{1,\ldots,M_i\} \\ ES_i \leq t \leq LS_i + p_{i,o} - 1}} ite\left(sm_{i,o} \wedge (S_i \leq t) \wedge (t < S_i + p_{i,o});\ b'_{i,k,o};\ 0\right) \right) \leq B'_k$$

$$\forall R_k \in \{R_1, \ldots, R_v\}, \forall t \in \{0 \ldots UB\}$$

Notice that an atom of the form $(k_1 < S_i + k_2)$ where k_1 and k_2 are constants is equivalent to the atom $(k < S_i)$ where $k = k_1 - k_2$. Moreover, an atom of the form $(k < S_i)$ is equivalent to $\neg(S_i \leq k)$. Actually the internalization mechanism of Yices deals with these polymorphisms (as seen in Section 10.1.2), so it is enough to consider atoms of the form $(S_i \leq k)$ to capture all the internal Boolean variables.

The decide allowed set is defined as[1]:

$$W = \{S_i \leq k \mid \forall A_i \in A, ES_i \leq k \leq LS_i\}$$

This heuristic will only apply to the *time* formulation, since the *task* formulation does not contain any atom which assigns or bounds the values of the variables S_i.

Define $z^1_{i,j}$

This heuristic prioritizes the decision of $z^1_{i,j}$ variables for the *task* formulation. The decide allowed set is defined as:

$$W = \{z^1_{i,j} \mid \forall A_i, A_j \in A, i \neq j, \neg incomp(i,j)\}$$

Define $z^2_{i,j}$

This heuristic prioritizes the decision of $z^2_{i,j}$ variables for the *task* formulation. The decide allowed set is defined as:

$$W = \{z^2_{i,j} \mid \forall A_i, A_j \in A, i \neq j, \neg incomp(i,j)\}$$

Define $z^1_{i,j}$ and $z^2_{i,j}$

This heuristic is designed to prioritize the decision of both $z^1_{i,j}$ and $z^2_{i,j}$ variables for the *task* formulation. The decide allowed set is defined as:

$$W = \{z^1_{i,j}, z^2_{i,j} \mid \forall A_i, A_j \in A, i \neq j, \neg incomp(i,j)\}$$

[1]The atom $(t < S_i + p_{i,o})$, which is equivalent to $\neg(S_i \leq t - p_{i,o})$ can make k smaller than ES_i and hence not being included in the decide allowed set. But if that is the case, this atom becomes trivially true (by definition $S_i \geq ES_i$), and it is not included in the encoding.

10.3.2 *Order* Heuristics

In the case of *order*, a heuristic will be defined by a complete order of a subset of the variables. We will formally define it as a tuple (O, f), where O is the set of variables in the order, and f a function such that:

- $f(x, y)$ needs to be only defined for $x \neq y$ as we do not accept repetitions in O.
- $f(x, y)$ is true if x precedes y in the order and false otherwise.
- Exactly one of $f(x, y)$ and $f(y, x)$ is true.
- It is transitive: $f(x, y)$ and $f(x, z)$ implies $f(x, z)$

Prioritize leftmost activities

Similarly to the *bound start times* heuristic, this heuristic will prioritize bounding the start times of the variables, but in a specific order. O is defined as:

$$O = \{S_i \leq k \mid \forall A_i \in A, ES_i \leq k \leq LS_i\}$$

f is defined to first bound the activities with lowest ES, and for a same activity the smaller start times will be prioritized. In case of ties, a lexicographical order will be used.

$$f(S_i \leq k_1, S_j \leq k_2) = \begin{cases} true & \text{if } ES_i < ES_j \\ false & \text{if } ES_j < ES_i \\ i < j & \text{if } ES_i = ES_j \text{ and } i \neq j \\ k_1 < k_2 & \text{if } i = j \end{cases}$$

Prioritize leftmost times

This heuristic resembles the *prioritize leftmost activities* heuristic, but it prioritizes the atoms that enforce an start time of some activity to be small. O is defined equally:

$$O = \{S_i \leq k \mid \forall A_i \in A, ES_i \leq k \leq LS_i\}$$

f is defined to first decide the atoms which enforce a smallest start time for some activity. Ties are broken prioritizing the activity with lowest ES, and then following the lexicographical order.

$$f(S_i \leq k_1, S_j \leq k_2) = \begin{cases} true & \text{if } k_1 < k_2 \\ false & \text{if } k_2 < k_1 \\ true & \text{if } k_1 = k_2 \text{ and } ES_i < ES_j \\ false & \text{if } k_1 = k_2 \text{ and } ES_j < ES_i \\ i < j & \text{if } k_1 = k_2 \text{ and } ES_i = ES_j \end{cases}$$

10.4 Results

We have run a set of experiments on the set j30SAT to evaluate the performance of the previous heuristics, and obtain information of the effects of prioritizing each kind of variable in the decisions. We have used the *mult* encoding, which has shown to be the one that introduces less auxiliary variables. For each one of the previous heuristics, we have solved the benchmark set with three different settings for the polarity of the decided literals:

default Use the default polarity selection mode of Yices on all variables, which uses the last Boolean value assigned to the variable.

true Decide the true literal for the variables in W or O, and use Yices' default for the remaining variables.

false Decide the false literal for the variables in W or O, and use Yices' default for the remaining variables.

The results are shown in Table 10.1, which for each heuristic and polarity decision mode contains the first quartile, the median, the third quartile and the mean solving times, and the number of instances solved. The time of the unsolved instances has been counted as 3600. Each heuristic has been used with the encoding for which it is suitable, and the *Assign modes* heuristic has been used with both *time* and *task*. The table also contains the same results using the VSID heuristic with default polarity.

We can see that clearly the proposed heuristics are far from beating the VSID heuristic, since all the new heuristics solved less instances than VSID, and the quartiles are in general higher. However, there are some interesting results that could motivate further research in the use of heuristics specific for each encoding:

- The *assign modes* heuristic has significantly smaller quartiles than the other heuristics, both for *time* and *task*. For the *time* encoding, the third quartile is between 2 and 4,5 seconds depending on the polarity, whereas the *bound start times* has values of 12,90, 40,51 and 93,53. The differences are more pronounced in the heuristics for *task*, which has the third quartile smaller than 9 seconds with the *assign modes* heuristic and greater that 200 seconds with the remaining heuristics.

- The order heuristics behave significantly better with true polarity than with the other polarities, i.e., deciding that the activities start as soon as they can. Notice that if the polarity is set to false, less than the 75% of the instances are solved within the timeout of 3600 seconds with both order heuristics, whereas with true polarity the 75% of the instances are solved with at most 35 seconds approximately. The performance is also significantly better compared to using the default polarity selection mode. There are not significant differences between the two order heuristics.

- The *bound start times* heuristic, which prioritizes over the same set of variables as the order heuristics but without pre-establishing an order, solves more instances and has lower quartiles than the order heuristics. As happened with the order heuristics, the false polarity is the worst choice. In this case, the default polarity is the best to solve the hard instances (with lower third quartile and more instances solved), but the true polarity is the fastest to compute the easiest solutions (with lower first quartile and median).

- Between deciding over $z^1_{i,j}$, $z^2_{i,j}$, and both $z^1_{i,j}$ and $z^2_{i,j}$, the better choice is to decide over $z^2_{i,j}$. The default polarity is significantly better for the easiest instances, but the true

heuristic	polarity	25%	median	75%	mean	n. solved
VSID time	**default**	0,71	1,19	2,38	144,328	536
VSID task	**default**	0,48	1,095	3,81	202,686	525
Assign modes time	**default**	0,67	1,14	2,83	328,662	508
	true	0,60	1,025	2,32	322,996	508
	false	0,76	1,28	4,26	346,103	503
Assign modes task	**default**	0,48	1,06	8,89	419,371	493
	true	0,34	0,605	4,67	412,365	491
	false	0,53	1,25	8,58	400,137	495
Leftmost activities	**default**	1,22	3,97	461,80	822,659	438
	true	0,60	1,245	36,75	609,583	467
	false	1,28	5,15	3600	1047,33	403
Leftmost times	**default**	1,20	3,355	354,64	802,44	439
	true	0,60	1,22	34,73	620,206	465
	false	1,27	4,38	3600	1044,13	403
Bound start times	**default**	1,09	2,555	12,90	328,705	510
	true	0,66	1,42	40,51	376,786	506
	false	1,21	3,735	93,53	432,864	501
Define z1	**default**	0,39	1,12	322,49	782,296	443
	true	0,78	1,755	692,80	867,773	426
	false	0,46	1,51	676,10	853,768	432
Define z2	**default**	0,77	2,335	301,72	784,469	443
	true	4,19	16,38	284,55	743,114	453
	false	2,36	26,05	3600	1187,39	383
Define z1 and z2	**default**	0,61	2,055	256,18	764,989	446
	true	2,67	6,97	200,69	796,575	441
	false	2,04	5,735	2456,37	953,061	422

Table 10.1: Comparison of the different heuristics for the MRCPSP. It contains the first quartile, the media, the third quartile and the mean of the solving time of the instances in j30SAT, and the number of instances solved. The total number of instances is 552.

polarity is better for the hardest instances, having a better third quartile and solving more instances. Recall that deciding $z^2_{i,j} = true$ means stating that activity S_j starts before S_i has ended.

Chapter 11

Performance of the Techniques

In this chapter we evaluate the performance of our system using the best of the proposals introduced in this thesis. Concretely:

- We use all the already existing and new preprocessings in the encodings in CNF.
- We use the general optimization algorithm presented in Chapter 9, consisting in detecting infeasibility with the transformation into MRCPSP-SAT of the problem, obtaining the first upper bound with PSS heuristic, and finally optimizing the makespan. As optimization strategy we use the linear search, and concretely for *time* encoding we use the *assign variables* algorithm which has shown in Chapter 9 to be the most competitive (together with the *mixed strategy*).
- We do not use the new heuristics on the decision of variables, which are in an early stage and require further investigation.
- We use the *ite* encoding to solve the easiest benchmarks, and *BDD* encoding to solve the hardest instances.

We test our system in all the instances of PSPLib introduced in Section 2.3, and also on the set MMLIB50 to evaluate the time performance with instances harder than the ones in PSPLib.

Table 11.3 contains a summary of the execution times for all the sets of PSPLib except the j30 set, which are easier than j30 and MMLIB50. We show the quartiles and mean of the solving times, and the number of instances solved. We have solved them with the *ite* encoding, both for *time* and *task*, which has shown to be the most suitable for easy instances. The results show that all the instances of all the benchmark sets have been solved both with *time* and *task*, never exceeding 500 seconds of solving time. Excluding the m5 set, which is the hardest among all these sets, all the instances are solved with less than 100 seconds. As a general rule, we can see that *task* is faster than *time* for the easiest instances, with smaller values of the first, second and third quartiles, whereas *time* is faster for the hardest instances, having a lower fourth quartile.

For the other benchmark sets, which are the hardest ones, we compare our system with the state-of-the-art exact solver for the MRCPSP presented in [40]. We have run both solvers in the same machine to make the times comparable[1]. Table 11.1 contains the results on j30UNSAT, in which we use our transformation of the problem into MRCPSP-SAT. Table 11.2 shows the results for j30SAT and MMLIB50 sets. We have used the *BDD* encoding, both for *time* and

[1] We are grateful to Dr. Petr Vílim et. al. to have shared with us the source code of their solver so that we can compare the performance of the solvers.

task, which has shown to be the best for the hardest instances. It can be seen that we clearly outperform the solver of Vílim et. al. in detecting infeasibility of the instances, with differences of around two orders of magnitude. For j30 and MMLIB50, we can see that Vílim et. al. are much faster in solving the easiest instances, but our system scales better and we close more instances, being the difference specially pronounced in the MMLIB50 set. Also in MMLIB50 we can appreciate that *task* is in clear disadvantage in front of *time*, because the former encoding has its size proportional to the square of the number of activities, and these instances contain many of them (concretely 50).

solver	**25%**	**median**	**75%**	**max**	**mean**	**n. solved**
MRCPSP-SAT	0,11	0,195	0,52	8,83	0,571	88
Vílim et. al.	27,145	53,07	107,123	462,7	91,484	88

Table 11.1: Solving times in seconds and number of instances solved of the j30UNSAT set compared with the solver presented in [40].

set	**solver**	**25%**	**median**	**75%**	**mean**	**n. solved**
j30SAT	**time BDD**	1,42	2,8	5,077	89,478	544
	task BDD	0,69	2,52	4,325	192,2257	531
	Vílim et. al.	0,02	0,04	0,8625	98,173	543
MMLIB50	**time BDD**	3,93	9,87	155,79	692,17	445
	task BDD	5,205	23,345	366,39	792,27	435
	Vílim et. al.	0,05	1,345	1028,9425	894,20	415

Table 11.2: Solving times in seconds and number of instances solved of the j30SAT and MMLIB50 sets compared with the solver presented in [40]. The unsolved instances have been counted as 3600 seconds.

set	encoding	25%	median	75%	max	mean	n. solved
j10	**time**	0,05	0,07	0,09	0,46	0,0872626	537
	task	0,03	0,03	0,04	0,17	0,0368901	537
j12	**time**	0,07	0,09	0,13	0,82	0,123327	547
	task	0,03	0,04	0,05	0,41	0,0512797	547
j14	**time**	0,10	0,13	0,21	1,96	0,192668	551
	task	0,05	0,06	0,09	3,61	0,105154	551
j16	**time**	0,13	0,18	0,28	6,43	0,291782	550
	task	0,06	0,09	0,15	9,30	0,203018	550
j18	**time**	0,16	0,25	0,37	10,16	0,485906	552
	task	0,08	0,13	0,25	18,15	0,470598	552
j20	**time**	0,22	0,325	0,56	36,14	0,952256	554
	task	0,12	0,2	0,39	81,80	1,38693	554
c15	**time**	0,13	0,17	0,29	8,85	0,319637	551
	task	0,07	0,11	0,20	38,03	0,301125	551
c21	**time**	0,13	0,17	0,30	2,83	0,300725	552
	task	0,05	0,07	0,12	2,76	0,158859	552
m1	**time**	0,01	0,02	0,02	0,12	0,0190625	640
	task	0,01	0,01	0,02	0,18	0,0187344	640
m2	**time**	0,06	0,08	0,11	0,73	0,108295	481
	task	0,04	0,05	0,08	1,80	0,0821622	481
m4	**time**	0,23	0,35	0,71	23,23	0,826793	555
	task	0,09	0,15	0,29	33,20	0,521441	555
m5	**time**	0,34	0,595	1,67	422,29	4,1743	558
	task	0,13	0,235	0,62	468,01	2,8795	558
n0	**time**	0,09	0,15	0,25	23,27	0,298447	470
	task	0,04	0,07	0,12	48,74	0,263787	470
n1	**time**	0,13	0,18	0,29	15,75	0,30529	637
	task	0,06	0,08	0,14	28,32	0,214772	637
n3	**time**	0,13	0,18	0,28	3,94	0,307033	600
	task	0,06	0,08	0,15	5,70	0,212733	600
r1	**time**	0,08	0,10	0,14	1,71	0,140506	553
	task	0,05	0,06	0,09	5,77	0,113472	553
r3	**time**	0,18	0,27	0,48	5,82	0,517235	557
	task	0,07	0,11	0,23	10,38	0,289892	557
r4	**time**	0,24	0,36	0,66	16,30	0,744565	552
	task	0,09	0,14	0,25	23,40	0,453768	552
r5	**time**	0,31	0,49	0,88	19,55	1,1307	546
	task	0,10	0,16	0,35	55,73	0,706868	546

Table 11.3: Solving times in seconds and number of instances solved for all the benchmark sets of PSPLib except j30, with *time ite* and *task ite* encodings.

Chapter 12

Conclusions

In this thesis we have focused on the use of SMT to solve the Multimode Resource-Constrained Project Scheduling Problem (MRCPSP). We have employed as a starting point two already existing encodings for this problem, namely the *time* and the *task* encodings, and we have presented a series of improvements which have helped to reduce the size of the encodings in terms of number of variables, atoms and clauses, and in many times lead to an improvement of the performance. Some studies have been conducted to better understand the characteristics of these encodings. The main contributions of this thesis and the conclusions derived from them are:

- Some new preprocessings for the MRCPSP have been introduced, concretely the *energy precedences*, the *start time window incompatibilities*, the *resource incompatibilities* and the *resource disjunctions*. They have clearly shown to serve to reduce the size of the encodings, albeit not all of them suppose an important speedup for the solver. Specifically, it has been seen that the problem instances with a lot of start time window incompatibilities and resource disjunctions are in general easy to solve, and the use of the preprocessings does not make an important difference. Contrarily, the resource incompatibilities and the energy precedences appear in the hardest instances, and an improvement was observed in the *task* encoding.

- We have conducted some experiments to comprehend better the dependence of *time* on the UB and the independence of *task* on it. The results show that, with instances of very similar complexity from the point of view of the combinatorics of the possible solutions, but with different UB, *time* suffers from the increase in the size of the encoding while the *task* encoding presents always the same performance.

- We have seen that using CNF expressions helps to substantially reduce the size of the encodings, and in the case of *time* it also supposes a performance improvement.

- We have presented two alternative formulations for the constraints over the demands on resources that, contrarily to the original (*ite*) encoding, avoid the use of any *if-then-else* expression: *mult* which uses LIA expressions, and *BDD* which uses an implementation of Pseudo-Boolean constraints based on ROBDDs. The *mult* encoding has shown to be the most compact in terms of number of Boolean variables and clauses. The *ite* encoding showed to be the best approach for *time* in solving the easiest instances. But the most important results have been obtained with *BDD* both for *time* and *task*, with which although requiring an initialization time that penalizes the overall performance of the easiest

instances, it makes a noticeable improvement for the hardest instances, being able to solve more of them in the given timeout of 3600 seconds. The number of Boolean variables in *BDD* increases substantially due to the ROBDD implementation of the constraints over resources, but thanks to it we are able to reduce the number of atoms, and to move from the theory of LIA to IDL.

- We have proposed an ad-hoc optimization approach consisting on three main steps: determining the feasibility of the instance, find an upper bound for the makespan, and optimize it. We propose, for the procedure of determining the feasibility, a simplification of the problem that only needs to deal with the resources over non-renewable constraints, and that has shown to be very quick. The use of the PSS heuristic serves to quickly find an UB for the makespan significantly smaller than the trivial UB. Finally, we have shown, for the *time* encoding, how to simplify the problem to the SMT solver as a linear search optimization advances. Two different approaches to simplify the problem have shown an equally good performance. On the one hand, the *assign variables* algorithm asserts the value of some Boolean variables, and on the other hand, the *mixed strategy* (based on the *minimize encoding* algorithm) encodes the problem from scratch every time. Another difference between them is that the former preserves the learning of the previous iterations, whereas the latter discards it, and the results show that it does not make a real difference for this problem.

- Finally we have made an initial work towards the use of encoding-specific heuristics for the *decide* operations of the SMT solver. After a study of the implementation of Yices, a state-of-the-art SMT solver, it has been modified to include a framework that lets to define *decide allowed* and *order* based heuristics. Many heuristics for *time* and *task* have been studied. They are still far from being comparable to VSID, but the results suggest that there are important differences in prioritizing the use of some variables or with respect to prioritizing some other, and it also makes an important difference which polarity we use to decide these variables. If we prioritize the decision of a set of variables, in many cases deciding these variables with a same polarity turns out to be better than using the default criterion of the SMT solver. We have also seen that the *order* heuristics, which are more rigorous and emulate a constructive heuristic, are generally worse than the *decide allowed* heuristics which work in combination with VSID. Finally, the results show that deciding first the decision modes is generally better than prioritizing other sets of variables, especially for *task*. There is future work in this field to explore more refined heuristics and try to outperform VSID for the MRCPSP. The framework developed in this thesis could serve for this purpose.

The final report on the performance of our system with the currently used benchmark sets show that we are very competitive. All the instances of all the benchmark sets in PSPLib but j30 are solved with less than 500 seconds, and most of them with less than 50 seconds. In the comparison with the state-of-the-art solver of [40], we are able to solve more instances. In what regards to determining infeasibility, our system is significantly faster. Finally, it is also valuable that our system consists in two encodings that can be used on any off-the-shelf SMT solver (not only Yices), and that the preprocessings that we present can be used in other approaches to solve the MRCPSP different from SMT.

Bibliography

[1] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger. A new look at bdds for pseudo-boolean constraints. *Journal of Artificial Intelligence Research*, pages 443–480, 2012.

[2] J. Alcaraz, C. Maroto, and R. Ruiz. Solving the multi-mode resource-constrained project scheduling problem with genetic algorithms. *Journal of the Operational Research Society*, 54(6):614–626, 2003.

[3] C. Ansótegui, M. Bofill, M. Palahí, J. Suy, and M. Villaret. Satisfiability Modulo Theories: An Efficient Approach for the Resource-Constrained Project Scheduling Problem. In *Proceedings of the Ninth Symposium on Abstraction, Reformulation, and Approximation (SARA)*, pages 2–9. AAAI, 2011.

[4] C. Artigues, S. Demassey, and E. Neron. *Resource-constrained project scheduling: models, algorithms, extensions and applications*. John Wiley & Sons, 2013.

[5] F. Ballestín, A. Barrios, and V. Valls. Looking for the best modes helps solving the mrcpsp/max. *International Journal of Production Research*, 51(3):813–827, 2013.

[6] P. Baptiste. Constraint-Based Schedulers, Do They Really Work? In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 5732 of *LNCS*, page 1. Springer, 2009.

[7] M. Bofill, M. Palahı, J. Suy, and M. Villaret. Boosting weighted csp resolution with shared bdds. In *12th International Workshop on Constraint Modelling and Reformulation (ModRef 2013)*, pages 57–73, 2013.

[8] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *European Journal of Operational Research*, 149(2):268–281, 2003.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[10] R. K. Chakrabortty, R. A. Sarker, and D. L. Essam. Event based approaches for solving multi-mode resource constraints project scheduling problem. In *Computer Information Systems and Industrial Management*, pages 375–386. Springer, 2014.

[11] J. Coelho and M. Vanhoucke. A new approach to minimize the makespan of various resource-constrained project scheduling problems. In *Proceedings of the 14th International Conference on Project Management and Scheduling*, pages 1–4. TUM School of Management, 2014.

[12] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[13] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[14] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[15] U. Dorndorf, E. Pesch, and T. Phan-Huy. A Branch-and-Bound Algorithm for the Resource-Constrained Project Scheduling Problem. *Mathematical Methods of Operations Research*, 52:413–439, 2000.

[16] A. Drexl and J. Gruenewald. Nonpreemptive multi-mode resource-constrained project scheduling. *IIE transactions*, 25(5):74–81, 1993.

[17] B. Dutertre. Yices 2 manual. *Computer Science Laboratory, SRI International, Tech. Rep*, 2014.

[18] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[19] B. Dutertre and L. M. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[20] M. J. Geiger. Iterated variable neighborhood search for the resource constrained multi-mode multi-project scheduling problem. *arXiv preprint arXiv:1310.0602*, 2013.

[21] S. Hartmann. Project scheduling with multiple modes: a genetic algorithm. *Annals of Operations Research*, 102(1-4):111–135, 2001.

[22] A. Horbach. A Boolean Satisfiability Approach to the Resource-Constrained Project Scheduling Problem. *Annals of Operations Research*, 181:89–107, 2010.

[23] J. E. Kelley. The critical-path method: Resources planning and scheduling. *Industrial scheduling*, 13:347–365, 1963.

[24] R. Kolisch. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*, 90(2):320–333, 1996.

[25] R. Kolisch and A. Sprecher. PSPLIB - A Project Scheduling Problem Library. *European Journal of Operational Research*, 96(1):205–216, 1997.

[26] O. Koné, C. Artigues, P. Lopez, and M. Mongeau. Event-Based MILP Models for Resource-Constrained Project Scheduling Problems. *Computers & Operations Research*, 38:3–13, January 2011.

[27] O. Liess and P. Michelon. A constraint programming approach for the resource-constrained project scheduling problem. *Annals of Operations Research*, 157(1):25–36, 2008.

[28] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[29] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[30] A. O. El-Kholy. *Resource Feasibility in Planning*. PhD thesis, Imperial College, University of London, 1999.

[31] L. J. W. Pritsker, A. Alan B. and P. S. Wolfe. Multiproject Scheduling with Limited Resources: A Zero-One Programming Approach. *Management Science*, 16:93–108, 1996.

[32] A. Schutt, T. Feydy, P. Stuckey, and M. Wallace. Explaining the Cumulative Propagator. *Constraints*, 16(3):250–282, 2011.

[33] A. Schutt, T. Feydy, P. J. Stuckey, and M. Wallace. Why Cumulative Decomposition Is Not as Bad as It Sounds. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 5732 of *LNCS*, pages 746–761. Springer, 2009.

[34] A. Schutt, T. Feydy, P. J. Stuckey, and M. G. Wallace. Solving rcpsp/max by lazy clause generation. *Journal of Scheduling*, 16(3):273–289, 2013.

[35] R. Sebastiani and P. Trentin. OptiMathSAT: A Tool for Optimization Modulo Theories. In *Proc. International Conference on Computer-Aided Verification, CAV 2015*, volume 9206 of *LNCS*. Springer, 2015.

[36] R. Sowinski, B. Soniewicki, and J. Weßglarz. Dss for multiobjective project scheduling subject to multiple-category resource constraints. *European Journal of Operational Research*, 79:220–229, 1994.

[37] J. Suy Franch et al. A satisfiability modulo theories approach to constraint programming. 2012.

[38] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

[39] V. Van Peteghem and M. Vanhoucke. An experimental investigation of metaheuristics for the multi-mode resource-constrained project scheduling problem on new dataset instances. *European Journal of Operational Research*, 235(1):62–72, 2014.

[40] P. Vilím, P. Laborie, and P. Shaw. Failure-directed search for constraint-based scheduling. In *Integration of AI and OR Techniques in Constraint Programming*, pages 437–453. Springer, 2015.

[41] L. Wang and C. Fang. An effective estimation of distribution algorithm for the multi-mode resource-constrained project scheduling problem. *Computers & Operations Research*, 39(2):449–460, 2012.

[42] L. Wang and J. Liu. Solving multimode resource-constrained project scheduling problems using an organizational evolutionary algorithm. In *Proceedings of the 18th Asia Pacific Symposium on Intelligent and Evolutionary Systems, Volume 1*, pages 271–283. Springer, 2015.