university of groningen / faculty of economics and business

Solving resource-constrained project scheduling problems subject to no-overlap constraints using boolean satisfiability encoding

Author:
Maarten E. de Jager
Supervisor:
dr. N. van Foreest
Second assessor:
prof. dr. R.H. Teunter

Abstract

This thesis proposes a Booleaan Satisfiabiliy Problem (SAT) encoding for finding heuristic solutions to the Resource Constrained Project Scheduling Problem (RCPSP) subject to an additional set of no-overlap constraints. The aim of such a scheduling problem is to schedule a set of tasks taking into account precedence, resource and no-overlap constraints. Tasks which share a no-overlap constraint cannot be executed in parallel. We compare the performance of our encoding to that of a Mixed Integer Linear Program (MIP) and find that our SAT encoding is relatively efficient at dealing with no-overlap constraints. Furthermore, we find that using a SAT approach for finding feasible schedules subject to a planning horizon can be competitive, especially for larger and heavily resource-constrained projects and projects subject to many no-overlap constraints. SAT performance seems particularly sensitive to the given planning horizon, both very tight and wide planning horizons significantly increase computing times.

Operations Research Master Thesis

Contents

1 Introduction

Many, especially large, industrial projects require scheduling of tasks in order to be completed efficiently. Due to this importance, project scheduling has been the subject of much research aiming to improve the quality of schedules and the time in which these can be computed.

In a general setting, a project only has limited amount of resources. Resources are often present in the form of workforce, machines or materials. Each task has a duration expressed in a discrete number of time units referred to as periods, for example days, hours or minutes. The general goal of such scheduling is to complete all tasks in minimum time, taking into account available resources. An optimal schedule thus minimizes the number of periods in which all tasks can be completed. The completion time of the last task is referred to as the makespan of the schedule.

As projects get more complex, some tasks are subject to precedence constraints. Before a task can be started, all its predecessor tasks must be completed. A practical example of such a constraint would be an engineering project for which certain components need to be installed prior to others.

Additionally, some tasks are subject to no-overlap constraints. Two tasks that share a no-overlap constraint cannot be executed in parallel, that is, at most one of the two tasks can be active during a given period. Furthermore, some tasks can be constrained by start and due dates, that is, some tasks can only be executed in a certain time periods.

Even though the scheduling problems with no-overlap constraints bear much practical relevance, only limited research considers this special case. Hartmann (2010) provides an extensive overview of variations and extension of the general scheduling problem, yet the addition of no-overlap constraints is only mentioned very briefly. One of the few papers that does consider no-overlap constraints is Vanhoucke (2016) which extends the search procedure proposed by Coelho & Vanhoucke (2011).

To find the schedule with minimum makespan of regular scheduling problems without no-overlap constraints many algorithms have been proposed. Due to the NP-completeness of this optimization problem only small instances can be solved to optimality. These algorithms are often also capable of finding optimal schedules in case no-overlap constraints are present, however performance is generally sub-optimal as most algorithms are not specifically developed to deal with no-overlap constraints.

To solve larger industrial sized problem, the use of heuristics is required. Hartmann and Kolisch (2000) and Hartman and Kolisch (2006) provide an review of existing heuristics and find that meta-heuristics perform best. These heuristics mainly aim to find a good quality schedule given a certain computing time limit.

Alternatively, an inverse problem statement can be formulated as follows. Instead of finding an optimal schedule within a certain limit of computing time, find a schedule that completes all tasks within a given planning horizon. Any schedule with a makespan smaller than the planning horizon is a feasible solution. In case such a schedule can be generated, the problem is called satisfiable and the solution is given by the schedule that satisfies the problem. In case no schedule can be found that completes all tasks within the given planning horizon, the problem is call unsatisfiable.

This reformulation of the original scheduling problem allows the problem to be modeled as a SAT problem. Constraints of the scheduling problem are presented in SAT encoding using clauses. A SAT solver attempts to find some feasible schedule such that all clauses are satisfied. This SAT approach can be used to both find heuristic schedules and optimal schedules. In the latter case, the planning horizon is reduced until the problem is no longer feasible and thus an optimal schedule has been found.

Due to significant continuous improvements, SAT approaches to some specific scheduling problems can compete with existing exact solutions and heuristics. As a result SAT solvers have recently gained some attention in project scheduling research, however remain under-utilized. Formulating the resource constraint project scheduling problem as a SAT problem has been proposed by some papers with varying degrees of success. Most focus on finding exact solutions to the RCPSP such as Horbach (2010). To the best of our knowledge, no paper has proposed heuristics for solving the

RCPSP with no-overlap constraints using SAT solvers.

One advantage of using SAT encoding for solving such problems over existing heuristics such as genetic algorithms is adaptability. Introducing new constraints to the scheduling problem in SAT encoding can easily be done by adding new clauses to the boolean formula and does not require changes to the search algorithm. Furthermore, compared to more advanced exact solutions and heuristics SAT encoding approaches tend to be easier to implement. One considerable downside of solving scheduling problems with SAT encoding is that SAT solvers cannot return the optimal solutions found so far within a given computing time limit as the planning horizon in the model is static. Additionally, SAT encoding this particular scheduling problem relies heavily on an estimate of the upper bound of the scheduling problem. Too strict of an upper bound will not yield a feasible solution, too wide of an upper bound and computing times are significantly increased.

This paper aims to use SAT encoding to efficiently find feasible solutions for large RCPSPs subject to no-overlap constraints. The paper is ordered as follows. Section 2 discusses existing literature on RCPSPs and the use of SAT solvers in the context of project scheduling. Section 3 formally introduces the scheduling problem. Section 4 proposes a SAT encoding for finding feasible schedules for RCPSPs subject to no-overlap constraint. Section 5 discusses the Z3 solver used for this thesis. Section 6 discusses performance of the proposed SAT encoding using a generated set of RCPSPs instances and compares the performance with that of MIP solver Gurobi. Section 7 concludes the paper and proposes possible improvements and variations for future work.

2 Literature

The RCPSP introduced by Pritsker et al. (1969) has been a widely researched problem in the project management field. Hartmann & Briskorn (2010) provides an extensive review of variations and extensions of the general RCPSP. Most developed variations can be divided into 3 classes, alterations of activity concepts (e.g. activity preemption, multiple modes and setup times), alternative network concept (e.g. time lags and logical dependencies) and extensions to resource concepts (e.g. nonrenewable resources and doubly constrained resources).

The special case RCPSP with no-overlap constraints was first introduced by Kuster (2006). Here scheduling problems with no-overlap constraints are solved in the context of airport logistics. Several papers expand on this paper, however do not focus on larger sized scheduling problems with no-overlap constraints. Vanhoucke (2016) is one of the few papers that does consider industrial sized problems with no-overlap constraints. Though not the central focus of the paper, the extensive heuristic search procedure proposed by Coelho and Vanhoucke (2011) is extended to also cope with no-overlap constraints.

Literature on RCPSP solutions can broadly be divided into two groups, exact and heuristic solutions. Exact algorithms focus on finding the optimal makespan, that is, minimize the planning horizon required to complete all tasks. Many different algorithms have been proposed, each efficient at solving specific scheduling problems. Most literature on exact algorithms for the RCPSP is quite old as in recent years the focus has been on developing heuristic solutions due to its practical relevance. A selection of prominent exact algorithms is given by Erenguc & Conway (2001), Sprecher et al. (1997) and Demeulemeester & Herroelen (1992).

Blazewisz et al. (1969) has shown that finding the optimal solution to the RCPSP is NP-hard. Consequently, solving larger problems optimally is not feasible and heuristic solutions are required. Hartmann & Kolisch (2000) and Hartmann & Kolsich (2006) and more recently Pellerin et al. (2020) provide an extensive review of the most efficient heuristic algorithms. A large variety of methods are used in heuristic algorithms such as x-pass, tabu search, simulated annealing and genetic methods. Generally, more complex heuristics generate higher quality solutions at the cost of computing time and ease of implementation. Genetic algorithms such as Hartmann (1998) and more recently Goncharov & Leonov (2017) have been shown to be one of the most efficient algorithms, both in terms of quality solutions and computing time.

SAT solvers have only recently been used in algorithm proposals for solving general RCP-SPs and certain variants. Most papers use SAT solvers to find exact solutions to RCPSPs such as Horbach (2010) which focuses on the single mode RCPSP and Coelho & Vanhoucke (2011) which focuses on the multi-mode variant. SAT solvers are to this day rarely, if ever, used to develop heuristic solutions for RCPSP, even though the SAT problem inherently lends itself well for heuristic approaches. Furthermore, Kolisch & Sprecher (1995) have shown that increased resource-scarceness , such as no-overlap constraints, significantly increase the hardness of scheduling problems. As a result, exact solutions can only solve very small instances with no-overlap constraints making heuristic solutions even more relevant for RCPSPs with no-overlap constraints.

This thesis will focus on efficiently finding heuristic solutions to the single mode variant of the RCPSP subject to no-overlap constraints given a planning horizon using SAT encoding. The considered scheduling problem can be represented by $PS|PREC|C_{max}$ using notation introduced by Blazewicz (1969) and Brucker et al. (1999). Note that the goal of our heuristic approach is different from most approaches mentioned in Hartmann & Kolisch (2000) and Hartmann & Kolsich (2006). Instead of finding a best possible schedule given a computing time limit, we attempt to efficiently find a schedule given a planning horizon.

3 Problem formulation

Suppose we have a project consisting of a number of activities which all need to be completed before some deadline. Each activity requires some time to be completed and, once started, cannot be discontinued. The objective is to schedule all activities such that the project is completed before some planning horizon T. This schedule must take into account a number of constraints, namely: resource, precedence and no-overlap constraints.

Section 3.1 considers the most simple scheduling problem subject to none of these constraints. Section 3.2 adds a set of precedence constraints to the problem introduced in section 3.1. Section 3.3 introduces an additional set of resource constraints and section 3.4 considers the addition of a set of no-overlap constraints.

3.1 Trivial case

First consider a project consisting of just one task with a given duration d and limitless resources. Of course scheduling one task with given planning horizon T is trivial, however serves as a good setting to introduce notation. In order to formally define a model for this scheduling problem, we first need to discretize the planning horizon T into T periods of equal length. Consequently, duration d is expressed in periods. Each period can for example represent an hour, a day or a week. For each period t we introduce a boolean variable y_t defined as

$$y_t = \begin{cases} 1 \text{ If the activity starts in period } t \\ 0 \text{ If the activity does not start in period } t. \end{cases} \tag{1}$$

For this particular scheduling problem only one constraint is required. The activity needs to start in time for it to be finished before period T. This constraint can be written as

$$\sum_{t=1}^{T-d+1} y_t = 1. \tag{2}$$

This constraint states that the activity needs to have started at period $T - d + 1$ or earlier as otherwise it will not be finished in time. Furthermore, this constraint enforces the requirement that the activity can only start once.

Now consider the same type of project but with an activity set V consisting of $n > 1$ activities. Naturally constraint (2) needs to hold for every activity as well. Similarly to definition (1), we define a new set of variables given by

$$y_{it} = \begin{cases} 1 \text{ If activity } i \text{ starts in period } t \\ 0 \text{ If activity } i \text{ does not start in period } t. \end{cases} \tag{3}$$

Using these variables we can rewrite constraint (2) to hold for multiple activities such that

$$\sum_{t=1}^{T-d_i+1} y_{it} = 1 \quad i \in A. \tag{4}$$

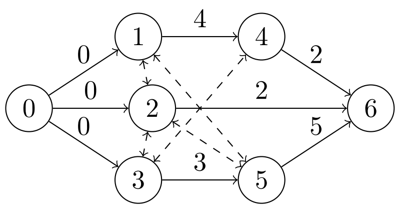
Similarly to constraint (2), this constraints ensures that each activity i starts in period $T - d_i + 1$ or earlier where d_i is the duration of activity i. Furthermore, this constraint ensures that any activity can only start exactly once.

3.2 Precedence constraints

Consider a project as described in the previous section, that is, a project with an activity set V and limitless resources. Now suppose the project is subject to a set of precedence constraints. Activities that are subject to a precedence constraint require some other activities to be finished before they themselves can start.

These precedence relations can be represented in an acyclical directed graph. Each node in the graph represents an activity and each directed arc from node i to node j represents a precedence relation indicating that activity i is required to be finished before activity j can be started.

Consider an example of such a graph G defined by $G = (V, A)$, where V is the set of activities (nodes) and A is the set of precedence relations (arcs), dashed arcs can be ignored for this section. The graph below presents G along with each activity i's duration d_i labeling outgoing arcs. For example, the arc (1,4) states that activity 1 is required to be completed before task 4 can start with activity 1 and 4 having durations 4 and 2 respectively. Usually, two dummy activity nodes, called the source and sink nodes, with durations 0 are added to ensure the project starts and ends with one activity. In this instance node 0 and node 6 are the source and sink node respectively.



Activity	Resource 1	Resource 2
1	4	0
2	2	1
3	0	1
4	3	1
5	4	1
Capacity	5	2

For each arc (i, j) it must hold that activity j cannot start before activity i is finished. The starting time of activity i, denoted by s_i can be written as

$$s_i = \sum_{t=1}^{T} t y_{it}. \tag{5}$$

If an activity i starts in period t and thus $y_{it} = 1$, ty_{it} is equal to the starting period of activity i. The finishing period f_i of activity i is equal to the starting period of activity i plus its duration, that is,

$$f_i = s_i + d_i. \tag{6}$$

Similarly to activity i, the starting period of activity j, s_j, can be written as

$$s_j = \sum_{t=1}^{T} t y_{jt} \tag{7}$$

In order for every arc (i, j) to ensure that activity j cannot start before activity i is finished we must have

$$f_i \leq s_j \quad (i,j) \in A \tag{8}$$

Finding a feasible schedule for projects that are only subject to precedence constraints and have limitless resources at disposal is relatively easy. Using the critical path algorithm proposed by Floyd (1962), a feasible schedule with minimal makespan can be computed quickly even for very large projects. This makespan can be used to compute a reasonable planning horizon for projects with additional constraints. Usually this planning horizon estimated by $T = k * t_{opt}$ where t_{opt} is the minimum makespan of the project with just precedence constraints.

3.3 Resource constraints

Now consider a project as in the previous section, but with limited resource. This project has a set of resources R^J at its disposal consisting of J different resource groups. The available capacity of resource group j is given by R_j. Each activity in the project now requires a set of resources during its execution. During this time, the required resources are not available for other activities to use. For each activity i in the activity set V, r_{ij} denotes how many resources activity i requires of resource group j. After an activity is finished, all its required resources are released and available again for other activities to use.

An example of a set of resource capacities and requirements is presented in table presented above. For example, the set of available resources is given by $R^J = \{R_1, R_2\} = \{5, 2\}$. The resource requirement of, for example, activity 1 for resource group 1 is given by $r_{1,1} = 4$.

Whether activity i is active in period t is denoted by a_{it} defined by

$$a_{it} = \sum_{u=t-d_i+1}^{t} y_{iu}. \tag{9}$$

where y_{iu} is as defined in definition 3 and $a_{it} = 1$ if activity i is active in period t and zero otherwise. The amount of resources an activity i uses of resource group j in period t is given by

$$a_{it} r_{ij} \tag{10}$$

The sum of resources used by all activities cannot exceed the total capacity in any period t and for any resource group j. This can be written as

$$\sum_{i=1}^{n} a_{it} r_{ij} \leq R_j \quad \forall j \in [1,..,J],\ \forall t \in [1,..,T] \tag{11}$$

3.4 No overlap constraints

Now we consider a new type of constraint to the general RCPSP. Consider the scheduling problem in the previous section, that is, a scheduling problem subject to precedence and resource constraints. In addition we introduce a set of no-overlap constraints O to the project.

The no-overlap constraints set O consists of pairs of activities. Two activities i and j that share a no-overlap constraints (i, j) cannot be executed in parallel. As such they need to be executed in sequence in no specific order. An example of such activities in the context for airport logistics are refueling and certain safety checks. Refueling is not required to perform the checks nor vice versa, but performing them simultaneously is impractical or dangerous.

An example of a set of no-overlap constraints is presented in figure 3.2. Each dashed edge between an activity i and j represents a no-overlap. For example, activity 1 and 2 share a no-overlap constraint and thus cannot be executed in parallel.

One straightforward way to formally define a no-overlap constraint is to introduce constraints similar to resource constraints (11). The main difference here is that we introduce a resource group for each no-overlap constraint with a capacity of 1. Each of the two activities sharing a no-overlap constraint will require this resource and consequently the two activities are restricted from being executed in parallel. This constraint can be written as

$$a_{it} + a_{jt} \leq 1 \qquad (i,j) \in O,\ t = [1,..,T] \tag{12}$$

An RCPSP with no-overlap constraints is thus not a class separate from the general RCPSP but rather a special case. Large RCPSP with many no-overlap constraints are essentially equal to RCPSP with very many resource constraints. A feasible solution for an RCPSP planning horizon T subject precedence, resource and no-overlap constraints must satisfy constraints (4), (8), (11) and (12).

4 SAT encoding

The SAT problem can be defined as follows. Given a boolean formula ϕ, find some assignment of variables such that ϕ evaluates to true. If such an assignment can be found, the SAT problem is called satisfiable and if no such assignment can be found, it is called unsatisfiable. This boolean formula has two main components:

1. **literals**, a literal is either a boolean variable (x) called a positive literal or a negation of a boolean variable ($\bar{x}$) called a negative literal, and
2. **boolean operators** such as $\wedge$ (AND), $\vee$ (OR), $\implies$ (implies) and $\iff$ (if and only if) which combine literals to form the boolean formula.

For most SAT problems to be solved efficiently, the boolean formula must be of Conjunctive Normal Form (CNF). A formula in CNF only contains $\wedge$ (AND) and $\vee$ (OR) operators. Furthermore, the structure of the formula must be an AND of ORs, that is, a conjunction ($\wedge$) of disjunctions ($\vee$). Each set of disjunctions is referred to as a clause. A boolean formula in CNF thus has the form $c \wedge c \wedge c \wedge ..$, where each c is a clause consisting of disjunctions of literals.

For example, consider a clause given by $x \vee \bar{y}$ stating that x must be true or y must be false. Suppose a second clause is given by $z \vee \bar{x} \vee y$. Then the boolean formula $(x \vee \bar{y}) \wedge (z \vee \bar{x} \vee y)$ is in CNF. An assignment which evaluates this formula to true is given by $(x, y, z) = (1, 0, 1)$ and thus the corresponding SAT problem is satisfiable.

In order to solve the RCPSP subject to no-overlap constraints using a SAT solver, we need to convert each constraint of the problem to CNF clauses. The disjunction of these clauses forms a boolean formula that can be solved by a SAT solver to find a feasible solution to the scheduling problem

This section will propose a SAT encoding of the scheduling problems considered in section 3. This is done by slowly adding constraints to the problem which are then translated into SAT clauses. Section 4.1 will introduce SAT encoding for the trivial scheduling problem considered in section 3.1. Section 4.2 will define SAT clauses needed to implement precedence constraints as introduced in section 3.2. Section 4.3 will convert the resource constraints introduced in section 3.3 into SAT clauses. Finally, section 4.4 will implement no-overlap constraints into our SAT encoding.

4.1 Completion clauses

First we consider a project consisting of only one activity with duration d. A schedule which completes this project within planning horizon T must have the activity start early enough such that it can be finished in time. In section 3.1 we wrote this condition as (2) given by

$$\sum_{t=1}^{T-d+1} y_t = 1. \tag{13}$$

In order to encode this constraint in SAT, we first define a set of boolean variables given by

$$y_t = \begin{cases} 1 \text{ If the activity starts in period } t \\ 0 \text{ If the activity does not start in period } t \end{cases} \tag{14}$$

for $t \in [1, .., T]$. We cannot directly convert the constraint to a SAT clause. Instead, we must split the constraint into $\sum_{t=1}^{T-d+1} y_t \geq 1$ and $\sum_{t=1}^{T-d-1} y_t \leq 1$. The first constraint can easily be encoded in SAT using a CNF clause given by

$$\bigvee_{t=1,..,T-d+1} y_t = 1. \tag{15}$$

This clause will only evaluate to true if the activity starts in period $T - d$ or earlier such that $y_t = 1$ for some $t \in [1, .., T - d + 1]$.

Suppose the activity has a duration of $d = 3$ and the project has a planning horizon given by $T = 5$. This problem has three different feasible schedules, which have the activity start in either period 1, 2 or 3. For this problem, the SAT clause is given by

$$\bigvee_{t=1,..,5-3+1} y_t = 1 \implies y_1 \vee y_2 \vee y_3 = 1. \tag{16}$$

Clearly, the feasible schedules which have the activity start in period 1 ($y_1 = 1$), 2 ($y_2 = 1$) or 3 ($y_3 = 1$) all satisfy this clause. This clause alone however, allows the activity to start multiple times and in periods that make it impossible for the activity to be completed in time. For example, the assignment $(y_1, y_2, y_3, y_4, y_5) = (0, 1, 1, 1, 1)$ satisfies the clause but is no feasible schedule.

In case of such an assignment, a feasible schedule can easily be obtained by selecting the earliest starting period. For example, the assignment $(0, 1, 1, 1, 1)$ will have the activity start in period 2. This way, we do not need clauses to ensure $\sum_{t=1}^{T-d+1} y_t \leq 1$, resulting in a much more compact encoding.

In case a project has an activity set A consisting of multiple activities, we need to extend clause (15). To do this, we introduce a new set of boolean variables defined by

$$y_{it} = \begin{cases} 1 \text{ If activity } i \text{ starts in period } t \\ 0 \text{ If activity } i \text{ does not start in period } t. \end{cases} \tag{17}$$

For each activity i, we must have a clause similar to (15) given by

$$S_i = \bigvee_{t=1,..,T-d_i+1} y_{it} = 1. \tag{18}$$

A feasible assignment must satisfy this clause for each activity, that is, the conjunction of all completion clauses must evaluate to 1. This conjunction is defined as

$$S = \bigwedge_{i=1,..,n} S_i = 1. \tag{19}$$

Note that these clauses also allow for assignments with multiple starting periods for a single activity. A feasible schedule can again be obtained by selecting the first starting period of each activity. This result will hold even after more constraints are added to the SAT encoding. A feasible schedule for a scheduling problem with n activities subject to no constraints that completes all activities within planning horizon T must evaluate the boolean formula $\phi = S$ to 1 or equivalently to true.

4.2 Precedence clauses

Consider a project consisting of a set of activities V as in the previous section but in addition these activities are subject to a set of precedence relations A as introduced in section 3.2.

Suppose activities i and j share a precedence relation (i, j) such that activity i must have finished before activity j can be started. In section 3.2 we wrote this condition as (8) which is equivalent to

$$s_i + d_i \leq s_j \tag{20}$$

where s_i and s_j are the starting periods of activity i and j respectively. For activity j to be started in some period t, activity i must have started in period $t - d_i$ or earlier. Whether prerequisite activity i starts in period $t - d_i$ or earlier and is thus finished before period t can be represented as a boolean variable z_{it} defined as

$$z_{it} = \bigvee_{u=1,..,t-d_i} y_{iu} = \begin{cases} 1 \text{ if activity } i \text{ is finished before period } t, \\ 0 \text{ if activity } i \text{ is not finished before period } t. \end{cases} \tag{21}$$

For activity j to be started in period t, we must have that $z_{it} = 1$. We can write this implication as

$$y_{jt} = 1 \implies z_{it} = 1. \tag{22}$$

Using the rewriting rule $x \implies y \rightarrow \bar{x} \vee y$, this implication can be written as a clause given by

$$P_{ijt} = \bar{y}_{jt} \vee z_{it} = 1 \tag{23}$$

If activity j starts in period t and thus $\bar{y}_{jt} = 0$, activity i must start in period $t - d_i$ or earlier, that is, $z_{it} = 1$. If activity j were not to start in period t, $\bar{y}_{jt} = 1$ and the clause would be satisfied and activity i would not need to be finished before period t.

This precedence clause must be satisfied for each precedence relation and period. Similarly to completion clauses, this conjunction of completion clauses can be defined as

$$P = \bigwedge_{t=1,..,T} \bigwedge_{(i,j)\in A} P_{ijt} = 1. \tag{24}$$

To see how these clauses work in practice, consider an example. Suppose a project consists of two activities with durations given by $d_1 = 2$ and $d_2 = 1$ and the planning horizon is given by $T = 4$. Furthermore, these activities share a precedence constraint $(1, 2)$ such that activity 1 must be finished before activity 2 can be started. We can write out the precedence clauses corresponding to this project for each value of t:

$$t = 1: \quad z_{1,1} = 0 \rightarrow P_{1,2,1} = \bar{y}_{2,1} \vee z_{1,1} = 1 \implies \bar{y}_{2,1} = 1, \tag{25a}$$

$$t = 2: \quad z_{1,2} = 0 \rightarrow P_{1,2,2} = \bar{y}_{2,2} \vee z_{1,2} = 1 \implies \bar{y}_{2,2} = 1, \tag{25b}$$

$$t = 3: \quad z_{1,3} = y_{1,1} \rightarrow P_{1,2,3} = \bar{y}_{2,3} \vee y_{1,1} = 1, \tag{25c}$$

$$t = 4: \quad z_{1,4} = y_{1,1} \vee y_{1,2} \rightarrow P_{1,2,4} = \bar{y}_{2,4} \vee y_{1,1} \vee y_{1,2} = 1. \tag{25d}$$

For an assignment to satisfy the precedence constraints, all clauses (25) need to evaluate to true. Evidently, clauses (25a) and (25b) imply that activity 2 cannot start in periods 1 and 2. In case activity 2 starts in period 3 such that $y_{2,3} = 1$ and thus $\bar{y}_{2,3} = 0$, clause (25c) forces activity 1 to start in period 1 such that $y_{1,1} = 1$. Similarly, if activity 2 starts in period 4, clause (25d) forces activity 1 to start in either period 1 or 2. These clauses thus only allow the three feasible schedules for this project given by $(s_1, s_2) = (1, 3)$, $(s_1, s_2) = (1, 4)$ and $(s_1, s_2) = (2, 4)$ where s_1 and s_2 are starting periods of activity 1 and 2 respectively.

For project scheduling problems subject to precedence constraints, the critical path algorithm proposed by Floyd (1962) can be used to determine early and late starting times for each activity. Activities cannot start earlier or start later than their respective early and late starting times. We denote the early and late start times of activity i by e_i and l_i. Given the early and late starting time of each activity, we can rewrite a significant portion of precedence and completion clauses to a more compact form.

Consider completion clauses (19), as each activity cannot start before period e_i, we have $y_{it} = 0 \; \forall t \in [1, .., e_i - 1]$. Similarly, given late start time l_i we have $y_{it} = 0 \; \forall t \in [l_i + 1, .., T]$. We can rewrite each completion clause (18) to

$$S_i = \bigwedge_{t=s_i,..,l_i} y_{it} = 1 \tag{26}$$

such that all completion clauses can be defined as

$$S = \bigwedge_{i=1,..,n} S_i = \bigwedge_{i=1,..,n} \bigvee_{t=e_i,..,l_i} y_{it} = 1. \tag{27}$$

Similarly, we can rewrite precedence clauses (24). As for each successor activity j, $y_{jt} = 0$ and thus $\bar{y}_{jt} = 1$ for all values of $t < s_j$ and $t > l_j$, precedence clauses are trivially satisfied for these values of t. As a result, the conjunction of precedence clauses P can be simplified to

$$P = \bigwedge_{t=e_j,..,l_j} \bigwedge_{(i,j)\in A} P_{ijt} = 1 \tag{28}$$

As for each prerequisite activity i, $y_{it} = 0$ for $t < e_i$, we can redefine $z_{i,t}$ as

$$z_{it} = \bigvee_{u=e_i,..,t-d_i} y_{iu} \tag{29}$$

A schedule which completes a project with limitless resources subject to precedence constraints and planning horizon T, evaluates the boolean formula $\phi = S \wedge P$ to 1.

4.3 Resource clauses

In addition to a set of precedence constraints, now consider a project that is also subject to resource constraints. In section 3.3 we defined resource constraints as (11) defined as

$$\sum_{i=1}^{n} a_{it} r_{ij} \leq R_j \quad \forall j \in [1, .., J], \; \forall t \in [1, .., T] \tag{30}$$

where $a_{ij} = 1$ if activity i is active in period t and 0 otherwise.

In order to convert these resource constraints to CNF clauses, a new set of activity variables is introduced. These variables are defined as

$$x_{it} = \begin{cases} 1 \text{ If activity } i \text{ is active in period } t \\ 0 \text{ If activity } i \text{ is inactive in period } t \end{cases} \tag{31}$$

for $t \in [1, .., T]$. An activity i that starts in some period t must be active in period t and the following $d_i - 1$ periods.

Suppose an activity i has a duration of 3 and starts in some period t. Activity i must then be active in period t, $t + 1$ and $t + 2$. This implication can be written as

$$y_{it} = 1 \implies x_{it} \wedge x_{i,t+1} \wedge x_{i,t+2} = 1 \tag{32}$$

which we can write as CNF clauses given by

$$(\bar{y}_{it} \vee x_{it}) \wedge (\bar{y}_{it} \vee x_{i,t+1}) \wedge (\bar{y}_{it} \vee x_{i,t+2}) = 1. \tag{33}$$

If activity i starts in period t, $\bar{y}_{it} = 0$ and thus x_{it}, $x_{i,t+1}$ and $x_{i,t+2}$ must all equal 1 to satisfy all clauses. If activity i were not to start in period t, $\bar{y}_{it} = 1$ and thus the clauses would be satisfied and activity i would not be required to be active in periods t, $t + 1$ and $t + 2$.

We can generalize these clauses for different periods and activities with different durations. For each activity i, we must have that

For each activity i we can generalize these clauses referred to as consistency clauses defined by

$$C_i = \bigwedge_{t=e_i,..,l_i} \bigwedge_{u=t,..,t+d_i-1} \bar{y}_{it} \vee x_{iu} = 1. \tag{34}$$

Of course this clause needs to evaluate to true for all activities. Thus, all consistency clauses can be defined as

$$C = \bigwedge_{i=1,..,n} C_i = 1. \tag{35}$$

Whether an activity i is active in period t and is thus using resources, is represented by x_{it}. The amount of resources of resource group j activity i uses in period t is given by

$$x_{it} r_{ij}. \tag{36}$$

The total amount of resources of resource group j used in period t by all activities combined is given by

$$\sum_{i=1}^{n} x_{it} r_{ij}. \tag{37}$$

As resources used cannot exceed resource capacity, resource constraints can be redefined as

$$\sum_{i=1}^{n} x_{i,t} r_{i,j} \leq R_j \qquad \forall t \in [1,..,T],\ \forall j \in [1,..,J]. \tag{38}$$

In literature, there are generally two ways these resource constraints can be implemented in SAT. The first approach is to also convert these constraints to CNF clauses, as is done with other constraints. This way, the boolean formula maintains its CNF property. The second approach is to not convert these constraints to CNF clauses, but rather let the SAT solver deal with them directly in the form of pseudo-boolean clauses. Converting resource constraints to CNF clauses allows the SAT problem to be solved by an unmodified SAT solver. The second approach requires a modified solver with so-called 'native support'.

In order to encode resource constraints into CNF clauses, Horbach (2010) proposes the use of minimal cover clauses. A big disadvantage of this method is the substantial overhead due to exponentially increasing number of minimal cover clauses. For large instances, the excessive number of clauses will result in memory overflow. Horbach (2010) recognizes this problem and proposes a method of circumventing this problem by changing the way the solver looks up clauses. This method however, requires modification of the solver and thus decreases the ease of implementation.

Instead of converting the constraints to CNF clauses, we propose to let the SAT solver deal with the resource constraints directly in the form of pseudo-boolean clauses. For each resource group j, these clauses are defined as

$$B_j = \bigwedge_{t=1,..,T} \sum_{i=1}^{n} x_{i,t} r_{i,j} \leq R_j = 1. \tag{39}$$

The conjunction of all pseudo-boolean resource clauses can then be defined as

$$B = \bigwedge_{j=1,..,J} B_j = 1. \tag{40}$$

A feasible schedule for a project subject to precedence and resource constraints which completes all tasks within planning horizon T evaluates the boolean formula $\phi = S \wedge P \wedge C \wedge B$ to 1

Of course, this requires the SAT solver to be modified such that it not only able to deal with clauses consisting of just literals. There exist a number of SAT solvers, such as Z3, which have this support and can very efficiently deal with such constraints.

Directly dealing with resource constraints in the form of pseudo-boolean clauses has a number of advantages. Firstly, this method circumvents the memory overflow problem due to excessive numbers of CNF clauses. Thus, no further modification of the solver is required. Secondly, Aloul (2002) has shown that directly dealing with resource constraints has a significant computational advantage when solving the SAT problem. In addition, clause generation is significantly faster as generating minimal cover clauses takes a significant amount of computations, especially for large projects.

Including these constraints directly into our encoding in the form of pseudo-boolean clauses takes away the CNF property of our SAT problem. The boolean formula now consists of not only clauses consisting of literals but also of pseudo-boolean clauses. How the Z3 solver deals with this difference and pseudo-boolean clauses in general will be explained in section 5.

4.4 No-overlap clauses

Now consider a scheduling problem subject to precedence, resource and in addition a set of no-overlap constraints O. In section 3.4 we defined no-overlap constraints as

$$a_{it} + a_{jt} \leq 1 \quad \forall (i,j) \in O, \; \forall t \in [1, 2.., T]. \tag{41}$$

where a_it represents whether activity i is active in period t.

If some activity i and j share a no-overlap constraints, they cannot be active in the same period. Using the fact that $a_{it} = x_{it}$, this condition can be expressed as

$$x_{it} \wedge x_{jt} = 0. \tag{42}$$

We can rewrite this condition to a CNF clause given by

$$\begin{aligned} x_{it} \wedge x_{jt} = 0 &\iff \overline{x_{it} \wedge x_{jt}} = 1 \\ &\iff \bar{x}_{it} \vee \bar{x}_{jt} = 1. \end{aligned} \tag{43}$$

This clause can be generalized for all no-overlap constraints and periods. These clauses are referred to as no-overlap clauses defined by

This clauses can be generalized for all periods. Activities i and j which share a no-overlap constraint (i,j) must satisfy the no-overlap clause N_i defined by

$$N_i = \bigwedge_{t=1,..T} \bar{x}_{it} \vee \bar{x}_{jt}. \tag{44}$$

As no-overlap clauses must be satisfied for all no-overlap constraints, the conjunction of all no-overlap clauses can be defined as

$$N = \bigwedge_{(i,j) \in O} N_i = 1 \tag{45}$$

The SAT encoding of no-overlap constraints is significantly more compact than its MIP counterpart, highlighting one of the advantages of solving such scheduling problem using SAT encoding. This compactness will directly improve the efficiency with which the SAT solver deals with no-overlap constraints. A feasible schedule for a project subject to precedence and resource constraints which completes all tasks within planning horizon T evaluates the boolean formula $\phi = S \wedge P \wedge C \wedge B \wedge N$ to 1.

5 Z3 solver

We run numerical analysis of our SAT encoding using the Microsoft Z3 solver introduced by De Moura and Bjørner (2010). While some of Z3's solver engines date back quite some time, most have been replaced or improved with the latest developments in solver theory. As a result, the Z3 solver is state of the art and competitive with modern SAT solvers. Furthermore, unlike most modern SAT solvers, Z3 has pseudo-boolean support which is required for our proposed encoding and a user-friendly interface.

The Z3 SAT solver is based on the Conflict Driven Clause Learning (CDCL) algorithm introduced by Marques-Silva & Sakallah (1999) and Bayardo & Shrag (1996). The CDCL-algorithm is largely inspired by the Davis–Putnam–Logemann–Loveland (DPLL) introduced by Davis et al. (1960) and Davis et al. (1962).

The DPLL algorithm is a branching algorithm which splits the SAT problem into smaller SAT problems. At each split a variable is assigned one of two ways, one branch assigns the variable true and branches further given this assignment, while the other branches further given the variable is assigned false. This branching continuous till either a solution has been found or all branches have been attempted. The algorithm is presented below in pseudo code below.

Algorithm 1 DPLL algorithm

Require: Boolean formula F consisting of clauses

```
 1: function DPLL(F)
 2:     if F has no clauses then
 3:         return True
 4:     if F has empty clauses then
 5:         return False
 6:     for every unit clause l do
 7:         F ← unit_propagate(F,l)
 8:     for all every pure literal do
 9:         F ← assign_pure_literal(F,l)
10:     l ← choose-literal(F)
11:     return DPLL(F∧$l$) or DPLL(F∧$\bar{l}$)
```

The DPLL function is a recursive function which takes a boolean formula F in CNF as its argument. Each recursion a variable x is picked based on some picking rule defined by the choose-variable function and two branches representing subproblems are created. One branch assigns $x = 1$ and thus $\bar{x} = 0$, while the second branch assigns $x = 0$ and thus $\bar{x} = 1$. After any assignment, we can reduce the boolean formula F. For the branch with corresponding assignment $x = 1$ this reduction happens in two ways:

1. **Removing clauses**, clauses which contain the assigned literal x are now satisfied and can thus be removed from the boolean formula corresponding to the branch, and

2. **Removing literals**, the negation of the assigned literal, in this case $\bar{x}$, is removed from all clauses in the boolean formula corresponding to the branch. For example, a clause $\bar{x} \vee y \vee z$ is reduced to $y \vee z$.

After each split, the boolean formula is reduced as some clauses and literals are removed.

This process of picking a variable, creating two branches and updating the corresponding boolean formula F continuous till either

- the boolean formula F is reduced such that is contains no more clauses. In this case all clauses are satisfied and thus a feasible solution has been found, or

- the boolean formula F contains a clauses which no longer contains any literals. Such an empty clause cannot be satisfied under the current assignment of variables. The current assignment of variables can thus not lead to a feasible solution. As a result, the branch comes to a dead end and returns false.

The algorithm ends if either one branch finds a feasible solution and returns true or all branches come to a dead end and return false.

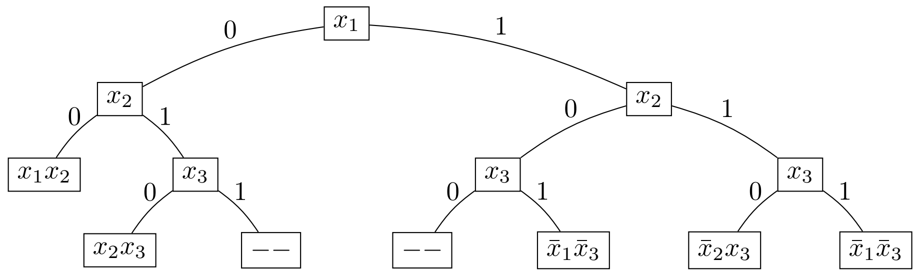


Figure 1: Branching tree representation of the DPLL algorithm for the boolean formula $F = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3)$

To see how this branching process works, consider a simple SAT problem with corresponding boolean formula defined by $F = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3)$. Figure 1 presents the branching tree resulting from the branching process. The reasoning of branching and updating the boolean formula works as follows, here removing (x_1) refers to removing the clause (x_1) from the boolean formula while removing x_1 refers to removing the literal x_1 from each clause.

$$\begin{aligned}
&x_1 \leftarrow 0 \implies \text{remove } (\bar{x}_1 \vee \bar{x}_3) \text{ and } x_1 \implies F = (x_2) \wedge (\bar{x}_2 \vee x_3)\\
&\quad x_2 \leftarrow 0 \implies \text{remove } x_2 \implies \text{clause } (x_1 \vee x_2) \text{ is empty}\\
&\quad x_2 \leftarrow 1 \implies \text{remove } (x_2) \text{ and } \bar{x}_2 \implies F = (x_3)\\
&\qquad x_3 \leftarrow 0 \implies \text{remove } x_3 \implies \text{clause } (\bar{x}_2, x_3) \text{ is empty}\\
&\qquad x_3 \leftarrow 1 \implies \text{remove } (x_3) \implies F = \emptyset, \text{ solution found}\\
&x_1 \leftarrow 1 \implies \text{remove } (x_1 \vee x_2) \text{ and } \bar{x}_1 \implies F = \bar{x}_3 \wedge (\bar{x}_2 \vee x_3)\\
&\quad x_2 \leftarrow 0 \implies \text{remove } (\bar{x}_2 \vee x_3) \text{ and } x_2 \implies F = (\bar{x}_3)\\
&\qquad x_3 \leftarrow 0 \implies \text{remove } (\bar{x}_3) \implies F = \emptyset, \text{ solution found}\\
&\qquad x_3 \leftarrow 1 \implies \text{remove } \bar{x}_3 \implies \text{clause } (\bar{x}_1 \vee \bar{x}_3) \text{ is unsatisfiable}\\
&\quad x_2 \leftarrow 1 \implies \text{remove } \bar{x}_2 \implies F = \bar{x}_3 \wedge x_3\\
&\qquad x_3 \leftarrow 0 \implies \text{remove } (\bar{x}_3) \text{ and } x_3 \implies \text{clause } (\bar{x}_2 \vee x_3) \text{ is unsatisfiable}\\
&\qquad x_3 \leftarrow 1 \implies \text{remove } (x_3) \text{ and } \bar{x}_3 \implies \text{clause } (\bar{x}_1 \vee \bar{x}_3) \text{ is unsatisfiable.}
\end{aligned}$$

To improve efficiency, the DPLL algorithm proposes two new concepts:

1. **unit clauses**, which are clauses which contain only one literal. These clause can thus only be satisfied by assigning this literal to true, and
2. **pure literals**, which are literals of which the negation is not present in the boolean formula. For example, the boolean formula only contains the positive literal x and does not contain the negative literal $\bar{x}$. For example, the positive literal x is a pure literal if the boolean formula F does not contain the negative literal $\bar{x}$. Since the boolean formula is in CNF, assigning pure literals to true is always a good choice and cannot cause the SAT problem to become unsatisfiable.

Every time a new variable is picked, two branches are created and the boolean formula is updated accordingly, the DPLL algorithm checks whether the updated boolean formula F contains a unit clause. For each unit clause, a unit propagation function assigns the appropriate literal to true such that the clause is satisfied and updates the boolean formula given this assignment. For each pure literal, a pure literal function assigns the literal to true and updates the boolean formula accordingly.

Including unit propagation and pure literal assignment makes branches converge to feasible solution or dead ends more quickly. As a result, this improve the efficiency of the algorithm. In the introduced example, after assigning $x_1 = 0$, the boolean formula is given by $F = x_2 \vee (\bar{x}_2 \vee x_3)$ which clearly contains a unit clause x_2. Unit propagation will immediately recognize this and assign x_2 to true. This way, the algorithm does not branch on an assignment $x_2 = 0$ which reduces the amount of computations.

The basic way this algorithm can deal with pseudo-boolean clauses can be summarised as follows. The resource constraints are represented in pseudo-boolean clauses of the form $\sum_{i=1}^{n} a_i x_i \leq k$ where x_i is a boolean variable and a_i and k_i are scalars. For each pseudo-boolean clause, the algorithm keeps track of the value of the sum of already assigned variables. If a variable is assigned true, its coefficient is added to the sum. If, after an assignment, the sum exceed the right-hand-side value, the algorithm backtrack much in the same way if it were to encounter an empty CNF clause.

In Z3, a solvers reasoning can be orchestrated with a methodology called tactics, using combinators called tacticals. Tacticals allow tactics to be combined in, for example, sequences using the *Then* tactic. Similarly, tactics can be applied in loops and and/or logic. Z3 provides a large variety of tactics (up to 100) which, if applied to the correct problem, can significantly improve performance. However, a downside of this feature is that documentation with regards to tactics is lackluster and thus finding optimal tactics for the current satisfiability problem relies heavily on trial and error.

If no tactics are set for a Z3 solver, Z3 resorts to a standard solver based on SAT model characteristics. This solver performs reasonably well for most SAT problems, however it is not optimized for specific SAT problems such as project scheduling problems subject to no-overlap constraints. Some solver efficiency can thus be gained by setting specific solver tactics.

As resource clauses are directly implemented in Z3 using pseudo-boolean encoding, a specific pseudo-boolean solver needs to be invoked. This can be done using the tactic 'pb-solver'. Performance can be improved by using the tactic 'solve-eqs' and 'simplify' which apply Gaussian elimination and simplify each sub-problem after each assignment. The increased performance due to these parameter settings is significant, especially for larger instances.

6 Results

In order to demonstrate the efficiency of our set encoding presented in section 4, we apply the SAT model to a set of instances in different settings using the python Z3 solver introduced by De Moura and Bjørner (2010). All instances are solved using an Intel I7 7700K 4.2 ghz processor. RCPSP instances used are from the PSPLIB, called PSD instances, and generated by Progen as proposed in Kolisch et al. (1995). This library contains sets of 30, 60, 90 and 120 activities, each consisting of 480 easier and harder problems.

Section 6.1 discusses computational SAT encoding results for the most trivial scheduling problem. Section 6.2 considers the same problem but adds an additional set of precedence constraints. Section 6.3 introduces resource constraints and section 6.4 discusses computational results taking into account no-overlap constraints.

6.1 Trivial case

First consider the trivial scheduling problem in which precedence, resource and no-overlap constraints are ignored. Thus, only the activity durations data of the PSD instances are used. As discussed in section 3.1, a feasible schedule needs to just satisfy completion clauses (27). Evidently, a feasible schedule can be obtained instantly by starting each activity immediately.

For this trivial case, the early and late start time of activity i are given by $e_i = 0$ and $l_i = T - d_i$. Consequently, the completion clauses (19) and (27) are equal. The number of clauses only increases with the number of activities n and is independent from the given planning horizon T. This would suggest that computing times do not increase with planning horizon T. However, T does affect the size of each clause which negatively impacts computing times.

There exists however a trade-off when increasing the planning horizon T. An increased planning horizon increases the size of clauses but also enlarges the solution space. On one hand a larger solution space implies more feasible schedules exist and thus it would be easier for the solver to find one. Naturally, this would result in a decrease in computing times. On the other hand, an increased planning horizon would increase the number of variables and thus enlarge the search space. To see how this trade-off balances out, we solve the SAT problem for different project sizes and planning horizons. The resulting computing times are presented in table 1.

k	1	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9
J30	0.022	0.024	0.027	0.028	0.030	0.031	0.033	0.035	0.036	0.038
J60	0.037	0.043	0.046	0.050	0.052	0.056	0.060	0.064	0.068	0.071
J90	0.054	0.060	0.064	0.070	0.076	0.079	0.080	0.090	0.094	0.100
J120	0.074	0.078	0.081	0.090	0.096	0.103	0.110	0.120	0.122	0.131

Table 1: Computing times (sec) of PSD instances only subject to completion clauses for different values of k with $T = kd_{max}$.

The scheduling problem with planning horizon T given by $T = kd_{max}$, where d_{max} is the longest activity duration, is solved for different values of k. Most notably, computing times increase with planning horizon T. Thus, for this scheduling problem the effect of a larger solution space is completely offset by the increased size of clauses. Note that the increase in computing times due to an increased planning horizon is on average higher for larger projects. Similarly, the increased computing times due to project size increase with planning horizon T. The number of clauses and the size of clauses thus share an interaction effect beside their individual effect on computing times. This effect becomes more apparent when more constraints are added to the scheduling problem in later sections.

One advantage of our SAT encoding is the compact way the completion constraints (4) are encoded in SAT. The alternative method of enforcing exactly one starting period for each activity by introducing more clauses negatively impacts compactness. Though our SAT encoding requires a second step to obtain a feasible schedule, it yield significantly faster computing times compared to this alternative encoding. Computing times are improved by about 5% for this trivial problem and significantly more for projects subject to more constraints.

6.2 Precedence constraints

Now we consider the same project scheduling problem, but with an additional set of precedence constraints. The PSD instances contain a set of precedence relations in which each activity has a maximum of 3 successors.

The number of precedence clauses (28) increases with planning horizon T and the number precedence relations. Again, a trade-off exists between an increased number of clauses and a larger solution space when increasing the planning horizon T. An optimal makespan for this project can be found using the critical path algorithm and is denoted by t_{cpm}. Table 2 presents computing times with planning horizons T defined by $T = kt_{cpm}$ for different values of k.

Figure 2 shows a representation of an example solution for a project subject to precedence constraints consisting of 60 tasks and planning horizon 80. As no resource constraints are present, there exists no limit to the number of activities that can be active in parallel. The figure shows a characteristic shared by most PSD instances, the number of successor activities is largest for activities with few prerequisite activities. As a result, the number of activities planned in parallel is largest at earlier periods and decreasing for later ones.

As with the trivial case, computing times are strictly increasing with planning horizon T for all different project sizes. The negative effect of an increased number of clauses and a larger search space thus offset the effect of a larger solution space. Furthermore, using the early and late start times of activities to convert precedence clauses (24) to (28) significantly reduces the

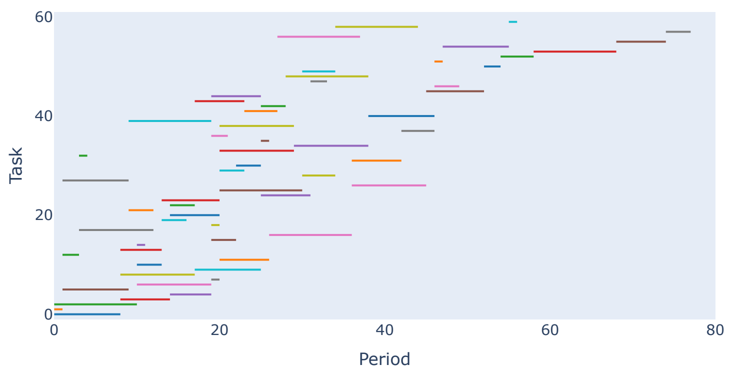


Figure 2: Example schedule for a project consisting of 60 tasks only subject to precedence constraints given a planning horizon 80.

k	1	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9
J30	0.086	0.097	0.109	0.115	0.120	0.133	0.142	0.148	0.155	0.176
J60	0.223	0.242	0.264	0.303	0.317	0.341	0.365	0.390	0.411	0.434
J90	0.400	0.446	0.478	0.519	0.563	0.609	0.644	0.688	0.734	0.780
J120	0.561	0.617	0.662	0.733	0.782	0.849	0.927	0.972	1.020	1.082

Table 2: Computing times (sec) of PSD instances only subject to precedence constraints for different values of k with $T = kt_{cpm}$.

number of clauses. As a result, computing times are on average 5% and 10% lower for small and large instances respectively.

6.3 Resource constraints

Now consider the same project as in the previous section, but in addition subject to a resource constraints. PSD instances all have 4 resource groups with varying capacities. Each activity requires either one resource group for easier instances or multiple for harder ones.

The introduction of resource constraints significantly complicates the scheduling problem and could be seen as a tipping point. For projects without resource constraints, a feasible and optimal schedule can easily be obtained using the critical path algorithm even for extremely large projects consisting of thousands of activities. For project subject to resource constraints however, optimal schedules can generally only be computed for projects consisting of less than 60 activities. In the presence of resource constraints a reasonable planning horizon T can thus only be estimated. A simple yet effective estimate takes the form $T = kt_{cpm}$ where t_{cpm} is the optimal makespan of the project ignoring resource constraints and k some scalar.

In order to emphasize the relation between computing times and the tightness for the planning horizon, we again compute schedules or varying planning horizons. However, using planning horizon $T = kc_{cpm}$ to emphasize this relation is troublesome as we do not know up front for each instance for what values of k the planning horizon is tight. Especially as computing times are reported as averages over all instances, which will be distorted as for each instance a planning horizon is considered tight for a different parameter value of k.

Instead of an estimate based on the critical path, we consider an estimate based on heuristics such that the planning horizon is given by $T = kt_h$ where t_h is the heuristic makespan. This way, a relatively low value of k such as 1.1 will result in a tight planning horizon for all instances and thus average computing times are not distorted. Heuristic makespans for each instance are provided by the PSPLIB. For instances that are closed, this makespan is equal to the optimal makespan. Heuristic solution are mostly provided by algorithms introduced by Elsayed (1982) , Kolisch and

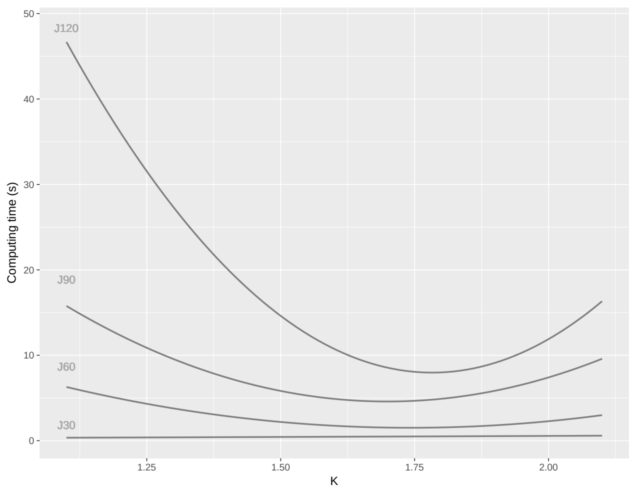


Figure 3: Computing times for instances subject to resource and precedence constraints for different values of k with planning horizon $T = kt_h$.

Drexl (1996) , Baar et al. (1999) , Bouleimen and Leqocq (1998) and Nonobe and Ibaraki (2006). Of course, in practice such a heuristic schedule is not available, but for this instance it enables us to highlight SAT solver behaviour.

Figure 4 shows an example of a representation of a feasible schedule for a project with 30 activities subject to resource constraints. Figure 3 presents computing times for different planning horizons based on heuristic makespans. Most notably, computing times are no longer strictly increasing in planning horizon T. The trade-off between a larger solution space, a larger search space and an increased number of clauses balances out quite differently due to the increased complexity of the scheduling problem. The negative effect of a small solution space seems to offset the effect of a smaller search space for small values of k, while for relatively higher values of k this effect is reversed and computing times increase.

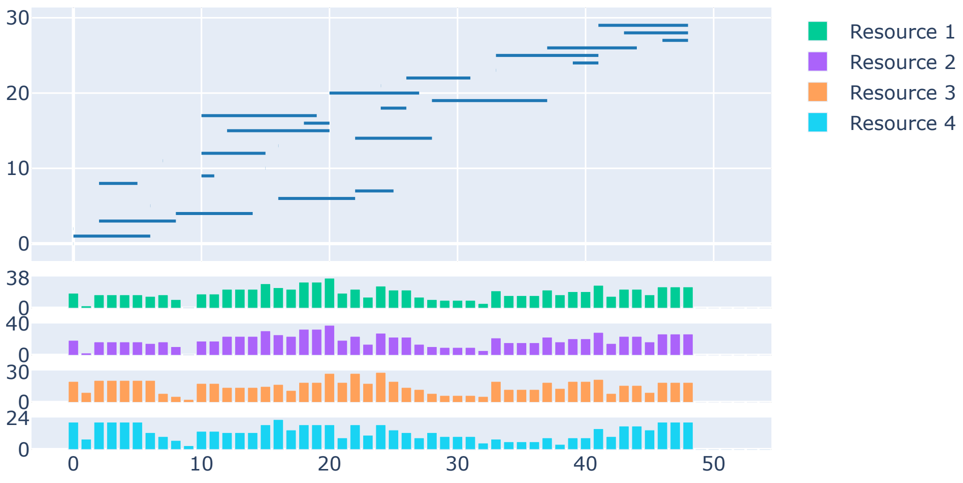


Figure 4: Representation of a feasible schedule for a project with 30 activities subject to precedence and resource constraints.

Note that this balancing act differs significantly across different project sizes. For the smallest instance, computing times are still increasing in planning horizon T, while for larger instances the effect of a larger solution space is much more significant. Furthermore, larger instances require higher over-estimations of the makespan in order to obtain optimal computing times. This must be interpreted with caution however as extreme over-estimations will cause the computing time to explode, especially for larger instances.

6.4 No-overlap constraints

In addition to resource constraints we add a set no-overlap constraints. PSD instances do not include no-overlap constraints and thus these must be generated. Pairs of activities which share a no-overlap constraint are randomly generated. The no-overlap density o is defined as $o = \frac{m}{n}$, that is, the number of no-overlap constraints m relative to the number of activities n in a project.

We can compare the performance of our SAT encoding in Z3 to a MIP solver such as Gurobi. In this MIP, a feasible schedule for a project scheduling problem subject to precedence, resource and no-overlap constraints must satisfy completion constraints (4), precedence constraints (8), resource constraints (11) and no-overlap constraints (12).

Table 3 presents computing times of our SAT encoding and Gurobi for differently sized scheduling problems under different parameter settings. Firstly, by varying planning horizon T defined as $T = kt_{cpm}$ for different values of k and secondly, by setting different no-overlap densities o. Evidently, adding additional constraints in the form of no-overlap constraints increases computing times. Note that for larger instances, the addition of no-overlap constraints affects the performance of Gurobi significantly more than our SAT encoding. This highlights an advantage of using SAT encoding in the presence of many no-overlap constraints.

Our results show that for smaller instances subject to few no-overlaps and tight planning horizons Gurobi performs significantly better than our proposed SAT encoding. However, as instance become larger and more no-overlap constraints are added, our SAT encoding gives significantly better performance for reasonable estimates of T.

The parabolic relation between computing times and planning horizon T as presented in figure 3 is also present for projects subject to no-overlap constraints. Both Gurobi and our SAT encoding are affected by how wide planning horizon T is set, though this effect is more significant for the SAT approach. This effect is shown in figure 5 which presents computing times as a function of planning horizon T for differently sized instances subject to no-overlap constraints. Clearly our SAT encoding can be competitive if planning horizons are not too tight or wide, especially for larger instances.

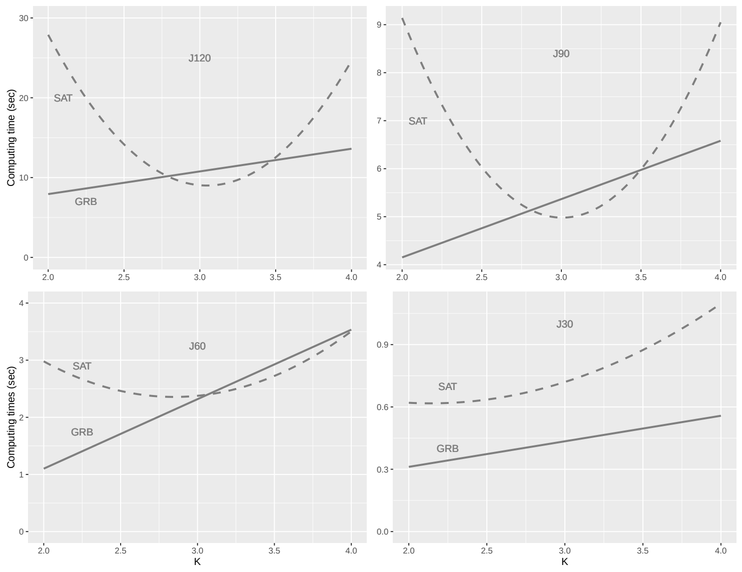


Figure 5: Computing times of SAT and Gurobi (GRB) for differently sized problems subject to no-overlap density $o = 2$.

Even though our SAT encoding is efficient at dealing with no-overlap constraints, competitiveness is mainly determined by our choice of planning horizon T. Only if the planning horizon is neither too wide nor too tight, our SAT encoding is competitive for larger instances subject to no-overlap constraints. For project consisting of 90 activities and a no-overlap density of 2, our SAT encoding and Gurobi solve all instances in 4.98 and 5.51 seconds respectively. For projects consisting of 120 activities and a no-overlap density of 2, our SAT encoding and Gurobi solve all instances in 9.04 and 11.62 seconds respectively. This suggests that the relative performance advantage over Gurobi increases with project size and thus we expect our SAT encoding to be more competitive for even larger instances subject to no-overlap constraints

		SAT			Gurobi		
	o	0	1	2	0	1	2
	k						
	2	0.57	0.61	0.62	0.13	0.21	0.31
J30	3	0.63	0.72	0.85	0.18	0.32	0.45
	4	1.09	1.11	1.20	0.36	0.41	0.55
	2	2.72	2.80	2.98	0.511	0.78	1.07
J60	3	2.00	2.20	2.38	0.70	1.15	2.38
	4	4.87	4.90	5.01	1.48	1.83	3.50
	2	7.89	8.12	9.14	2.10	3.02	4.08
J90	3	3.80	4.51	4.98	3.63	4.23	5.51
	4	11.79	12.05	14.05	4.74	5.78	6.51
	2	24.22	25.60	27.89	2.91	5.79	7.51
J120	3	7.85	8.56	9.04	6.16	9.31	11.62
	4	19.51	22.14	24.64	7.83	12.37	13.19

Table 3: Average computing times (sec) of PSD instances with different planning horizon $T = kt_{cpm}$ and no-overlap density o.

7 Concluding remarks

This thesis proposed a SAT encoding for finding feasible schedules to resource-constrained project scheduling problem subject to no-overlap constraints given a planning horizon. We report on the performance of this encoding given a number of parameter settings and find that it performs particularly well for relatively large planning horizons. Furthermore, we find that that the current encoding is efficient in dealing with larger sets of no-overlap constraints.

The results show that a SAT solver approach to a RCPSP subject to no-overlap can be preferred over finding heuristic solutions using a mixed integer linear program such as Gurobi. This is especially true for larger instances subject to many no-overlap constraints. This advantage only holds if a reasonable estimate of the makespan can be made.

The proposed SAT approach seeks to find a feasible schedule given a planning horizon whereas most heuristic solutions seek to find the best possible schedule within a given computing time or iteration limit. As the performance of our SAT encoding is very poor for very tight planning horizons, using existing heuristic solutions will still be preferable if the goal is to find a very high quality solutions. This quality and performance comes at a cost of ease of implementation and adaptability compared to a SAT or MIP approach. In case new types of constraints or problems are introduced, most advanced heuristics, such as genetic based ones, require extensive changes. This is however not necessary in case SAT encoding is used as most alterations of the considered scheduling problem only require an additional set of clauses without a need to overhaul the entire structure of the encoding.

The SAT proposed SAT approach could possibly be improved on a number of points. As mentioned in section 4.3, resource clauses are of a pseudo-boolean nature. In our SAT encoding resource constraints are converted to CNF clauses using cover clauses. An alternative and well known way of encoding a pseudo-boolean constraint is by representing the constraint using a Binary Decision Diagram (MDD) and encoding this diagram in SAT form. This approach has been used by various papers such as Abio et al. (2012).

Using MDD diagrams to code pseudo-boolean constraints in the context of resource constrained scheduling is not a new idea. However, the addition of no-overlap constraints allows us to encode MDD diagrams more efficiently. No-overlap constraints can be defined as At-Most-One (AMO) constraints as at most one activity can be active at any given period. Only recently a few papers have proposed combining constraints of this type with pseudo-boolean constraints into efficient encoding. This allows us to combine pseudo-boolean resource constraints with AMO no-overlap constraints efficiently and thus improve performance. A few papers have proposed methods to combine pseudo-boolean and AMO constraints in general such as Bofill (2020) and Bofill (2017) in the context of project scheduling, however no such approach has been attempted in the context of project scheduling in the presence of no-overlap constraints.

The proposed SAT encoding can serve as a good foundation for finding solutions to extensions of the considered scheduling problem using SAT solvers. Most extensions can easily be incorporated into the encoding by adding an additional set of clauses. Specifically extension and variations which do not lend themselves to be easily implemented in a MIP could benefit significantly from a SAT approach.

An example of such a scheduling problem would be a scheduling problem subject to or precedence (OR) constraints first described by Elmaghraby (1964). Constraints of this type can be explained as follows. Normal precedence relations requires all prerequisite activities for some successor activity to be completed. OR constraints require only one prerequisite activity to be completed before the successor activity can be started.

Such constraints can easily be implemented in SAT encoding by modifying the proposed precedence clauses. Instead of precedence clauses P defined in (28) defined as

$$P = \bigwedge_{t=1,..,T} \bigwedge_{(i,j)\in A} P_{ijt} = 1, \tag{46}$$

OR precedence clauses can be defined as

$$P = \bigwedge_{t=1,..,T} \bigvee_{(i,j)\in A} P_{ijt} = 1. \tag{47}$$

Note that this encoding results in fewer but larger clauses compared to the generic precedence relations. However, the overall compactness of the encoding remains as the number of literals does not change.

On the contrary, implementing OR constraints in a MIP requires significant changes to the program used for generic precedence constraints. As OR constraints cannot be implemented as linear constraints directly, an additional set of constraints and slack variables is required. As a result, the compactness is significantly reduced which can negatively affects performance.

Encoding scheduling problems in SAT opposed to MIP can be beneficial for many more variations and extensions of the RCPSP. For example, other variants of logical activity dependencies such as exclusive OR constraints, but also variations with respect to resource and activity concepts. Furthermore, SAT solvers technology has been subject to striking progress, especially in recent years. According to Alouneh et al. (2018) this trend is likely to continue. As a result, SAT solvers are likely to gain more prominence in the field of resource constrained scheduling in the near future.

References

Abío, I., R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger (2014), "A new look at bdds for pseudo-boolean constraints", *Journal of Artificial Intelligence Research*, **45**, 443–480.

Aloul, F.A., A. Ramani, I.L. Markov, and K.A. Sakallah (2002), "Generic ilp versus specialized 0-1 ilp: an update", *International Conference on Computer Aided Design*, 450–457.

Alouneh, S., S. Abed, M. H. Al Shayeji, and R. Mesleh (2018), "A comprehensive study and analysis on sat-solvers: advances, usages and achievements", *Artificial Intelligence Review*, **52**(4), 2575–2601.

Baar, T, P Brucker, and S. Knust (1999), "Tabu search algorithms and lower bounds for the resource-constrained project scheduling problem", *Meta-Heuristics*, 1–18.

Bayardo, R. and R. Schrag (1996), "Using csp look-back techniques to solve exceptionally hard sat instances", *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, 46–60.

Blazewicz, J., J.K. Lenstra, and A.H.G. Rinnooy Kan (1969), "Scheduling subject to resource constraints: Classification and complexity", *Discrete Applied Mathematics*, **5**(1), 11–24.

Bofill, M., J. Coll, J. Suy, and M. Villaret (2017), "Compact mdds for pseudo-boolean constraints with at-most-one relations in resource-constrained scheduling problems", *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, 555–562.

Bofill, M., J. Coll, J. Suy, and M. Villaret (2020), "An mdd-based sat encoding for pseudo-boolean constraints with at-most-one relations", *Artificial Intelligence Review : An International Science and Engineering Journal*, **53**(7), 5157–5188.

Bouleimen1998, K. and H. Lecocq (1998), "A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem", *Tech.rep. Service de Robotique et Automatisation*.

Brucker, P., A. Drexel, R. Möhring, K. Neumann, and E. Pesch (1999), "Resource-constrained project scheduling: notation classification, models and methods", *European Journal of Operational Research*, **112**(1), 3–41.

Coelho, J and M. Vanhoucke (2011), "Multi-mode resource-constrained project scheduling using rcpsp and sat solvers", *European Journal of Operational Research*, **213**(1), 73–82.

Davis, M., G. Logemann, and D. Loveland (1962), "A machine program for theorem-proving", *Communication of the Acm*, **5**(7), 394–397.

Davis, M. and H. Putnam (1960), "A computing procedure for quantification theory", *Journal of the Acm*, **7**(3), 201–215.

De Moura, L. and N. Bjørner (2008), "Z3: An efficient smt solver", *Lecture Notes in Computing Science*, 337–340.

Demeulemeester, E. and W. Herroelen (1992), "A branch-and-bound procedure for the multiple resource-constrained project scheduling problem", *Management Science*, **38**(12), 1803–1818.

Elmaghraby, S. E. (1964), "An algebra for the analysis of generalized activity networks", *Management Science*, **10**(3), 494–514.

Elsayed, E. (1982), "Algorithms for project scheduling with resource constraints", *International Journal of Production Research*, **20**(1), 95–103.

Erenguc, S. and T. Ahn (2001), "The resource constrained project scheduling problem with multiple crashable modes: an exact solution", *Naval Research Logistics*, **28**, 107–127.

Floyd, R.W. (1962), "Algorithm 97: shortest path", *Communications of the ACM*, **5**(6), 345.

Goncharov, E.N. and V.V. Leonov (2017), "enetic algorithm for the resource-constrained project scheduling problem", *Autom Remote Control*, **78**.

Harmtmann, S. (1998), "A competitive genetic algorithm for resource-constrained project scheduling", *Naval Research Logistics*, **45**(7), 733–750.

Hartmann, S. and D. Briskorn (2010), "A survey of variants and extensions of the resource-constrained project scheduling problem", *Journal of Operational Research*, **207**(1), 1–14.

Hartmann, S. and R. Kolisch (2000), "Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem", *European Journal of Operation Research*, **127**(2), 394–407.

Hartmann, S. and R. Kolisch (2006), "Experimental investigation of heuristics for resource-constrained project scheduling: an update", *European Journal of Operational Research*, **174**(1), 23–37.

Horbach, A. (2010), "A boolean satisfiability approach to the resource-constrained project scheduling problem", *Annals of Operations Research*, **181**(1), 89–107.

Kolisch, R. and E Drexl (1996), "Adaptive search for solving hard project scheduling problems", *Naval Research Logistics*, **43**(1), 23–40.

Kolisch, R., K. Sprecher, and Drexl K. (1995), "Characterization and generation of a general class of resource-constrained project scheduling problems", *Manegement Science*, **41**(1), 1693–1703.

Kuster, J and J Dietmar (2006), "Handling airport processes based on resource-constrained project scheduling", *Advances in Applied Artificial Intelligence*, 166–176.

Marques-Silva, J.P. and K.A. Sakallah (1999), "Grasp: a search algorithm for propositional satisfiablility", *Ieee Transactions on Computers*, **48**(5), 506–521.

Nonobe, K. and T. Ibaraki (2006), "A metaheuristic approach to the resource constrained project scheduling with variable activity durations and convex cost functions", *Perspectives in Modern Project Scheduling*, **92**.

Pellerin, R., N. Perrier, and F. Berthaut (2020), "A survey of hybrid metaheuristics for the resource-constrained project scheduling problem", *European Journal of Operational Research*, **280**(2), 395–416.

Pritsker, A, L Watters, and P Wolfe (1969), "Multiproject scheduling with limited resources: A zero-one programming approach", *Management Science*, **16**(1), 93–107.

Sprecher, A., S Hartmann, and A. Drexl (1997), "An exact algorithm for project scheudling with multiple modes", *Operations-Research-Specktrum*, **19**(3), 195,203.

Vanhoucke, M. and J. Coelho (2016), "An approach using sat solver for the rcpsp with logical constraints", *European Journal of Operation Research*, **249**(2), 577–591.