SAT-based optimisation for the resource-constrained project scheduling problem with time-dependent resource capacities and requests

Jelle Pleunes
Supervisor: Emir Demirović
EEMCS, Delft University of Technology, The Netherlands

June 18, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

In this paper, a variant of the resource-constrained project scheduling problem is discussed. This variant introduces time-dependence for resource capacities and requests, making the problem a more realistic model for many practical applications such as production scheduling and medical research project planning. The main aim of this paper is to define a Boolean satisfiability (SAT) formulation for this variant, such that schedules with a minimal total duration can be found efficiently using a SAT solver. We introduce such a formulation which is then used to implement an exact solving approach, of which performance is compared to another approach based on satisfiability modulo theories (SMT). Our experiments show that the SAT-based approach is efficient, in that it outperforms the SMT-based approach for test instances with a larger amount of activities.

1 Introduction

The Resource-Constrained Project Scheduling Problem (RCPSP) is a popular scheduling problem consisting of a set of resources, and a set of activities that use (request) these resources. All activities must be scheduled by means of assigning a start time. A commonly used goal is to minimise the total duration of the project (makespan). Each resource has a limited availability (capacity) which may not be exceeded at any point during the execution of the project. Additionally, there is a notion of order between activities, enforced by precedence constraints. Figure 1 visualises the RCPSP with a small example problem instance. The problem was originally introduced around 1969 [1], with more recent works summarising problem variations and current solving approaches [2].

A common practical application for the standard RCPSP is the modeling of industrial processes, such as production lines or construction projects. In many cases the standard variant of the problem is not used directly, but rather one of its extensions. The Resource-Constrained Project Scheduling Problem with Time-Dependent Resource Capacities and Requests (RCPSP/t) is a variant of the more extensively researched RCPSP, and it introduces time-dependence for resource demands and capacities. This makes the RCPSP/t more suitable for applications where these values cannot be assumed to be constant. Differences compared to the standard RCPSP are illustrated in Figure 1. An example application is a real-world medical research project [3], where staff availability and laboratory equipment are modeled by resource capacities, and where experiments are modeled by activities. Laboratory staff are not constantly available due to vacation or illness (resource capacity), and many experiments do not require constant attention of a researcher as some waiting time may be involved (resource request). Also, certain equipment may only be required at the start or the end of an experiment (resource request). Another application of the model is that of aggregated production scheduling [4], where assembly areas and production orders are modeled as resource capacities and activities, respectively.

In terms of solving the RCPSP, there is no polynomial time solving algorithm, and the problem was proven NP-hard in the strong sense [5]. The RCPSP can be seen as a special case of the RCPSP/t, meaning that the RCPSP/t is also NP-hard. Many (meta-)heuristic and exact approaches for solving the RCPSP exist, with the first exact approaches using Mixed Integer Linear Programming (MILP) [6]. Later exact approaches include Constraint Programming (CP) formulations [7], Boolean satisfiability (SAT) formulations [8], and Satisfiability Modulo Theories (SMT) formulations [9]. Exact approaches guarantee that an optimal solution is found, but solving times quickly become too large for practical use, even

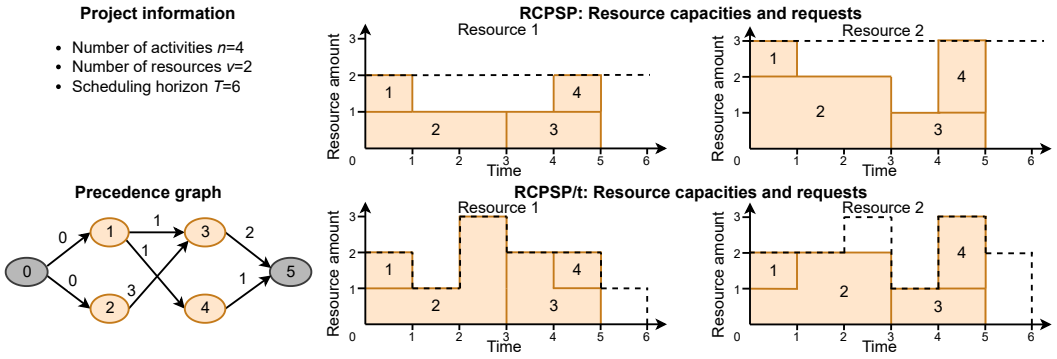

Figure 1: An example RCPSP instance, and an adaption of this instance to the RCPSP/t. These problems only differ in terms of resource constraints, so the left part of the figure is shared for both instances. The top right shows resource resource constraints for the RCPSP, and the bottom right shows this for the RCPSP/t.
The precedence graph is an activity-on-node directed acyclic graph, with dummy start and end activities (gray), and activity durations indicated on the edges. For resource constraints, the dotted line indicates capacity, and the orange blocks indicate requests of activities. The shown schedule (solution) $SOL = (0, 0, 0, 3, 4, 5)$ is optimal (in the sense that the makespan is minimal) for both the RCPSP and the RCPSP/t instances, and respects the precedence and resource constraints in both cases.

for small problem instances [10]. For solving the RCPSP/t, (meta-)heuristic and exact approaches also exist. In terms of heuristic algorithms, an approach using a schedule generation scheme (SGS) with a priority rule [3] exists. Also known to exist are two meta-heuristic algorithms: a genetic algorithm (GA) [4] and a GA-based memetic algorithm (MA) [11]. The GA generally has better performance than the priority rule heuristic, and the MA generally performs better still. For exact solving, to our knowledge, the first and only approach in literature is one based on SMT [9].

SMT is a generalisation of SAT, with SMT solvers (such as Yices [12]) containing additional logic for determining satisfiability of formulas containing expressions from different theories like (non-)linear arithmetic. Valid clauses are $(x - y < 5)$ or $(x = 2^y)$, for instance. SAT is a popular problem that was proven to be NP-complete [13]. The goal is to determine whether a satisfying assignment of Boolean variables $(x_1, x_2, ...)$ exists for some formula usually written in conjunctive normal form (CNF), with clauses (like $(\neg x_1 \vee x_2)$) and literals (like x_1 or $\neg x_1$). In practice the RCPSP/t can be reduced (encoded) into SAT, allowing existing SAT solvers to be applied, but this had not yet been done in existing work. SAT solvers can be seen as general-purpose; they can be applied to problems in numerous different domains such as planning and software verification. The general idea of these solvers is to select a variable, assign it a value, infer the implied consequences (propagation), and to backtrack upon encountering a conflict, eventually finding a satisfying assignment (model).

The main research question of this paper is: "Can the RCPSP/t be solved efficiently using an exact SAT-based approach?". To answer this question, a SAT encoding of the problem must be defined first. Then the encoded problem should be solved by a SAT solver, showing good performance. We define solver 'performance' as lower times for certifying

optimal solutions, and/or better objective values for solutions found within some time limit. Another question to be considered, is whether it is possible to improve performance of a SAT-based solving approach, by augmenting the variable selection procedure of the solver using state-of-the-art heuristic ideas. Due to time limitations, answering this question is left to future work, however we do provide ideas for augmentations. SAT solvers are already being modified to improve performance in specific domains [14], for instance MiniSat [14] was adapted to efficiently solve benchmark instances of the RCPSP [8]. There, the variable selection procedure was modified to increase branching priority for so-called 'process' variables. Existing research into exact RCPSP/t solving has only used SMT [9] without modifying any solver, so it would be interesting to see if problem-specific augmentations can improve performance. SAT solvers are more suitable than SMT solvers for implementing augmentations, since the algorithm is simpler. Before researching an exact SAT-based approach, we explore a heuristic solving approach to have a baseline for comparing performance of exact approaches, and to find ideas for SAT solver augmentations for future work.

We start by formally defining the RCPSP/t in Section 2. This work describes (Section 3) and provides implementations written in C++ for a priority rule heuristic algorithm [3], and exact approaches based on SMT [9], SAT, and MaxSAT (maximum satisfiability, an optimisation extension of SAT). These last two use a new SAT encoding for the RCPSP/t, which is an adaptation of the SMT encoding [9] where the precedence constraints use a SAT encoding [8], instead of the original integer difference logic (IDL) encoding. Then, in Section 4 performance measurements on the J30 and J120 data sets [3] are presented and discussed. Our measurements show that the exact approaches are significantly slower than the simple heuristic approach, but they provide higher quality solutions. It can also be seen that on average the SAT-based and MaxSAT-based approaches outperform the SMT-based approach for large test instances. In Section 5 responsibility and ethical aspects of this research are discussed. Finally, we discuss our conclusions and suggestions for future works in Section 6. Future works can use any (Max)SAT solver for solving the RCPSP/t exactly, using the SAT formulations proposed in this work, making research into heuristic solver augmentations easier.

2 Formal definition of RCPSP/t

The resource-constrained project scheduling problem with time-dependent resource capacities and requests (RCPSP/t) consists of finding a start time for each activity (schedule) minimising the duration of the project (makespan), while respecting precedence relations and resource availabilities [15]. A set of activities is given, all of which should be assigned a start time. Each activity is defined to have a duration and a set of successors. An activity can only start if all its predecessors have closed (finished) at or before that time. At each time step during an activity's execution a resource request is defined, for each (renewable) resource. A renewable resource capacity is defined for each time step within the predefined scheduling horizon, for each of these resources. At no time should the cumulative request of the running activities exceed the capacity of any resource.

Formally, we can define the RCPSP/t as a tuple (V, p, E, R, B, b) [9].

- $V = \{0, 1, ..., n, n+1\}$ is the set of all activities, where 0 and $n+1$ are dummy activities for the start and the end of the project respectively.
- $p \in \mathbb{N}^{n+2}$ is the vector containing the duration for each activity, where p_i is the duration of activity i. We have that $p_0 = p_{n+1} = 0$, but $p_i > 0$ for all $0 < i < n + 1$.

- E is the set of pairs of activities representing precedence relations. If $(i, j) \in E$ then activity i must finish at or before the time step at which j starts. An activity-on-node precedence graph $G = (V, E)$ can be constructed. Any cycle in G would make the problem instance trivially infeasible, so it is assumed that the precedence graph is acyclic. Furthermore, it is a convention that a path exists from dummy activity 0 to any $i \in V$, and that from any $i \in V$ a path exists to dummy activity $n + 1$.
- $R = 1, ..., v$ is the set of renewable resources.
- $B \in \mathbb{N}^{v \times T}$ is a matrix of renewable resource capacities, where $B_{k,t}$ is the available amount of resource $k \in R$ at time step $t \in \{0, ..., T - 1\}$. Here T is the predefined scheduling horizon of the problem instance, which also acts as an upper bound on the makespan because resource capacities are not defined after this time.
- b is a three-dimensional irregular matrix, where $b_{i,k,e} \in \mathbb{N}$ is activity i's request for resource k at time step e after the start of i. The matrix is irregular, because not all activities have the same duration. Dummy activities have duration 0, so they do not have any resource requests.

A solution is a vector $SOL \in \mathbb{N}^{n+2}$ with a start time $SOL_i \in SOL$ for all $i \in V$, satisfying all precedence and resource constraints. The decision variant of the problem consists of determining whether any valid schedule exists given some upper bound on the makespan. An optimal solution (meaning it has a minimal makespan) can be found by repeatedly solving the decision variant of the problem with an incrementally decreasing upper bound.

3 Studied solving approaches

This section describes the solving methods studied in this paper. The first method is not exact, while the rest are exact. We provide an implementations written in C++[1].

A preprocessing step required for all solving approaches discussed in this section is to calculate, for each activity $i \in V$, the earliest and latest feasible start time (ES^*_i and LS^*_i), and the earliest and latest feasible close time (EC^*_i and LC^*_i). These times are the earliest or latest feasible considering precedence constraints, also keeping in mind resource feasibility (indicated by the '*'), meaning that times with insufficient resources are excluded [3]. Following the recursive definition by Hartmann [3], we have that $ES^*_i = \max\{ES^*_j + p_j \mid j \in P_i\}$ and $LC^*_i = \min\{LC^*_j - p_j \mid j \in S_i\}$. Here P_i and S_i denote the predecessors and successors of i, respectively. Naturally, we also have that $LS^*_i = LC^*_i - p_i$ and $EC^*_i = ES^*_i + p_i$. Finally, for dummy activities we know that $ES^*_0 = LS^*_0 = 0$ and $LS^*_{n+1} = UB$. Here UB is some upper bound on the makespan which is usually the predefined scheduling horizon T of the problem, but it could also be some computed value, as will be explained later. This preprocessing step is able to detect relatively simple cases where the instance is infeasible (not all cases). When we have that $EC^*_i > T$ or $LS^*_i < 0$ for any $i \in V$, then the instance is proven to be infeasible. If resources are scarce it is also possible that $LC^*_i - ES^*_i < p_i$ occurs, which also proves that there is no feasible solution [3].

For all studied exact approaches the encoding uses the same resource constraints, which are encoded into SAT in the way described below. The resource constraints are pseudo-Boolean (PB), so they are of the form $\sum_{i=1}^{n} q_i x_i \# K$ with constants $q_i, K \in \mathbb{Z}$, Boolean

[1]Simple heuristic algorithm: `https://github.com/jpleunes/rcpspt-heuristic`.
Exact approaches using SMT/SAT/MaxSAT encoding: `https://github.com/jpleunes/rcpspt-exact`.

0/1 variables x_i, and $\# \in \{<, \leq, =, \geq, >\}$. Multiple ways of encoding PB constraints into SAT exist. A paper by Bofill et al. [9] studies, among others, a SAT encoding of PB constraints based on Binary Decision Diagrams (BDD), and a SAT encoding of PB at-most-one (PB(AMO)) constraints based on Multi-valued Decision Diagrams (MDD). Compared to the PB encoding, PB(AMO) was shown experimentally to result in better performance. However, in this work we used the PB encoding (based on BDD), since it is simpler to implement while still giving good performance. Each PB constraint is encoded by first constructing a Reduced Ordered BDD (ROBDD), then generating two SAT clauses for each non-terminal node, and finally completing the encoding by adding three unary clauses [16].

3.1 Simple heuristic algorithm

The first studied solving approach is a tournament heuristic using a priority rule, as described by Hartmann [3]. A preprocessing step of this approach is to calculate the priority value for each activity $i \in V$. An activity's priority value can be based (partially or completely) on one of the bounds of the time window $[ES_i^*, LC_i^*]$. Different definitions for the priority value can be used. An example is the critical path $CP_i = T - LS_i^*$, with longer critical paths getting higher priority. This definition uses mostly the precedence constraints. The influence of resource constraints can be increased by using the "critical path and resource utilisation" (CPRU) [3] definition, which we will use. The resource utilisation (RU) value is based on the fraction of the available resource capacity that is requested by an activity (and its successors). A larger RU increases priority. Using CPRU very slightly improves solution quality, but also slightly increases run times [3].

The scheduling algorithm is based on a serial schedule generation scheme (serial SGS). Such a scheme constructs schedules where no activity can be left-shifted. This approach reduces the size of the search space, excluding schedules that are of low average quality. Optimal solutions could also be excluded, however. The algorithm runs multiple passes (1,000 in our case) and finally picks the best schedule that is generated from these. Within each pass, the activities are scheduled one by one. The next activity to be scheduled is determined by running a so-called 'tournament', which is a randomised procedure where the activity's priority value corresponds to the probability of being selected.

3.2 Exact approach using SMT encoding

The first exact solving approach that we study is based on Satisfiability Modulo Theories (SMT). The approach is mostly based on a paper by Bofill et al. [9].

3.2.1 Problem encoding

The formulation used for this approach uses Boolean and integer variables. The formulation is time-indexed, meaning that time is discretised in unit intervals, such that variables correspond to these intervals. This naturally fits the time-dependent definition of resource capacities and requests in the RCPSP/t. Boolean 0/1 variables $y_{i,t}$ are used to to denote that activity $i \in V$ starts at time t. For precedence constraints integer variables S_i are used to denote the start time of activity i. In this formulation, precedence constraints are expressions in the form of integer difference logic (IDL), which are basic rewritings of $y - x < c$ with variables $x, y \in \mathbb{Z}$ and constant $c \in \mathbb{Z}$. The complete formulation used in this work mostly follows the paper by Bofill et al. [9], and looks as follows:

$$S_0 = 0 \tag{1}$$

$$S_i \geq ES_i^* \qquad \forall i \in V \setminus \{0\} \quad (2)$$

$$S_i \leq LS_i^* \qquad \forall i \in V \setminus \{0\} \quad (3)$$

$$S_j - S_i \geq l_{i,j} \qquad \forall (i,j,l_{i,j}) \in E^* \quad (4)$$

$$y_{i,t} \leftrightarrow S_i = t \qquad \forall i \in V, \forall t \in STW(i) \quad (5)$$

$$\sum_{\substack{i \in A, \text{ s.t.} \\ t \in RTW(i)}} \sum_{\substack{e \in 0..p_i-1 \text{ s.t.} \\ t-e \in STW(i)}} b_{i,k,e} \cdot y_{i,t-e} \leq B_{k,t} \qquad \forall k \in R, \forall t \in H \quad (6)$$

Constraint (1) sets dummy activity 0 to start at time 0. Constraints (2) and (3) enforce that activities do not start outside of their feasible start times. Constraints (4) enforce the (extended) precedence relations. A precedence relationship (immediate or extended) between two activities i and j is encoded using a time lag $l_{i,j}$, which is a lower bound on the difference between the start times of these activities. This is the minimum length of any path from i to j in the precedence graph $G = (V, E)$, where E is the set of weighted edges. These time lags can be computed using the well-known Floyd-Warshall algorithm. The values $l_{i,j}$ can also be seen as the edge weights for the extended precedence graph $G^* = (V, E^*)$, meaning that $(i, j, l_{i,j}) \in E^*$. The extended precedence graph G^* is the transitive closure of the precedence graph G, which means that it contains an edge $(i, j, l_{i,j})$ for any two activities i and j between which a path exists in G. Constraints (5) enforce consistency between the Boolean and integer start variables, with $STW(i)$ denoting the start time window $[ES_i^*, LS_i^*]$ of activity i. Our definition uses the values ES_i^* and LS_i^* in multiple constraints, while Bofill et al. [9] used different values $ES_i = l_{0,i}$ and $LS_i = UB - l_{i,n+1}$. Their definition results in larger time windows, because the minimum path lengths between two activities are used without taking into account that an activity can only start once all predecessors have finished, ultimately increasing the size of the encoding. The resource constraints (6) ensure that for each resource $k \in R$, the capacity $B_{k,t}$ is respected for all time steps $t \in H$. Here $H = \{0, ..., UB - 1\}$ is the set of time steps within the upper bound on the makespan UB. The calculation of the upper bound is explained after this problem encoding, as part of the minimisation of the makespan. $RTW(i)$ denotes the run time window $[ES_i^*, LC_i^* - 1]$ of activity i, and p_i is the duration. If the literal $y_{i,t-e}$ is true, then that implies that at time t activity i has been running for e units of time. This is also why the resource request at time e ($b_{i,k,e}$) is used.

The time lags $l_{i,j}$ used for constraints (4) can be increased using energetic reasoning [9]. This makes the SMT encoded problem more constrained, reducing the search space. The reasoning is based on the fact that for any two activities $i, j \in E^*$, the time lag $l_{i,j}$ must be wide enough such that all activities a that must be executed between i and j (meaning that $(i, a, l_{i,a} \in E^*) \wedge (a, j, l_{a,j} \in E^*)$) can be executed without exceeding any resource constraints. This means that a lower bound on the distance between the end of i and the start of j can be computed for some resource k [9]:

$$RLB_{i,j,k} = \left\lceil \frac{1}{\max_{t \in H}(B_{k,t})} \cdot \sum_{\substack{a \in V \text{ s.t.} \\ (i,a,l_{i,a}) \in E^*, \\ (a,j,l_{a,j}) \in E^*}} \sum_{u \in \{0,\ldots,p_a-1\}} b_{a,k,u} \right\rceil \quad (7)$$

This means that the updated time lags $l'_{i,j}$ can be calculated as follows [9]:

$$l'_{i,j} = \max(l_{i,j}, p_i + \max_{k \in R}(RLB_{i,j,k})) \qquad \forall (i,j,l_{i,j}) \in E^* \quad (8)$$

In practice, however, this definition was found to lead to incorrect results, with feasible problem instances being classified as infeasible. The reason for this is unknown. In this work, an alternative definition that is still able to increase some time lags (while giving correct results) was used:

$$l'_{i,j} = \max(l_{i,j}, \max_{k \in R}(RLB_{i,j,k})) \qquad \forall (i,j,l_{i,j}) \in E^* \quad (9)$$

Whenever a time lag is increased this way, the update must be propagated to other time lags. This is done by running the Floyd-Warshall algorithm again.

3.2.2 Minimisation of the makespan

For this approach, the general idea is to call the solver multiple times for checking satisfiability, with an incrementally decreasing upper bound. Once the encoded problem becomes unsatisfiable, we know that the last found solution was optimal.

It is desirable for the initial upper bound to be as small as possible, so that fewer satisfiability checks have to be run on the solver. Our implementation calculates an initial upper bound UB by running the priority rule heuristic algorithm described earlier. The amount of passes that are run was chosen to be the number of (non-dummy) activities n multiplied by 5 (an arbitrarily chosen number), so that the number of passes increases as the problem becomes harder. This differs from the work by Bofill et al. [9], where the Parallel Scheduling Generation Scheme (PSGS) was used. In this work the PSGS is not used, because an implementation of the heuristic algorithm was already readily available to be used. If the heuristic algorithm finds a solution the makespan of that solution is used as the upper bound, if not the horizon T from the problem definition is used. A lower bound on the makespan LB is also returned, in the form of the earliest feasible start time of the end dummy activity, ES^*_{n+1}. To allow for fair comparisons of different encodings, all randomness was eliminated from the heuristic algorithm by setting a seed. Thus, for one instance of the problem the same initial solution (or no initial solution) will always be found.

The optimisation procedure was inspired by the paper by Bofill et al. [9], and can be found in Algorithm 1. Function $check(ENC)$ calls the solver to determine satisfiability of the encoded problem ENC. It returns $(\top, MODEL)$ if a model was found, $(\bot, \{\})$ otherwise. The function $getSolution(ENC, MODEL)$ gets the solution vector SOL for the original problem, from model $MODEL$ of encoding ENC. The makespan of SOL is SOL_{n+1}.

3.3 Exact approach using SAT encoding

A new exact approach that we propose is based on the standard SAT problem, using another time-indexed formulation. The most important difference with the SMT formulation is that SAT clauses are used for precedence constraints, instead of IDL clauses. This means that the entire encoding consists of only propositional logic, allowing for SAT solvers to be used, instead of more complex SMT solvers.

3.3.1 Problem encoding

The formulation used for this approach uses Boolean 0/1 variables $y_{i,t}$ to denote that activity $i \in V$ starts at time t, and Boolean 0/1 variables $x_{i,t}$ to denote that i is running at time t. We will refer to variables $y_{i,t}$ as *start* variables, and $x_{i,t}$ as *process* variables. The formulation

Algorithm 1 Minimise makespan

Require: Encoded problem instance ENC, lower bound LB, and upper bound UB
Ensure: Return an optimal solution if ENC is satisfiable, an empty vector otherwise

```
(SAT, MODEL) ← check(ENC)
if SAT then
    SOL ← getSolution(ENC, MODEL)
    UB ← SOL_{n+1} − 1
else
    return {}
end if
while SAT and UB ≥ LB do
    ENC ← ENC ∪ {S_{n+1} ≤ UB}
    (SAT, MODEL) ← check(ENC)
    if SAT then
        SOL ← getSolution(ENC, MODEL)
        UB ← SOL_{n+1} − 1
    end if
end while
return SOL
```

contains clauses based on a paper by Horbach [8] for precedence constraints. The resource constraints are the same as in the SMT encoding, they are based on the paper by Bofill et al. [9]. The complete formulation used in this work looks as follows:

$$y_{0,0} \tag{10}$$

$$\neg y_{i,s} \bigvee_{t \in \{ES_j^*, \ldots, \min(s-p_j, LS_j^*)\}} y_{j,t} \qquad \forall (j,i) \in E, \forall s \in STW(i) \tag{11}$$

$$\bigvee_{s \in STW(i)} y_{i,s} \qquad \forall i \in V \setminus \{0\} \tag{12}$$

$$\neg y_{i,s} \vee x_{i,t} \qquad \forall i \in V, \forall s \in STW(i), \forall t \in \{s, ..., s+p_i-1\} \tag{13}$$

$$\neg x_{i,t} \vee x_{i,t+1} \vee y_{i,t-p_i+1} \qquad \forall i \in V, \forall t \in \{EC_i^*, ..., LC_i^*-1\} \tag{14}$$

$$\sum_{\substack{i \in A, \text{ s.t.} \\ t \in RTW(i)}} \sum_{\substack{e \in 0..p_i-1 \text{ s.t.} \\ t-e \in STW(i)}} b_{i,k,e} \cdot y_{i,t-e} \leq B_{k,t} \qquad \forall k \in R, \forall t \in H \tag{15}$$

Constraint (10) sets dummy activity 0 to start at time 0. Constraints (11) enforce precedences by ensuring that for each $i \in V$, all predecessors start early enough to allow i to start at the time that is being considered. Here it is important to use $\min(s - p_j, LS_j^*)$, and not simply $s - p_j$, because the time window of j and the time window of i may have a 'gap' in between. Such gaps are caused by the consideration of resource feasibility for calculating earliest and latest feasible times. Constraints (12) make sure that each activity starts (at least) once. Constraints (13) enforce consistency between start variables and the corresponding process variables. Constraints (14) are redundant, but numerical tests in other work have shown that including these reduces the execution time of the solver [8]. Constraints (15) enforce resource constraints, with the same definition as in the SMT formulation.

3.3.2 Minimisation of the makespan

The general idea for minimising the makespan is the same as that for the SMT-based approach. Algorithm 1 is also used here, with one exception. On line 9 it is no longer possible to directly add the clause $S_{n+1} \leq UB$ to the encoding, since the encoding does not have integer variables S_i. To solve this, whenever UB is assigned a new value, the old value is kept in a separate variable UB_{OLD}. Then instead of line 9, multiple unit clauses are added to the encoding, preventing activity $n+1$ from starting after UB:

$$\neg y_{n+1,t} \qquad \forall t \in \{UB+1, ..., UB_{OLD}\} \quad (16)$$

It is not enough to simply use $\neg y_{n+1,UB+1}$, because the solver may find a solution with a makespan that is multiple time steps smaller than that of the previous solution.

3.4 Exact approach using MaxSAT encoding

The last exact solving approach that we study uses a MaxSAT encoding, which is a small extension to the previously described SAT encoding. The weighted MAX-SAT problem is a generalisation of SAT, where the objective is to satisfy all hard clauses, while maximising the sum of weights assigned to the soft clauses. In other words, the cumulative weight of the soft clauses that are violated should be minimised. On their own, standard SAT solvers only solve decision problems, while MaxSAT solvers solve optimisation problems (such as the RCPSP/t) directly without an external optimisation procedure (such as Algorithm 1). Such an optimisation solver may perform better for our purposes.

3.4.1 Problem encoding

The previously described SAT encoding is used here (with all clauses as hard clauses). The objective of minimising the makespan is defined by adding soft unit clauses $y_{n+1,t}, \forall t \in STW(n+1)$, starting with weight 1 for $t = LS^*_{n+1}$, and incrementing the weight for every step that t decreases. This ensures that the solver is penalised more for violating earlier start times for activity $n+1$ (the start time of $n+1$ is the makespan). To prevent $n+1$ from being scheduled multiple times to satisfy more soft clauses, we also add hard clauses:

$$\neg y_{n+1,t} \vee \neg y_{n+1,u} \qquad \forall t, u \in STW(n+1) \text{ s.t. } t \neq u \quad (17)$$

These clauses ensure that $n+1$ is scheduled at most once.

4 Experiments

Measurement results for all implementations are shown and discussed in this section. For this section we will refer to the priority rule heuristic algorithm as 'PR', the SMT encoding-based algorithm as 'SMT', the SAT encoding-based algorithm as 'SAT', and the MaxSAT encoding-based algorithm as 'MaxSAT'.

4.1 Experimental setup

The data sets used for testing are the J30 (2,880 instances, 30 activities per instance) and J120 (3,600 instances, 120 activities per instance) sets that were adapted to be used for the RCPSP/t by Hartmann [3]. These test instances[2] are based on the frequently used J30 and

[2]The instances can be downloaded from: `http://www.om-db.wi.tum.de/psplib/newinstances.html`.

J120 instances from the PSPLIB problem library [17]. To our knowledge, these adapted instances are the only publicly available test data for the RCPSP/t.

Measurements were run on a machine with an Intel® Xeon® Gold 6248R processor at 3.00GHz with 8GB RAM. The SMT and SAT approaches used Yices 2.6.4 [12] as the solver, communicating through the provided C API. The solver context was configured with `multi-check` mode to allow for multiple satisfiability checks to be called, with a decreasing upper bound. For SMT the default solver logic was configured as `QF_IDL` (quantifier-free integer difference logic), and for SAT this was configured as `NONE` (propositional logic only). The programs were given a time limit of 60 seconds per problem instance, after which they were interrupted, causing the program to output its current best solution.

For the MaxSAT approach the amount of RAM was increased to 32GB, as 8GB was too little. The Pumpkin solver was used. This is a MaxSAT solver that was provided by our supervisor E. Demirović. The solving approach starts by using the encoder program to write the encoded problem to a WCNF-format file. Then the Pumpkin solver reads this file, and is given a time limit of 60 seconds for solving (using the `-time` argument). The resulting model (file location is specified using the `-output-file` argument) is finally read by the encoder program, which converts it to a solution vector for the RCPSP/t problem.

4.2 Results

Table 1 shows the the average size for different encodings, in terms of number of variables and number of clauses. 'SAT' refers to the encoding used by both SAT and MaxSAT. For SMT, the small number of integer variables corresponds to the number of activities (including dummy activities). Any clauses added for minimising the makespan are not counted.

encoding(#act)	#Bv	#iv	#cl
SMT(30)	99,270	32	198,584
SAT(30)	100,177	0	202,498
SMT(120)	2,632,783	122	5,267,390
SAT(120)	2,642,216	0	5,317,721

Table 1: Average size for different encodings on the J30 and J120 data sets. #act denotes the number of (non-dummy) activities per instance of the data set, while #Bv, #iv, and #cl denote the average number of Boolean variables, integer variables, and clauses respectively.

For the performance measurements of exact approaches, encoding time, search (solving) time, and total execution time were measured. The encoding time includes calculating the initial solution using the heuristic algorithm, initialising the solver (for SMT and SAT), encoding the problem instance into the solver's logic, and writing to the WCNF file (for MaxSAT). Search time is the time spent on the optimisation procedure. The total execution time includes these measured times, and I/O operations (reading and parsing the input file and writing results to the console).

We count an instance as *certified* when an optimal solution is found, or when the solver finds the instance to be infeasible. Furthermore, an instance is counted as *solved* when it is *certified*, or when any solution (not certified to be optimal) is found. MaxSAT is not always able to provide a valid intermediate solution before finding the optimum, in which case the initial heuristic solution is simply considered. As a measure of solution quality, the average distance of the makespan to some 'lower bound' Δ_{LB} is used. This bound is not always the optimal solution, but rather an approximation. For each instance, the bound is

the makespan that was found in the work by Bofill et al. [9] where a PB(AMO) encoding of resource constraints was used, with a time limit of 600 s. The results[3] for the PB(AMO) encoding were used, because these were shown to have the smallest makespans overall. We compute the distance as: $\frac{MS-LB}{LB} \cdot 100$, with makespan MS and 'lower bound' LB.

In the results for the 120-activity instances (Table 2) it could be seen that SMT spent 3-4 seconds longer on encoding than SAT, on average. Thus, SMT effectively had less search time for hard instances where the time limit was reached. To allow for a fair comparison, additional measurements were taken for SMT with a time limit extended by 4 seconds (totalling 64 s). There are no such additional measurements for MaxSAT, because here the 60 s time limit only applies to searching, rather than the entire solving approach.

Due to some seemingly random inconsistencies when measuring CPU time for SMT and SAT, search time was reported as 0 for less than 1% of instances, while the program reached the 60 s time limit and spent less than 10 seconds on encoding. In these exceptional cases the search time was corrected using $t_{search} = t_{total} - t_{enc}$, which is very close to what this time would have been in reality. Measurements for MaxSAT also contain one incorrect data point, where the solver marked a feasible instance as infeasible. The point was manually corrected, as if the solver certified this instance. Overall, comparisons are not affected in any significant way.

Table 2 shows the measurements of solving times and solution quality for the different exact solving approaches. Values are from the second run of measuring, due to bugs in the implementation for the first run. Complete measurement data (for both runs, data for run 1 manually corrected) are included in the GitHub repository for the exact approaches.

Table 2 also shows measurements of execution times and solution quality for the priority rule heuristic algorithm (PR). The encoding and search times are not applicable here. In terms of certifying solutions, the only certifications that this algorithm is able to provide are when the preprocessing steps prove that the instance is infeasible. Performance was measured with the number of passes (tournaments) set to 100 and 1,000. The measured total execution times again include I/O operations.

4.3 Discussion

To start, the average sizes of the SMT and SAT encodings are similar in terms of the number of Boolean variables. In the SMT encoding these variables are only used for the PB encoding of resource constraints. The increase in size going from SMT to SAT, caused by the addition of process variables which are only used for redundant constraints, is relatively small.

The two encodings are also similar in terms of the average number of clauses. The SMT encoding only contains $\mathcal{O}(n^2)$ clauses for precedence constraints, since pairs of activities are considered in constraints (4). It should be noted that these are IDL clauses, so these are more complex for the solver to check compared to propositional logic clauses. The small amount of clauses for precedence constraints means that the vast majority of clauses are generated as part of the PB encoding of resource constraints. The increase in the amount of clauses going from SMT to SAT is caused by a combination of multiple clauses being required for encoding one precedence relation (although these are not IDL clauses in this case), and the fact the SAT encoding includes redundant clauses. The increase in size is again relatively small.

For both encodings, the number of Boolean variables and the number of clauses both increase by a factor of roughly 26 when increasing the size of the problem from 30 to 120

[3]These results were downloaded from: `https://ima.udg.edu/Recerca/lai/rcpspt2smt/index.html`.

	t_{enc}	t_{search}	t_{total}	#s	#c	Δ_{LB}
J30 (2,880 instances, 2,826 feasible)						
SMT	0.12	0.80	0.93	2,880	**2,875**	**0.00**
SAT	0.10	1.33	1.44	2,880	2,845	0.02
MaxSAT	0.21	1.43	1.64	2,880	2,843	0.02
PR(100 pass)	n.a.	n.a.	0.00	2,864	43	1.63
PR(1,000 pass)	n.a.	n.a.	0.05	2,866	43	0.79
J120 (3,600 instances, 3,600 feasible)						
SMT	7.22	27.90	35.13	3,600	1,758	6.71
SMT(64 s)	7.07	30.94	38.01	3,600	1,801	6.23
SAT	3.73	28.54	32.28	3,600	**1,854**	4.25
MaxSAT	6.74	35.87	42.60	3,600	1,822	**3.18**
PR(100 pass)	n.a.	n.a.	0.07	3,600	0	8.20
PR(1,000 pass)	n.a.	n.a.	0.53	3,600	0	6.79

Table 2: Performance measurements for the different solving approaches. The average encoding, search (solving), and total execution times are denoted by t_{enc}, t_{search}, and t_{total} respectively. Times are measured in seconds. The amount of *solved* instances is denoted by #s, and the amount *certified* is denoted by #c. The average deviation from the 'lower bound' is denoted by Δ_{LB}, in %.

activities (a factor 4). This underlines how quickly the complexity of the RCPSP/t grows.

In terms of performance on the J30 data set, SMT certified the largest number of instances (99.8%). SAT certified 98.8% and MaxSAT certified 98.7% in comparison. All exact approaches were able to solve all instances. SMT also resulted in the best average solution quality (smallest Δ_{LB}). For these small instances, the encoding times were quite similar, while the average search time was 66.3%-78.8% lower for SMT. The increased search time for SAT and MaxSAT is likely due to the time limit being reached more often.

On the J30 data set, PR solved roughly the same amount of instances when running with either 100 or 1,000 passes (99.4%-99.5% solved). The average solution quality improved significantly when the amount of passes was increased from 100 to 1,000. With ten times more passes the total execution time also increased roughly tenfold (as expected), but this time still remained negligible overall. The J30 data set contains 54 infeasible instances, 43 of which are found to be infeasible by the preprocessing steps.

For the J120 data set, SAT was able to certify the largest number of instances (51.5%), while MaxSAT certified 50.6% and SMT certified 48.8%. When compensating for the longer encoding times for SMT by increasing the time limit to 64 seconds, 50.0% could be certified. The increased time limit also improved average solution quality for SMT, but MaxSAT and SAT clearly have better average solution quality. The improved solution quality going from SAT to MaxSAT could be caused by the longer search times, due to MaxSAT not needing to encode the problem in this time, and due to MaxSAT's stopping mechanism allowing the time limit to be exceeded. This time limit is exceeded by more than 5 seconds ($t_{search} > 65$) for 922 out of 3,600 instances, with $t_{search} = 69.0$ on average in these cases, and up to $t_{search} = 112.2$ in extreme cases. On the other hand, MaxSAT has a small disadvantage with invalid intermediate solutions being discarded, with the initial heuristic solution being considered instead. For SMT, the average solution quality (with 60 s and 64 s time limits) is still close to that of PR with 1,000 passes. This means that, within the given time limit, the exact optimisation procedure of SMT improved initial solutions a small amount, compared

to SAT. The exact approaches solved all instances in J120. Finally, SAT has the lowest average total execution time out of the exact solutions, partially due to its low average encoding time. Altogether, the lower execution times and higher quality solutions give an indication that the SAT approach scales better than the SMT approach when the number of activities in the problem increases, while SMT is more efficient for problems with a small number of activities. MaxSAT also seems to scale better than SMT, giving the highest quality solutions, but it should be noted that average execution times were higher.

PR solved all J120 instances with only 100 passes. Average execution time increased significantly going from 100 to 1,000 passes, but all instances were solved in less than 1 second. Average solution quality also improved with more passes. No instances could be certified, since they are all feasible.

4.4 Comparison with state-of-the-art

To our knowledge, prior to this work no SAT encoding existed for solving the RCPSP/t specifically, so no direct comparisons can be made. However, the SAT and MaxSAT approaches can be compared to an SMT approach, as was done in this section. The SMT-based approach proposed by Bofill et al. [9] was the only exact approach for this problem, to our knowledge. They also provide an executable file for their solving approaches, but the source code is not publicly available. In this work, comparisons were chosen to be done using our own implementation of the SMT approach, so that the SAT implementation is as similar as possible with the encoding being the only difference.

One limitation is that due to time restrictions the exact approaches described in this work do not use PB(AMO) encodings for resource constraints, because the implementation for this is more involved compared to PB encodings. However, since this limitation applies to all studied exact approaches, we can still study the effects of using SAT instead of SMT. We have shown experimentally that, on average for larger test instances, the SAT and MaxSAT approaches give higher quality solutions within the set amount of time. This can be seen as an advancement to the state-of-the-art of exact solving of the RCPSP/t.

Another limitation is that only one solver was measured per exact approach, due to time restrictions. Different solvers supporting the WCNF format (or DIMACS CNF format, with minor adjustments to the code) can easily be used, and may lead to different performance. With the new (Max)SAT encoding available, many (Max)SAT solvers can be used to solve the RCPSP/t exactly, because support for IDL encodings (used in the SMT approach) is not required. 'Normal' (Max)SAT solvers also generally have simpler implementations than SMT solvers, which means that heuristic augmentations for even further improving performance can be implemented more easily with this new (Max)SAT encoding.

5 Responsible research

Multiple aspects must be taken into account when comparing the performance of multiple programs for solving the same problem. The first point is that different algorithms should be compared as fairly as possible. Implementations should preferably all be written by the same programmer, or at least a programmer with comparable skill, to prevent variations that may impact the efficiency of the respective programs. If possible, all implementations should use the same programming language (with the same compiler/interpreter), to prevent variations in performance caused by different kinds of optimisations. All tests should then be run on the same system (not necessarily the machine used for programming, think of a

computing cluster) to ensure that the algorithm being tested is the only variable. In this work, all measured implementations were written by the same programmer, all in the same language, compiled using the same compiler, and all run on the same high performance computing cluster. This paper also describes all studied solving approaches in detail, allowing researchers to write their own implementations.

Another aspect is the reproducibility of the research. All source code that has been written for this work is publicly available, which allows the implementations to be checked for any errors, and for (parts of) the code to be adapted for further research. This also makes running new measurements easier, in case a researcher decides not to write their own implementation. Again, since this paper describes the studied solving approaches in detail, researchers can write their own implementations or use this information for their research in other ways.

The last point discussed here is to make measurement data sets publicly available. Firstly, this makes it possible for others to verify whether the paper presents results objectively and thoroughly. Second, this may aid in future research, where for example problem solutions from these measurement data can be used as a baseline for improvements. Finally, when reproducing the research the original measurement data can be used to check whether the methodology matches that of the original paper. For this work the measurement data are available in the GitHub repositories of the respective implementations.

6 Conclusions and future work

This work explored the possibility of efficiently solving the RCPSP/t using an exact (Max)SAT-based approach. The question of improving performance by heuristically augmenting a SAT solver was introduced, but such augmentations could not be implemented due to time limitations. A simple heuristic solving algorithm was also studied, providing baseline solutions that are improved upon by the exact solving approaches.

The priority rule heuristic algorithm solved 99.8% of all test instances within 1 second, but it does not certify optimality of its solutions. Out of the studied solving approaches it had the lowest average solution quality. Then an SMT approach based on existing work was studied. This approach solved all test instances, and overall it gave the highest average quality solutions for the smaller 30-activity test instances. A new SAT approach, partially based on the SMT approach, was then studied. This approach also solved all test instances, and on average it gave higher quality solutions for the larger 120-activity tests instances. Finally, a MaxSAT approach (a small extension to the SAT approach) gave even higher average quality solutions, possibly due to higher search times.

Conclusions

We conclude that it is possible to exactly solve the RCPSP/t efficiently using both SAT-based and MaxSAT-based approaches, with our experiments showing that these even outperform an SMT-based approach on large test instances. The question of whether performance can be improved by heuristic augmentations to the solver could not yet be answered, but the new (Max)SAT encoding facilitates research into this question.

A new SAT encoding for the RCPSP/t, with a MaxSAT extension, can be seen as the main contributions of this work. Other notable contributions are solver performance measurements for three problem encodings, and C++ implementations for different solving

approaches: a heuristic algorithm, an SMT-based approach, a SAT-based approach, and a MaxSAT-based approach.

A limitation is that the encodings used in this work do not use the more efficient PB(AMO) encoding for resource constraints, because the implementation is more complex. Thus, there is a good chance that the performance of the exact approaches would be improved by replacing the current PB encoding with a PB(AMO) encoding. Another limitation is that only one solver was measured per problem encoding.

Future work

The SAT encoding proposed in this paper can be used in future work for solving the RCPSP/t exactly, using any (Max)SAT solver. The provided C++ implementation can write a MaxSAT encoding to a file in WCNF format, and can also be adapted to write a SAT or SMT encoding to a file instead of communicating with the solver directly, although we recommend researchers to write their own implementations.

Future work could additionally explore different problem-specific heuristic augmentations to the variable selection procedure of a SAT solver. The Pumpkin MaxSAT solver, or an extensible SAT solver such as MiniSat [14] would be a good options for implementing heuristic augmentations. The ordering of variables can be determined statically, meaning before the search procedure starts, or dynamically meaning that the ordering may change during the search procedure. For a static ordering, a possibility would be to always branch on start variables $y_{i,t}$ first, and sort them using some priority value for the corresponding activity i. The priority value could use the definition from the heuristic algorithm discussed in this paper, for example. For each activity, these variables can also be ordered by ascending time t, since we would ideally like to quickly find a solution where activities are scheduled early. A dynamic ordering possibility is to prioritise process variables $x_{i,t}$ by giving them a higher branching priority, as this has previously been shown experimentally to improve performance for an exact approach for solving the RCPSP [8].

References

[1] A. A. B. Pritsker, L. J. Watters, and P. M. Wolfe. "Multiproject scheduling with limited resources: a zero-one programming approach". In: *Manag Sci* 16 (1969), pp. 93–107.

[2] S. Hartmann. *Application of Mathematics and Optimization in Construction Project Management*. Springer Cham, 2021. Chap. Optimization models and solution techniques, pp. 25–50. ISBN: 978-3-030-81123-5. DOI: https://doi.org/10.1007/978-3-030-81123-5.

[3] S. Hartmann. "Project scheduling with resource capacities and requests varying with time: a case study". In: *Flexible Services and Manufacturing Journal* 25.1-2 (2013), pp. 74–93. URL: https://www.proquest.com/scholarly-journals/project-scheduling-with-resource-capacities/docview/1271881162/se-2?accountid=27026.

[4] S. Hartmann. "Time-varying resource requirements and capacities". In: *Handbook on Project Management and Scheduling*. Vol. 1. Springer, 2015, pp. 163–176.

[5] J. Blazewicz, J. Lenstra, and A. Kan. "Scheduling subject to resource constraints: classification and complexity". In: *Discrete Applied Mathematics* 5.1 (1983), pp. 11–24. ISSN: 0166-218X. URL: https://www.sciencedirect.com/science/article/pii/0166218X83900124.

[6] J. H. Patterson and W. D. Huber. "A horizon-varying, zero-one approach to project scheduling". In: *Management Science (pre-1986)* 20.6 (1974). ISSN: 990. URL: https://www.proquest.com/scholarly-journals/horizon-varying-zero-one-approach-project/docview/205804796/se-2?accountid=27026.

[7] A. Schnell and R. F. Hartl. "On the efficient modeling and solution of the multi-mode resource-constrained project scheduling problem with generalized precedence relations". In: *OR Spectrum* 38.2 (2016), pp. 283–303. ISSN: 1436-6304. DOI: https://doi.org/10.1007/s00291-015-0419-6.

[8] A. Horbach. "A Boolean satisfiability approach to the resource-constrained project scheduling problem". In: *Annals of Operations Research* 181 (2010), pp. 89–107. DOI: https://doi.org/10.1007/s10479-010-0693-2.

[9] M. Bofill et al. "SMT encodings for resource-constrained project scheduling problems". In: *Computers & Industrial Engineering* 149 (2020). DOI: https://doi.org/10.1016/j.cie.2020.106777.

[10] H. Chen et al. "A hyper-heuristic based ensemble genetic programming approach for stochastic resource constrained project scheduling problem". In: *Expert Systems with Applications* 167 (2021), p. 114174. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2020.114174. URL: https://www.sciencedirect.com/science/article/pii/S0957417420309118.

[11] H. F. Rahman, R. K. Chakrabortty, and M. J. Ryan. "Scheduling project with stochastic durations and time-varying resource requests: A metaheuristic approach". In: *Computers & Industrial Engineering* 157 (2021). DOI: https://doi.org/10.1016/j.cie.2021.107363. URL: https://www.sciencedirect.com/science/article/pii/S0360835221002679.

[12] B. Dutertre and L. de Moura. *The Yices SMT solver*. Tech. rep. Computer Science Laboratory, SRI International, 2006. URL: https://yices.csl.sri.com/papers/tool-paper.pdf.

[13] S. A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[14] N. Eén and N. Sörensson. "An extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing*. Ed. by E. Giunchiglia and A. Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518. ISBN: 978-3-540-24605-3.

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

[16] I. Abío et al. "A new look at BDDs for pseudo-Boolean constraints". In: *Journal of Artificial Intelligence Research* 45 (2012), pp. 443–480.

[17] R. Kolisch and A. Sprecher. "PSPLIB-a project scheduling problem library". In: *European Journal of Operational Research* 96 (1996), pp. 205–216.