A SAT based exact method to the RCPSP/Max

Guilherme Henrique Ismael de Azevedo
Universidade Federal Fluminense
Rua Passo da Pátria, 156, sala 309 - Bloco D, São Domingos, Niterói - RJ
guilhermehen@hotmail.com

Artur Alves Pessoa
Universidade Federal Fluminense
Rua Passo da Pátria, 156, sala 309 - Bloco D, São Domingos, Niterói - RJ
artur@producao.uff.br

ABSTRACT

In this article, we present an exact method to solve RCPSP/Max and its particular cases like RCPSP. Our method approach the problem by a relaxation of RCPCP/Max to the satisfiability Problem (SAT) to find feasible solutions and prove optimality. We also consider a workload approach to reduce the domain of decision variables and, as a consequence, reduce the SAT relaxations. Our method was tested with RCPSP/Max benchmark instances with 10 to 500 activities and 5 resources and with RCPSP benchmark instances with 30 to 120 activities and 4 resources. In total were considered 4470 instances and we solved 126 of previously unsolved instances in up to 600s. Our method also has better time solving relation than the best known methods.

KEYWORDS. Project scheduling. Resource constraints. Satisfiability problem.

AD & GP - OR in Administration & Production Management, OC - Combinatorial Optimization

1. Introduction

Optimizing project planning is important in every knowledge area since it can avoid waste of time and resources, specially in great projects like rocket launch, building hydro-electrical plants or oil and gas platforms [Pinedo, 2004]. The resource constrained project scheduling problem (RCPSP) is a well-known model for such optimization that is defined as follows. Let $V = \{0, 1, \ldots, N, N+1\}$ be the set of activities to be scheduled using a set $\mathscr{R} = \{1, \ldots, m\}$ of resources during a time horizon $[0, T)$. Each activity $j \in V$ has an execution time p_j and uses a specific amount r_{ij} of resource i at every instant of its processing time. Resource $i \in \mathscr{R}$ has R_i units available during all the time horizon. The schedule should also respect activity precedences, which are represented by a directed graph $G = (V, A)$. For each $(j, l) \in A$, we use d_{jl} to denote the time lag from activity j to l. In this case, l must start at least d_{jl} time units after j starts and j has to start at most $-d_{jl}$ units of time after l starts. For the classic RCPSP, only strict precedences are allowed, ie, if $(j, l) \in A$ then $d_{jl} = p_j$. For the RCPSP/max, $-\infty \leq d_{jl} \leq \infty$ and non-positive cycles are allowed in A. Activities 0 and $N+1$ represent the beginning and the end of the project, having null execution times and null use of resources. Every activity with no other predecessor will succeed 0, and every activity with no other successor will precede $N+1$. In this study, the objective is to find the minimum completion time for the project, also known as *makespan* and also defined as the finish time of activity $N+1$.

RCPSP and its variations are NP-hard as it generalizes shop problems, like open shop or job shop [Blazewicz et al., 1983]. Bartusch et al. [1988] show that even proving an instance is feasible is NP-hard for RCPSP/Max variation.

In this study, we present an exact method to solve RCPSP/Max and its particular cases like RCPSP. Our method approach the problem by a relaxation of RCPCP/Max to the Satisfiability Problem (SAT) to find feasible solutions and prove optimality. We also consider Azevedo e Pessoa [2012] workload approach to reduce the domain of decision variables and, as a consequence, reduce the SAT relaxations.

Our method was tested with the ProGen/max benchmark instances for RCPSP/Max with 10 to 500 activities and 5 resources. Those instances were developed by Schwindt [1995] and are public available in PSP, a public benchmark repository. Before this study, 107 of these instances were open, i.e., they had no known or proven optimal solution. We could find and prove optimality for 55 of these instances with the time limit of 600s.

We also tested it on benchmark instances for RCPSP with 30 to 120 activities and 4 resources. We could close 71 of the 495 previously open instances of this benchmarks.

This study is divided as follows: in section 2 we present a literature review on exact methods to solve resource constrained scheduling and on SAT solving techniques. The exact method proposed in this study is presented in 3 and the results of our computational experiment are presented in 4. Last our conclusions and future work are presented in 5.

2. Theoretical Framework / Literature Review

In this section, we present a literature review on exact methods to solve project scheduling problems and present some base studies to our method. Next we present a theoretical framework on Satisfiability problem and solving algorithm.

Pinedo [2004] presents a Mixed Integer Programming (MIP) formulation for the RCPSP. de Lemos evaluates the use of three MIP formulations and the impacts of adding two cutting planes types during the solution procedure.

Horbach [2010] presents a SAT formulation for the RCPSP with a custom process to include resource constraint clauses whenever they are violated by a partial allocation.

Due to the complexity of the problem, calculate lower bounds on the *makespan* can be useful in the decision-making process regarding the project [Azevedo e Pessoa, 2012].

A workload relaxation procedure to reduce the execution intervals and calculate a lower bound on the *makespan* was presented in Azevedo e Pessoa [2012].

According to Schutt et al. [2013], the best exact method known to solve the RCPSP was presented in Schutt et al. [2011] that considers Constraint Programming (CP) techniques. That article considers Lazy Clause Generation (LCG) to represent the domain of decision variables and included disjunction clauses for pairs of activities that can not be executed in parallel. A custom propagator called *cumulative*, deeply studied in Schutt et al. [2009], was used to represent resource constraints. Schutt et al. [2013] generalizes that method to the RCPSP/Max variation. As described in those works, they used a CP platform that includes a SAT Solver which can learn about the problem during its resolution, reducing search space. No better results were found in the literature for RCPSP and RCPSP/Max benchmark instances.

As shown above, there are different approaches to solve RCPSP and RCPSP/Max. In this article, we developed a SAT relaxation to find and prove the optimality of solutions. In 2.1, we discuss SAT characteristics and resolution methods. The RCPSP and RCPSP/Max base methods for our approach are respectively presented in 2.2 and 2.3.

2.1. Satisfiability Problem - SAT

The SAT is composed of binary variables and a propositional formula. This formula is in the Conjunctive Normal Form (CNF) if it is a set ("and","$\wedge$") of clauses, which are disjunctions ("or", "$\vee$") of literals. A literal can be a variable or its negation ("not", "$\neg$"). A formula is *satisfiable* if it is possible to find an assignment that makes the formula evaluate *true*.

The basic algorithm to solve SAT is known as Davis-Putnam-Logemann-Loveland (DPLL) and was proposed by Davis e Putnam [1960]. Marques-Silva e Sakallah [1999] presented the Conflict-Driven Clause Learning (CDCL) which was a major improvement on SAT solving methods. Once a conflict is found, it is analyzed to identify its causes and a new clause is created. In this case, backtracking is performed to the level where the new clause will be considered to propagate fixations, reducing the search space for solutions.

2.2. RCPSP reduction to SAT

In this section, we present the RCPSP relaxation to SAT presented by Horbach [2010]. This procedure requires to calculate a valid execution interval ($[ES_j, LF_j)$) for each activity ($j \in V$), that can only start on its early-start time (ES_j) or later and has to be completely executed before its late-finish time (LF_j). Once those intervals are calculated, Horbach [2010] considers two sets of variables s_{jt} and u_{jt}. Variable s_{jt} is *true* if $j \in V$ starts at instant $t \in \{ES_j, \ldots, LS_j\}$ and u_{jt} is *true* if $j \in V$ is being executed at instant $t \in \{ES_j, \ldots, (LF_j - 1)\}$. Horbach [2010] formulation is presented below:

$$\neg s_{jt_1} \vee u_{jt_2} \qquad \begin{array}{l} t_2 \in \{t_1, \ldots, t_1 + p_j - 1\}\,; \\ t_1 \in \{ES_j, \ldots, LS_j\}\,; \forall j \in V \end{array} \tag{1}$$

$$\neg s_{jt_1} \vee \bigvee_{t_2 \in \{\max\{t_1 + p_j; ES_l\}, \ldots, LS_l\}} s_{lt_2} \qquad \forall\,(j,l) \in A; t_1 \in \{ES_j, \ldots, LS_j\} \tag{2}$$

$$\bigvee_{t \in \{ES_j, \ldots, LS_j\}} s_{jt} \qquad \forall j \in V \tag{3}$$

$$\bigvee_{j \in C} \neg u_{jt} \qquad t \in \{0, \ldots, T-1\}\,; C \in \mathscr{C} \tag{4}$$

Clauses defined by (1) guarantee the correct relation between the start-time variables and processing variables. The precedence relations between activities are included by clauses (2). Clauses (3) define that all activities have to start. The resource constraints are defined by (4).

The set $C \subset V$ considered in (4) is such that $\sum_{j \in C} r_{ij} > R_i$ for a resource $i \in \mathscr{R}$ and $\mathscr{C}$ is the set of all sets with the described characteristic. As $\mathscr{C} \subset 2^V$, Horbach [2010] does not considers a promising approach to include all resource constraint clauses. Instead, those clauses are included on demand in a custom procedure similar to cut planes generation for Mixed Integer Programing.

2.3. Exact solving methods for RCPSP/Max

In this section we present two exact approaches to solve RCPSP/Max: the workload-based lower bound introduced by Azevedo e Pessoa [2012] and the Constraint Programming (CP) approach by Schutt et al. [2013].

2.3.1. Workload-based lower bound

Azevedo e Pessoa [2012] present a workload-based procedure to update activities execution intervals. This procedure keeps an execution interval ($[ES_j, LF_j)$) for each activity ($j \in V$), that can only start on its early-start time (ES_j) or later and has to be completely executed before its late-finish time (LF_j). For activity $N+1$, that represents the project endpoint, the ES represents a lower bound and LF represents an upper bound on the optimal *makespan*.

The procedure initializes the execution interval of $j \in V$ as $[ES_j, LF_j) = [0, T)$, where T is a valid upper bound on the *makespan*. Next, the intervals are checked and updated in order to guarantee precedence relations. This can be done with a procedure similar to the Critical Path Method (CPM) [Kelley Jr, 1961], i.e, for $(j,l) \in A$ if the constraint $ES_j + d_{jl} \leq ES_l$ is violated then ES_l is updated to $ES_j + d_{jl}$ and if the constraint $LF_l - d_{jl} \geq LF_j$ is violated then LF_j is updated to $LF_l - d_{jl}$. This procedure is repeated until no constraint is violated or until $LF_j - ES_j < p_j$ for a $j \in V$ which means that there is no solution with *makespan* less than or equal to T. Then it considers a workload-based relaxation of RCPSP/Max to check if all workload that must be scheduled before (or after) $j \in V$ starts (or finishes) fits in $[0, ES_j)$ (or in $[LF_j, T)$). If that workload does not fit in the interval, than ES_j is improved (or LF_j is reduced) by one unit and the precedence relations are considered to propagate this update to other activities.

To check if all workload that must be scheduled in $[0, ES_j)$ (or in $[LF_j, T)$), first Azevedo e Pessoa [2012] temporarily updates all activities execution intervals regarding j starts at ES_j (or finishes at LF_j). Next, they calculate the part of each activity that must be scheduled in the test interval under this temporary update. For each resource of the RCPSP/Max, the workload of each activity is scheduled by a greedy method considering the temporary intervals and the workforce available, but not precedence relations.

2.3.2. Constraint Programming formulation

Schutt et al. [2013] present a CP formulation for the RCPSP/Max that considers *cumulative* propagators, Lazy Clause Generations (LCG) and disjunctive clauses.

Schutt et al. [2009] study the use of *cumulative* propagator and describe it as equivalent to resource constraints. This procedure keeps a time table with the cumulative use of resources for each instant of time regarding the execution intervals of all activities. Thus, for a partial solution if it is not possible for an unfixed activity j to be executed in a specific time t due the cumulative use of a resource then the propagator will remove this instant from the domain of the execution interval of j.

According to Schutt et al. [2013], when one include a LCG on an integer variable x whose domain is $\{l, \ldots, u\}$ it implicitly creates two sets of binary variables $[x = t]$ and $[x \leq t]$ for each $t \in \{l, \ldots, u\}$. It also will include on demand the required constraints to guarantee that if $x = t$ then $[x = t]$ is *true* and if $x \leq t$ then $[x \leq t]$ is be *true*. Schutt et al. [2013] applies LCG in the integer variables S_j that define the start time of activities.

The disjunctive clauses are included for each pair of activities (j, l) that can not be executed in parallel due to resource consumption in such a way that or j precedes l or l precedes j. Schutt et al. [2013] include this redundant constraint believing it can help the solver to quicker determine or propagate information about start time variables.

3. Proposed method

In this section, we present the method proposed in this article to solve the RCPSP/Max and its particular cases. This method addresses the problem in two ways: destructive and constructive.

In the destructive approach, we reduce the feasibility RCPSP/Max to SAT, considering a hypothetical upper bound (T'). If the reduction is feasible/*satisfiable* and a solution is found, then T' (or the solution's *makespan*) will be an upper bound for the optimal *makespan*. On the other hand, if the reduction is infeasible/*unsatisfiable*, we proved that there is no solution for RCPSP/Max with *makespan* less than or equal to T', in other words $T' + 1$ will be a lower bound for the optimal *makespan*. To find the optimal solution and prove its optimality, we use a procedure similar to binary search on the value T'.

The constructive part of our method considers the workload-based procedure proposed in Azevedo e Pessoa [2012] to update the execution intervals of all activities and the lower bound on the *makespan*. When we perform this procedure and update execution intervals, we can create smaller size SAT relaxations that should be faster to generate and to solve. By updating the lower bound, we reduce the search interval on the value of T'.

Before performing both parts described above, we pre-process the input data in order to check instances feasibility and to improve precedence relations.

In 3.1, we present the pre-processing method here proposed. Section 3.2 presents our RCPSP/Max reduction to SAT. The workload-based procedure considered to update activities execution interval and its integration to SAT reduction are presented in 3.3.

3.1. Pre-processing

The proposed pre-processing is divided in two parts and its main goals are to check instance feasibility and improve the precedence relations graph, which may lead to a better trivial lower bound and better interval update propagations.

In the first part, we check resource infeasibility, i.e, if there is $r_{ij} > R_i$ for any resource $i \in \mathscr{R}$ and for any activity $j \in V$. In this case, j can not be scheduled since it requires more units of resource i than available on any time, then instance is infeasible.

In the second part, we check for positive cycles in the precedence graph. We consider the indirect precedence between activities, that is, if activity j precedes l which precedes k, then l and k are indirect successors of j. Formally, let the directed graph $G' = (V, A')$ represents the indirect precedences. For each, $(j, l) \in A'$ we use d'_{jl} to denote the time lag from activity j to l. G' is such that $A' \supset A$ and if $(j, l) \in A'$ then $(j, k) \in A'$ with related $d'_{jk} \geq d'_{jl} + d'_{lk}$. If $d'_{jk} + d'_{kl} > 0$ then there is a positive cycle in A.

Note that, even if no resource infeasibility and no positive cycle were found, the instance may still be infeasible when we consider resources and precedences together. For example, if $d'_{jl} + d'_{lj} = 0$ for $(j, l), (l, j) \in A'$ then j and l must start at the same time. Also consider that we have $r_{ji} < R_i$, $r_{li} < R_i$ and $R_i < r_{ji} + r_{li}$ for a resource i, which implies that j and l can not be scheduled in parallel. Considering both situations, as it is not possible for a pair of activities start at the same time and not be scheduled in parallel, then the problem is infeasible.

In this study, we propose to include some resource information in the precedence network. When the precedence relation between two activities allows them to be scheduled in parallel, we check if the resource constraints also allow it. If not, we update the indirect precedence and add a new precedence. Formally, if $(j, l) \in A'$ with $-p_l < d'_{jl} < p_j$ and there is i such that $R_i < r_{ji} + r_{li}$, then we can update d'_{jl} to p_j and add a new direct precedence from j to l with $d_{jl} = p_j$. Note that even for feasible instances, the new direct precedence added is useful since it benefits the propagations using precedence relations and may improve the lower bound on the *makespan*.

In the example described above, the proposed procedure would update d'_{jl} to p_j and d'_{lj} to p_l, leading to a positive cycle in G' and proving the infeasibility.

3.2. SAT formulation for RCPSP/Max

In this section, we present SAT formulation for the feasibility RCPSP/Max. Our formulation is based on the SAT formulation of Horbach [2010] and the CP formulation from Schutt et al. [2013], but we also introduced some new ideas, we beleave can benefit SAT solver.

In this formulation, we consider the sets of binary variables $[x_{jt}]$ and $[s_j \leq t]$. $[s_j \leq t]$ is *true* if activity $j \in V$ starts at instant $t \in [ES_j, LF_j)$ and $[x_{jt}]$ is *true* if activity $j \in V$ is being executed at instant $t \in [ES_j, LF_j)$. Note that the exact start time of activity j can be computed by expression $[s_j \leq t] \wedge \neg [s_j \leq (t-1)]$ after solve the sat relaxation.

$$\neg [s_j \leq t] \vee [s_j \leq (t+1)] \qquad \forall t; ES_j \leq t < LS_j \tag{5}$$
$$[s_j \leq LS_j] \qquad \forall j \in V \tag{6}$$
$$\neg [s_j \leq t] \vee [s_k \leq (t - d_{kj})] \qquad \begin{array}{l} \forall j \in V; \forall (k,j) \in A; \\ \forall t | ES_j \leq t \leq LS_j \wedge ES_k \leq (t - d_{kj}) \leq LS_k \end{array} \tag{7}$$
$$\neg [s_j \leq t_1] \vee [s_j \leq (t_1 - 1)] \vee [x_{jt_2}] \qquad \begin{array}{l} \forall j \in V; ES_j < t_1 \leq LS_j; \\ t_1 \leq t_2 \leq (t_1 + p_j - 1) \end{array} \tag{8}$$
$$\neg [s_j \leq t_1] \vee [x_{jt_2}] \qquad \begin{array}{l} \forall j \in V; t_1 = ES_j; \\ t_1 \leq t_2 \leq (t_1 + p_j - 1) \end{array} \tag{9}$$
$$\neg [x_{jt}] \vee [s_j \leq t] \qquad \forall j \in V; ES_j \leq t \leq LS_j \tag{10}$$
$$\neg [x_{jt}] \vee \neg [s_j \leq (t - p_j)] \qquad \forall j \in V; ES_j \leq t \leq LS_j \tag{11}$$
$$\bigvee_{j \in \lambda_{ki}} \neg [x_{jt}] \qquad \forall t; \forall k; \forall \lambda_{kt} \tag{12}$$

The set of clauses defined by (5) guarantees the relation between variables $[s_j \leq t]$ and their interpretation. (6) ensures all activities are executed. The precedence relations are defined by (7). Clauses (8), (9), (10) and (11) insure the relation between $[s_j \leq t]$ and $[x_{jt}]$ so that $[x_{jt_1}]$ is *true* if, and only if, a variable $[s_j \leq t_2]$ is also *true* for $t_2 \in [(t_1 - p_j + 1), (t_1 + p_j - 1)]$. Resource constraints are defined by (12).

In (12), $\lambda_{ti} \subseteq V_t$ represents a set of activities that can not be executed all at the same time $t \in [0, T'[$, that is, at least one activity $j \in \lambda_{ti}$ can not be executed at time t. In the worst case, we could have 2^N subsets with this characteristics, however, unnecessary clauses would be generated. Note that if $\lambda_t \subset \lambda_t^{'}$ then $\lambda_t^{'}$ would generate a clause that is a perfect superset of the clause generated by λ_t, which means that the $\lambda_t^{'}$ is not needed. In other words, only minimal subsets of activities that can not be executed in parallel are considered.

Even considering only minimal $\lambda_{tj} \subset V_t$, the quantity of clauses (12) could be very large and even identify the minimal λ_{tj} could be too time consuming. For this reason, in this article we proposed to include only clauses related to λ_{ti} such that $2 \leq |\lambda_{ti}| \leq 3$. The remaining clauses may be included on demand in two different procedures: inside the

SAT Solver, as in Horbach [2010] or after the resolution of the SAT, which requires a new solution round. Both ways were implemented and tested and are compared in this work.

To create the feasibility RCPSP/Max relaxation to SAT, one must define a hypothetical upper bound (T') on the *makesan* to test and then calculate the execution intervals $[ES_j, LF_j)$ for all $j \in V$. If the SAT relaxation is *satisfiable*, then we found a feasible solution for the RCPSP/Max and T' (or the *makespan* of the solution found) is an upper bound. Other wise, we proved that there is no feasible solution for RCPSP/Max with *makespan* equal to or less than T', then $T' + 1$ is a new lower bound. To find solutions and prove optimality, in this article we use an exponential search, a procedure similar to binary search, on the value of T'.

In 3.3, we present a workload-based method to update execution intervals of each activity and, as consequence, we may obtain a better lower bound. This procedure contributes reducing the size of SAT relaxations, as the quantity of variables and clauses may reduce, and the search space for the optimal solution on the value of T'.

3.3. Execution intervals improvement by workload-based relaxation

In this section, we present the way we consider the workload-based procedure to update activities execution intervals introduced by Azevedo e Pessoa [2012] and how we integrated it to SAT. In that work, this procedure was used to calculate lower bounds on the *makespan*. In this article, We use this procedure in order to generate a smaller size RCPSP/Max reduction to SAT.

That procedure requires a valid upper bound to initialize the execution intervals. In this article we consider $T = \sum_{j\in V} \left(\max_{(j,l)\in A} \{d_{jl}, p_j\}\right)$.

While developing this study, we noticed that only the SAT could solve a great number of RCPSP/Max and RCPSP benchmark instances in a few seconds but the workload-based interval update could help solve some harder instances. In order to try to obtain the best of both approaches, we decided to start by solving SAT relaxation. If after 3.5s solving a single SAT relaxation it is not solved yet (*satisfiable* with a solution or *unsatisfiable*), we interrupt the solver and start the interval update procedure.

In the first time the interval update is required, we consider the current upper bound on the *makespan* to initialize the intervals. The improved intervals under this consideration can be held to any $T' \leq T$ under test, so we avoid repeating some updates. After that we regard the improved execution intervals and the current T' in test to further improve the intervals, which will only be valid under this consideration. The updated intervals are transfered to SAT relaxation by fixing the value of variables $[s_j \leq t]$. Next SAT relaxations will already have fewer variables and clauses since we already have valid improved intervals, but we can further improve intervals by updating them for the new T' under test. If after an interval update a T' under test is feasible a upper bound on the *makespan*, the improved intervals is held for the next tests.

Note that if a variable $[s_j \leq t]$ is fixed as *true* by the SAT solver before the interruption we can update LF_j to t and if this variable is fixed to *false* we can update ES_j to $t + 1$. This way we can also transfer some information from SAT relaxation to the workload-based interval updates.

4. Experiments

Our procedure were implemented in *C++* and compiled by GCC for Linux. In order to solve SAT relaxations we incorporated MiniSAT 2.2.0 [Eén e Sörensson, 2004] to our procedure. All tests were performed on a computer with Core i7 processor and 12 GB RAM and Ubuntu 14.04.3 LTS Operating System. We considered the time limit of 600s, the same as Schutt et al. [2013].

We have considered a total of 4470 benchmark instances, all publicly available in PSP. For RCPSP/Max we have considered 2430 instances of groups *UBO10*, *UBO20*, *UBO50*, *UBO100*, *UBO200*, *UBO500*, *j10*, *j20*, *j30*, *TestSetC* and *TestSetD*, respectively with 10, 20, 50, 100, 200, 500, 10, 20, 30, 100 e 100 activities. For the RCPSP, we have considered 2040 of groups *j30*, *j60*, *j90* e *j120*, respectively with 30, 60, 90 and 120 activities.

Besides generalized precedences, there is a major difference between RCPSP and RCPSP/Max benchmark instances: in general, RCPSP instances have more resources slack, which allow more than two activities to be processed in parallel. For RCPSP/Max benchmark instances, resources are tighter.

We have implemented four configurations of our method to allow a better understanding on the benefits of each mechanism. All configurations consider the pre-process here presented.

- ***Conf A***: runs only the SAT relaxation and resource clauses related to $|\lambda_{ti}| > 3$ are added on demand after solving;
- ***Conf B***: runs only the SAT relaxation and resource clauses related to $|\lambda_{ti}| > 3$ are added inside the SAT solver by a custom procedure;
- ***Conf C***: runs SAT relaxation and workload-based interval updates and resource clauses related to $|\lambda_{ti}| > 3$ are added on demand after solving;
- ***Conf D***: runs SAT relaxation and workload-based interval updates and resource clauses related to $|\lambda_{ti}| > 3$ are added inside the SAT solver by a custom procedure;

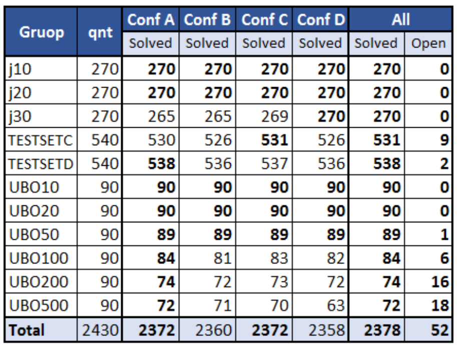
First, the four configurations of our method are compared among themselves and then to with the best known results. Since there are important differences between RCPSP and RCPSP/Max benchmark instances, we present results and comments apart.

4.1. Configuration comparison

For RCPSP/Max benchmark instances, all configurations of our method proved infeasibility for the 362 instances known to be infeasible. For 360 of those, infeasibility was proved by the pre-processing proposed in this article and for the other 2 instances SAT relaxation proved infeasibility in less than 40s. No new infeasibility was found.

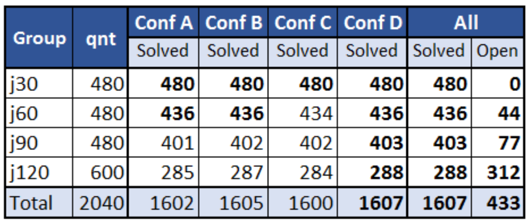
Tables 1 and 2 compare the number of instances solved before the time limit of 600s, respectively for RCPSP/Max and for RCPSP benchmark instances. First two columns presents the group of instances and the total number benchmark instances in the group. Following columns present results for the four configurations and the results considering all configurations. *Solved* presents the number of instances whose infeasibility were proven and feasible instances whose optimal solution was found and proved and *Open* presents the number of not solved.

Table 1: Configuration comparison for RCPSP/Max instances



Gruop	qnt	Conf A	Conf B	Conf C	Conf D	All	
		Solved	Solved	Solved	Solved	Solved	Open
j10	270	**270**	**270**	**270**	**270**	**270**	**0**
j20	270	**270**	**270**	**270**	**270**	**270**	**0**
j30	270	265	265	269	**270**	**270**	**0**
TESTSETC	540	530	526	**531**	526	**531**	**9**
TESTSETD	540	**538**	536	537	536	**538**	**2**
UBO10	90	**90**	**90**	**90**	**90**	**90**	**0**
UBO20	90	**90**	**90**	**90**	**90**	**90**	**0**
UBO50	90	**89**	**89**	**89**	**89**	**89**	**1**
UBO100	90	**84**	81	83	82	**84**	**6**
UBO200	90	**74**	72	73	72	**74**	**16**
UBO500	90	**72**	71	70	63	**72**	**18**
Total	2430	**2372**	2360	**2372**	2358	**2378**	**52**

Table 2: Configuration comparison for RCPSP instances



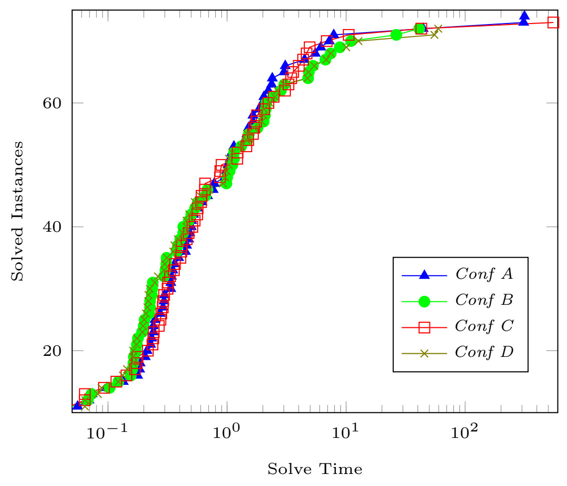
Group	qnt	Conf A	Conf B	Conf C	Conf D	All	
		Solved	Solved	Solved	Solved	Solved	Open
j30	480	**480**	**480**	**480**	**480**	**480**	**0**
j60	480	**436**	**436**	434	**436**	**436**	**44**
j90	480	401	402	402	**403**	**403**	**77**
j120	600	285	287	284	**288**	**288**	**312**
Total	2040	1602	1605	1600	**1607**	**1607**	**433**

In Table 1, it can be seen that only 52 RCPSP/Max instances were not solved by any of the presented configurations in 600s. None of the settings dominates the comparative, ie, no configuration has the best results for all groups under comparison. Only *Conf D* solves all instances in *j30*, but for *UBO500* it can solve much less instances. For groups with smaller instances (*j10*, *j20*, *UBO10* and *UBO20*), all of them were solved by all configurations. For greater instances (*UBO200* and *UBO500*), *Conf A* had a better performance. This result can also be observed for *TestSetD* and *UBO100*.

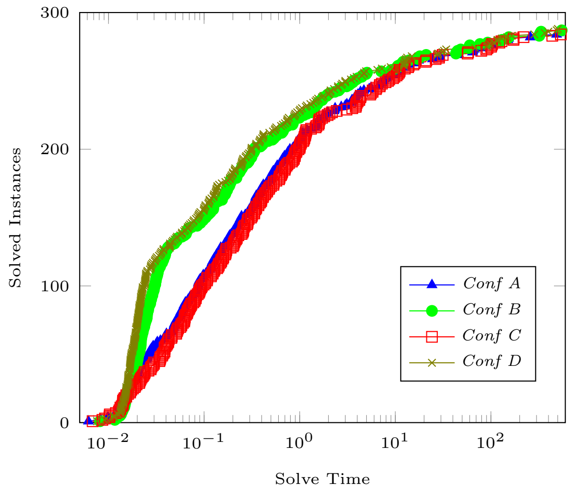
In Table 2, it can be seen that 433 RCPSP instances have not been resolved in up to 600s and that 312 of them are from group *j120*. *Conf D* dominates the comparative, i.e., it has the best performance for all groups and in general. This configuration was able to solve all instances also resolved by the other settings. The worst result was obtained by *Conf C*.

In order to compare the solving time for the different configurations of our methods, we present graphics by groups of instances with the number of instances solved by solving time. The x-axis presents the solving time and the y-axis presents the number of instances of the group that were solved with solving time less than or equal to that time. The scales of the graphs were adjusted according to the situation of each group to allow better comparative visualization. We consider group *UBO200* for RCPSP/Max and group *j120* for RCPSP. Other groups have a similar behavior.

Configuration comparison - Group *UBO200*



Configuration comparison - RCPSP Group *j120*



Generally considering all RCPSP/Max groups of instances, the four configurations solve large number of instances in a few instants, but from a certain point, there is greater dispersion for the time required to solve new instances. The four configurations have very similar curves, but the curve of *Conf B* is is below the other. Besides that, the majority of instances on groups *j10*, *j20*, *UBO10* and *UBO20* are solved in less than a second by all four configurations.

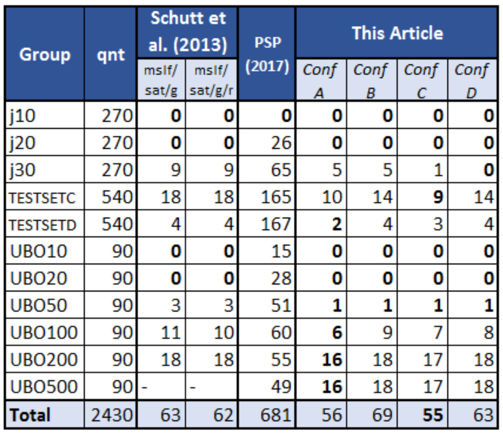
In the chart for group *j120*, one can observe that *Conf D* and *Conf B* present better performance until about 10s. Before 10s, the curve of *Conf D* is above the curve of *Conf B*. From this point forward, the curves for all configurations get closer. We have a similar behavior for RCPSP group *j90*, but curves get closer after 0.5s. At the beginning of the curves for RCPSP group *j60*, we have a behavior similar the observed for *j120*. However, after 0.05s and before 50s, *Conf A* and *Conf C* have better performance. After 50s, all curves get closer. Considering the four groups of RCPSP benchmark instances, all configurations are able to solve a large number of instances in a few seconds. For RCPSP group *j30*, all instances are resolved in just over 10s.

4.2. Literature comparison

In this section, we compare the results of the method here proposed to the literature best know results. As far as we are concerned, the best known results for RCPSP benchmark instances were found in Schutt et al. [2011] and for RCPSP/Max benchmark instances were found in Schutt et al. [2013]. For RCPSP/Max benchmark instances, the individual solving time on 6 configurations are available at `http://people.eng.unimelb.edu.au/pstuckey/rcpsp/rcpspmax_all.html`, which allowed us to compare the solving times by instances on the best configurations. On the other hand, Schutt et al. [2013] have not considered group *UBO500*, so we also consider PSP results. For RCPSP benchmark instances, we considered only PSP results, since no other public data with individual result was found.

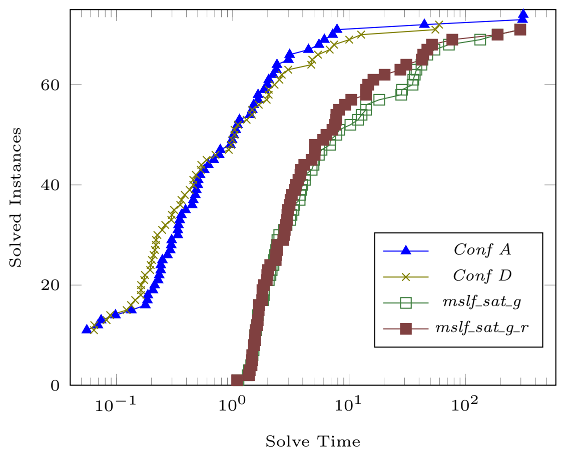
Table 3 compares the number of RCPSP/Max instances not solved before the time limit of 600s by our method, by Schutt et al. [2013] configurations *mslf/sat/g* and *mslf/sat/g/r* and by other results available at PSP.

Table 3: Literature comparison for RCPSP/Max not solved instances



Group	qnt	Schutt et al. (2013)		PSP (2017)	This Article			
		mslf/ sat/g	mslf/ sat/g/r		*Conf A*	*Conf B*	*Conf C*	*Conf D*
j10	270	**0**	**0**	**0**	**0**	**0**	**0**	**0**
j20	270	**0**	**0**	26	**0**	**0**	**0**	**0**
j30	270	9	9	65	5	5	1	**0**
TESTSETC	540	18	18	165	10	14	**9**	14
TESTSETD	540	4	4	167	**2**	4	3	4
UBO10	90	**0**	**0**	15	**0**	**0**	**0**	**0**
UBO20	90	**0**	**0**	28	**0**	**0**	**0**	**0**
UBO50	90	3	3	51	**1**	**1**	**1**	**1**
UBO100	90	11	10	60	**6**	9	7	8
UBO200	90	18	18	55	**16**	18	17	18
UBO500	90	-	-	49	**16**	18	17	18
Total	2430	63	62	681	56	69	**55**	63

Literature comparison - Group *UBO200*



One can observe in Table 3 that the settings of Schutt et al. [2013] did not obtain better results than the configurations of the method here proposed for any of the groups. Our method solved all instances solved by Schutt et al. [2013]. For group *UBO500*, all of our configurations solved all instances already solved in the Literature and, at least, 31 more instances.

In order to compare the solving time of the configurations of our methods and the method introduced by Schutt et al. [2013], we present a chart for group *UBO200* of RCPSP/Max benchmark instances. We have chosen *Conf A* and *Conf D* of proposed method to compare to the literature results. *Conf A* is the simpler configuration and had better solving time for *UBO100* and *UBO200*. *Conf D* was the only configuration to solve all *j30* instances and had also solved more instances of RCPSP. Generally considering all groups of instances, our configurations solve large number of instances in a few instants, as their curves are the majority of time above curves of Schutt et al. [2013] methods.

Table 4 presents our results for the previously unsolved RCPSP/Max benchmark instances, considering results from Schutt et al. [2013] and PSP. The first three columns show the group of instances, total quantity of instances and the unsolved instances in the group. The next five columns present the number of previously unsolved instances that were solved by our four configurations and by at least one of them.

This results show that all four configurations were able to reduce the total of unsolved instances. Considering all configurations, we solved 51,4% of these instances and

Table 4: Literature comparison for open RCPSP/Max instances

Grupo	Total	Unsolved	*Conf A*	*Conf B*	*Conf C*	*Conf D*	Any
j10	270	0	-	-	-	-	-
j20	270	0	-	-	-	-	-
j30	270	6	1	1	5	**6**	**6**
TESTSETC	540	17	7	3	**8**	4	**8**
TESTSETD	540	4	**2**	0	1	0	**2**
UBO10	90	0	-	-	-	-	-
UBO20	90	0	-	-	-	-	-
UBO50	90	3	**2**	**2**	**2**	**2**	**2**
UBO100	90	10	**4**	1	3	2	**4**
UBO200	90	18	**2**	0	1	0	**2**
UBO500	90	49	**31**	30	29	22	**31**
Total	2430	107	**49**	37	**49**	36	**55**

even the configuration that less solves instances has solved 33,6% of them. For group *j30*, we could close all instances with *Conf D*.

In Table 5, we can compare the results of our four configurations to the best known literature results available at PSP for RCPSP benchmark instances.

Table 5: Literature comparison for RCPSP unsolved instances

Group	qnt	Literature		Conf A	Conf B	Conf C	Conf D
		Solved	Open	Open	Open	Open	Open
j30	480	480	**0**	**0**	**0**	**0**	**0**
j60	480	401	79	**44**	**44**	46	**44**
j90	480	375	105	79	78	78	**77**
j120	600	289	**311**	315	313	316	312
Total	2040	1545	495	438	435	440	**433**

Table 6: Literature comparison for previously unsolved RCPSP instances

Group	Total	Open	*Conf A*	*Conf B*	*Conf C*	*Conf D*	Any
j30	480	0	-	-	-	-	-
j60	480	79	**35**	**35**	33	**35**	**35**
j90	480	105	29	29	29	**31**	**31**
j120	600	311	4	**5**	3	**5**	**5**
Total	2040	495	68	69	65	**71**	**71**

All of our configurations solved 62 more instances than previously solved in the literature regarding all RCPSP benchmark groups, 35 more instances considering *j60* and 28 more instances for *j90*. However for *j120* the literature known results have one more solved instance than solved in this study. Regarding previously solved RCPSP benchmark instances, our method could not find and prove optimality of solutions for 9 instances up until 600s, 3 instances from group *j90* and 6 from *j120*.

Table 6 presents our results for the previously unsolved RCPSP benchmark instances, considering results available at PSP. Note that our method solved 71 those unsolved instances, which represents 14.3% of those instances. Group *j30* had no open instances. For all other groups, we reduced the number of unsolved instances with all four configurations.

5. Conclusion and Future Work

The results presented in this article show that the method here proposed can solve a large number of RCPSP and RCPSP/Max benchmark instances in few instants. Regarding the total number of instances solved and the solving time relation, our method had better results than best known methods. For RCPSP/Max instances, we could solve all previously solved instances and also 55 other, which represents 51% of unsolved instances. In case of RCPSP, we could not solve 9 previously solved instances, but we solved 71 previously unsolved instances.

Comparing the four configurations of our method, we found better results for greater RCPSP/Max instances ($N \geq 100$) without the custom resource propagator (*Conf A* and *Conf C*), though *Conf D* was the only to solve all *j30* instances. For RCPSP instances we can observe a different behavior, as configuration *Conf D* had the best results, regarding number of instances solved and solving time relation.

This difference in behavior may not be caused by the particularities of RCPSP, but by resource slacks. These results indicate that for instances with few slacks it is better not to interfere inside the SAT solver to add new resource constraint clauses as only few may be required. On the other hand, for instances with more resource availability not considering the custom propagator may require a very large number of resolving the SAT relaxation to add all required resource constraints. In future studies, this relationship could be better investigated.

Results also indicates that more resource slack instances tend to be more difficult for SAT-based model here presented, and probably for other exact methods. In the future, one can check other ways to model the resource constraints in order to avoid this issue.

References

Project scheduling problem library - psplib. URL `http://www.om-db.wi.tum.de/psplib/`.

Azevedo, G. H. e Pessoa, A. (2012). Improved lower bounds for hard project scheduling instances. In *Anais do Congreso Latino-Iberoamericano de Investigación Operativa & XLIV Simpósio Brasileiro de Pesquisa Operacional*, p. 124–135.

Bartusch, M., Möhring, R. H., e Radermacher, F. J. (1988). Scheduling project networks with resource constraints and time windows. *Annals of operations Research*, 16(1):199–240.

Blazewicz, J., Lenstra, J. K., e Kan, A. R. (1983). Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24.

Davis, M. e Putnam, H. (1960). A computing procedure for quantification theory. *J. ACM*, 7(3):201–215. ISSN 0004-5411. URL `http://doi.acm.org/10.1145/321033.321034`.

de Lemos, M. V. W. Escalonamento de projetos com restrição de recursos: formulações de programação inteira com aplicação de planos de corte.

Eén, N. e Sörensson, N. (2004). *An Extensible SAT-solver*, p. 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-24605-3.

Horbach, A. (2010). A boolean satisfiability approach toÂtheÂresource-constrained project scheduling problem. *Annals of Operations Research*, 181(1):89–107. ISSN 1572-9338.

Kelley Jr, J. E. (1961). Critical-path planning and scheduling: Mathematical basis. *Operations research*, 9(3):296–320.

Marques-Silva, J. P. e Sakallah, K. A. (1999). Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521.

Pinedo, M. L. (2004). Planning and scheduling in manufacturing and services. In *Project Planning and Sheduling*, chapter 4, p. 51–79. Springer, Nova Iorque.

Schutt, A., Feydy, T., Stuckey, P. J., e Wallace, M. G. (2011). Explaining the cumulative propagator. *Constraints*, 16(3):250–282. ISSN 1572-9354. URL `http://dx.doi.org/10.1007/s10601-010-9103-2`.

Schutt, A., Feydy, T., Stuckey, P. J., e Wallace, M. G. (2013). Solving rcpsp/max by lazy clause generation. *Journal of Scheduling*, 16(3):273–289. ISSN 1099-1425.

Schutt, A., Feydy, T., Stuckey, P., e Wallace, M. (2009). *Why cumulative decomposition is not as bad as it sounds*, p. 746 – 761. Springer Verlag, Germany. ISBN 9783642042430.

Schwindt, C. (1995). Progen/max: A new problem generator for different resource-constrained project scheduling problems with minimal and maximal time lags.