Journal of Scheduling manuscript No.
(will be inserted by the editor)

Solving RCPSP/max by Lazy Clause Generation

Andreas Schutt · Thibaut Feydy · Peter J. Stuckey · Mark G. Wallace

Received: date / Accepted: date

Abstract We present a generic exact method for minimizing the project duration of the resource-constrained project scheduling problem with generalized precedence relations (RCPSP/max). This is a very general scheduling model with applications areas such as project management and production planning. Our method uses lazy clause generation, *i.e.*, a hybrid of finite domain and Boolean satisfiability solving, in order to apply nogood learning and conflict-driven search to the solution generation. Our experiments show the benefit of lazy clause generation for finding an optimal solution and proving its optimality in comparison to other state-of-the-art exact and non-exact methods. In comparison to other methods, our method is able to find better solutions faster on the RCPSP/max benchmarks. Indeed our method closes 573 open problem instances and generates better solutions in most of the remaining instances. Surprisingly, although ours is an exact method, it outperforms the published non-exact methods on these benchmarks in terms of the quality of solutions.

Keywords Project scheduling · Resource constraints · Time windows · Generalized precedence constraints · Lazy clause generation · Constraint programming

A. Schutt, T. Feydy, P. J. Stuckey
National ICT Australia, Department of Computing and Information Systems, The University of Melbourne, Victoria 3010, Australia
E-mail: {andreas.schutt,thibaut.feydy,peter.stuckey}@nicta.com.au

M. G. Wallace
Faculty of Information Technology, Monash University, Caulfield, Victoria 3145, Australia
E-mail: mark.wallace@monash.edu

1 Introduction

The resource-constrained project scheduling problem with generalized precedence relations (RCPSP/max)[1] consists of scarce resources, activities and precedence constraints between pairs of activities. Each activity requires some units of resources during its execution. The aim is to build a schedule that obeys the resource and precedence constraints. Here, we concentrate on renewable resources (*i.e.*, their supply is constant during the planning period), non-preemptive activities (*i.e.*, once started their execution cannot be interrupted), and finding a schedule with a minimal project duration. This problem is denoted as $PS|temp|C_{max}$ by Brucker *et al.* (1999) and $m,1|gpr|C_{max}$ by Herroelen *et al.* (1998). Bartusch *et al.* (1988, Theorem 3.10) show that the feasibility problem, whether an instance is feasible given an unlimited project duration, is NP-hard.

RCPSP/max is a very general problem. Practical scheduling problems can include substantially varied restrictions on the resources and activities. The following restrictions can be modeled with generalized precedence relations: minimal and maximal overlaps of activities, synchronization of start or end times for activities, change of the resource requirement during the activity's execution, fixed start times of activities, setup times, or non-delay execution of activities (see, *e.g.* Bartusch *et al.*, 1988; Neumann and Schwindt, 1997; Dorndorf *et al.*, 2000). Moreover, a variation of the resource availability over time can be modeled by adding fictitious activities.

RCPSP/max is widely studied and some of its applications can be found in Bartusch *et al.* (1988), *e.g.* civil

[1] In the literature RCPSP/max is also called as RCPSP with temporal precedences, arbitrary precedences, minimal and maximal time lags, and time windows.

engineering, building projects, and processor scheduling. A problem instance consists of a set of resources, a set of activities, and a set of generalized precedence constraints between activities. Each resource is characterized by its integral capacity, and each activity by its integral duration and its resource requirements. Generalized precedence relations express relations of start-to-start, start-to-end, end-to-start, and end-to-end times between pairs of activities. All these relations can be formulated as start-to-start time precedences. They have the form $s_i + d_{ij} \leq s_j$ where s_i and s_j are the start times of the activities i and j, respectively, and d_{ij} is an integral distance between them. If $d_{ij} \geq 0$ this imposes a *minimal time lag*, while if $d_{ij} < 0$ this imposes a *maximal time lag* between start times.

Example 1 A simple example of an RCPSP/max problem consists of the five activities a, b, c, d, and e with start times s_a, s_b, s_c, s_d, and s_e, durations $2, 5, 3, 1$, and 2 and resource requirements on a single resource $3, 2, 1, 2$, and 2 with a resource capacity of 4. Suppose we also have the generalized precedence relations $s_a + 2 \leq s_b$ (activity a ends before activity b starts), $s_b + 1 \leq s_c$ (activity b starts at least 1 time unit before activity c starts), $s_c - 6 \leq s_a$ (activity c cannot start later than 6 time units after activity a starts), $s_d + 3 \leq s_e$ (activity d starts at least 3 time units before activity e starts), and $s_e - 3 \leq s_d$ (activity e cannot start later than 3 time units after activity d starts). Note that the last two precedence relations express the relation $s_d + 3 = s_e$ (activity d starts exactly 3 time units before activity e).

Let the maximal project duration, in which all activities must be completed, be 8. Figure 1 illustrates the precedence graph between the five tasks and source at the left (time 0) and sink at the right (time 8), as well as a potential solution to this problem in the Gantt chart, where a rectangle for activity i has width equal to its duration and height equal to its resource requirements.

Note that additional edges between the source (sink) are drawn in the precedence graph. These edges reflect the constraints that the activities must be executed in the planning period created by start time of the source and end time of the sink, which is $[0, 8)$. □

Standard benchmarks for RCPSP/max consist of a number of challenging testsets: SM, CD, and UBO; accessible from PSPLib (PSPLib, 2010). The first exact method we are aware of to tackle RCPSP/max was proposed by Bartusch *et al.* (1988). They use a branch-and-bound algorithm to tackle the problem. Their branching is based on resolving (minimal) conflict sets[2] by the addition of precedence constraints breaking these sets. Other branch-and-bound methods based on the same idea were developped later (*e.g.* De Reyck and Herroelen, 1998; Schwindt, 1998a; Fest *et al.*, 1999). The results from Schwindt are the best published ones for an exact method on the testset SM.

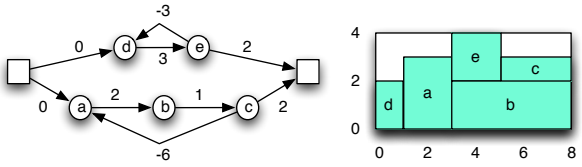


Fig. 1: The precedence graph and the Gantt chart of a solution to a small RCPSP/max.

Dorndorf *et al.* (2000) use a time-oriented branch-and-bound combined with constraint propagation for precedence and resource constraints. In every branch one unscheduled and "eligible" activity is selected and its start time is assigned to the earliest point in time that does not violate any constraint regarding the current partial schedule. On backtracking they apply dominance rules to fathom the search space. As far as we can determine, this exact approach outperforms other exact methods for RCPSP/max on the CD benchmark set.

Franck *et al.* (2001) compare different solution methods on the benchmark set UBO with instances having from 10 to 1000 activities. Their methods are truncated branch-and-bound algorithms, filter-beam search, heuristics with priority rules, genetic algorithms and tabu search. All methods share a preprocessing step to determine feasibility or infeasibility. The preprocessing step decomposes the precedence network into strongly connected components (SCCs) (which are denoted "cyclic structures" in Franck *et al.* (2001)). The preprocessing then determines a solution or infeasibility for each SCC individually using constraint propagation and a destructive lower bound computation. Once a solution for all SCCs is determined a first solution can be deterministically generated for the original instance; otherwise infeasibility is proven.

Ballestín *et al.* (2011) employ an evolutionary algorithm based on a serial generation scheme with an unscheduling step. Their crossover operator is based on so-called *conglomerates*, *i.e.*, set of cycle structures and other activities which cannot move freely inside a schedule. It tries to keep the "good" conglomerates of the parents for their children. This is the best published metaheuristic so far on the testsets UBO (up to instances with 100 activities) and CD.

[2] Conflict sets are set of activities for which their execution might overlap in time and violate at least one resource constraint if they are executed at the same time.

Cesta *et al.* (2002) propose a two layered heuristic that is based on a temporal precedence network and extension of this network by new temporal precedence relations in order to resolve minimal conflict sets. For guidance, constraint propagation algorithms are applied on the network. Their method is competitive on the benchmark set SM.

Oddi and Rasconi (2009) apply a generic iterative search consisting of a relaxation and flattening step based on temporal precedence relations which are used for resolving resource conflicts. In the first step, some of the temporal precedence relations are removed from the problem, and then, in the second step, others are added if a resource conflict exists. Their method is evaluated on some larger instances from UBO.

A special case of RCPSP/max is the intensively-studied Resource-constrained Project Scheduling Problem (RCPSP) where the precedence constraints $s_i + d_{ij} \leq s_j$ express that the activity j must start after the end of i, *i.e.*, d_{ij} equals to the duration of i. In contrast to RCPSP/max, the feasibility problem of RCPSP, *i.e.*, with no restriction on the project duration, is polynomial solvable, while both the decision and optimization problem are NP-hard (Blazewicz *et al.*, 1983). The reason for this is the absence of maximal time lags, *i.e.*, here activity executions can always be delayed to a point in time where enough resource units are available without breaking any precedence constraints. That is not possible for RCPSP/max.

To our knowledge, The best exact methods for RCPSP are the methods that use the advanced Boolean satisfiability (SAT) technology based on Davis-Putnam-Logemann-Loveland (DPLL) procedure (Schutt *et al.*, 2009, 2011; Horbach, 2010). The SAT technology enables them to take advantage of its nogood learning facilities in order to prune the search space. The methods (Schutt *et al.*, 2009, 2011) are generic and based on lazy clause generation (LCG) (Ohrimenko *et al.*, 2009) using the G12 Constraint Programming Platform (Stuckey *et al.*, 2005). LCG is a hybrid of a finite domain (FD) and a SAT solver. Their approaches model the cumulative resource constraint by either decomposing it into smaller primitive constraints, or creating a global cumulative propagator. The global propagation approach performs better as the size of the problem grows. In contrast, Horbach's approach is also based on a hybrid with SAT solving, but hand-tailored for RCPSP. He uses a linear programming solver to determine activity schedules and hybridize with the SAT solver. Overall, the global approach (Schutt *et al.*, 2011) gives the best current results for RCPSP.

In this paper, we apply the same generic LCG approach as in Schutt *et al.* (2011) to the more general problem of RCPSP/max. Because this problem is more difficult than pure RCPSP, we need to modify this approach, in particular to prove feasibility/infeasibility. We show that the approach to solving RCPSP/max performs better than published methods so far, especially for improving a solution once a solution is found, and proving optimality. We state the limitations of our current model and how to overcome them. We compare our approach to the best known approaches to RCPSP/max on several benchmark suites accessible via PSPLib (2010).

The paper is organized as follows. In Section 2 we give an introduction to LCG. In Section 3, we present our basic model for RCPSP/max and discuss some improvements to it. In Section 4, we discuss the various branch-and-bound procedures that we use to search for optimal solutions. In Section 5, we compare our algorithm to the best approaches we are aware of on 3 challenging benchmark suites. Finally, we conclude in Section 6.

2 Preliminaries

In this section, we explain LCG by first introducing FD propagation and DPLL-based SAT solving. Then we describe the hybrid approach. We discuss how the hybrid explains conflicts and briefly discuss how a cumulative propagator is extended to explain its propagations.

2.1 Finite Domain Propagation

Finite domain (FD) propagation (see, *e.g.* Tsang, 1993; Marriott and Stuckey, 1998) is a powerful approach to tackling combinatorial problems. An FD problem ($\mathcal{C}$,D) consists of a set of constraints $\mathcal{C}$ over a set of variables V, and a domain D, which determines the finite set of possible values of each variable in V. A *domain* D is a complete mapping from V to finite sets of integers. Hence given domain D, then $D(x)$ is the set of possible values that variable x can take. Let $\min_D(x) = \min(D(x))$ and $\max_D(x) = \max(D(x))$. Let $[\,l\,..\,u\,] = \{d \mid l \leq d \leq u, d \in \mathbb{Z}\}$ denote a *range* of integers, where $[\,l\,..\,u\,] = \emptyset$ if $l > u$. In this paper, we will concentrate on domains where $D(x)$ is a range for all $x \in V$. The *initial domain* is referred as D_{init}. Let D_1 and D_2 be domains, then D_1 is *stronger* than D_2, written $D_1 \sqsubseteq D_2$, if $D_1(v) \subseteq D_2(v)$ for all $v \in V$. Similarly, if $D_1 \sqsubseteq D_2$ then D_2 is *weaker* than D_1. For instance, all domains D that occur will be stronger than the initial domain, *i.e.*, $D \sqsubseteq D_{init}$. A *false domain* is a domain D that maps at least one variable x to the empty set, *i.e.*, $\exists x \in V$ with $D(x) = \emptyset$.

A *valuation* θ is a mapping of variables to values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. We extend the valuation θ to map expressions or constraints involving the variables in the natural way. Let *vars* be the function that returns the set of variables appearing in an expression, constraint or valuation. In an abuse of notation, we define a valuation θ to be an element of a domain D, written $\theta \in D$, if $\theta(v) \in D(v)$ for all $v \in vars(\theta)$. Note that there a false domain D has no element valuations. Define a *valuation domain* D as one where $|D(v)| = 1, \forall v \in V$. We can define the corresponding valuation θ_D for a valuation domain D as $\{v \mapsto d \mid D(v) = \{d\}, v \in V\}$.

Then a constraint $c \in \mathcal{C}$ is a set of valuations over $vars(c)$ which give the allowable values for a set of variables. In FD solvers, constraints are implemented by propagators. A *propagator* f implementing c is an inclusion-decreasing function on domains such that for all domains $D \sqsubseteq D_{init}$: $f(D) \sqsubseteq D$ and no solutions are lost, *i.e.*, $\{\theta \in D \mid \theta \in c\} = \{\theta \in f(D) \mid \theta \in c\}$. We assume each propagator f is *checking*, that is if D is a valuation domain then $f(D) = D$ if and only if θ_D restricted to $vars(c)$ is a solution of c. Conversely, $f(D)$ is a false domain if and only if θ_D restricted to $vars(c)$ is not a solution of c. Given a set of constraints $\mathcal{C}$ we assume a corresponding set of propagators $F = \{f \mid c \in \mathcal{C}, f \text{ implements } c\}$.

A *propagation solver* $solv(F, D)$ for a set of propagators F and current domain D repeatedly applies all the propagators in F starting from domain D until there is no further change in resulting domain. $solv(F, D)$ is some weakest domain $D' \sqsubseteq D$ which is a fixed point (*i.e.*, $f(D') = D'$) for all $f \in F$.

FD solving interleaves propagation with search decisions. Given an initial problem ($\mathcal{C}$,D) where F are the propagators for the constraints $\mathcal{C}$, we first run the propagation solver $D' = solv(F, D)$. If this determines failure (D' is a false domain) then the problem has no solution and we backtrack to visit the next unexplored choice. If D' is a valuation domain then we have determined a solution. Otherwise we pick a variable $x \in V$ where $|D'(x)| \geq 2$ and split its domain $D'(x)$ into two disjoint parts $U_1 \cup U_2 = D'(x)$ creating two subproblems $(\mathcal{C}, D_1)$ and $(\mathcal{C}, D_2)$, where $D_i(x) = U_i$ and $D_i(v) = D'(v), v \neq x$, whose solutions are also solutions of the original problem. We then recursively explore the first problem, and when we have shown it has no solutions we explore the second problem.

As defined above FD propagation is only applicable to *satisfaction problems*. Finite domain solvers solve optimization problems by mapping them to repeated satisfaction problems. Given an objective function o to minimize under constraints $\mathcal{C}$ with domain D, the finite domain solving approach first finds a solution θ to $(\mathcal{C}, D)$, and then finds a solution to $(\mathcal{C} \cup \{o < \theta(o)\}, D)$, that is, the satisfaction problem of finding a better solution than previously found. It repeats this process until a problem is reached with no solution, in which case the last found solution is optimal. If the process is halted before proving optimality, then the solving process just returns the last solution found as the best known. We note that a dichotomic search on the values of objective function values is also possible, and requires solving fewer satisfaction problems to optimally solve the problem, but it does not allow the reuse of all nogoods between the different satisfaction problems.

FD propagation is a powerful generic approach to solving combinatorial optimization problems. Its chief strengths are the ability to model problems at a very high level, and the use of global propagators, that is specialized propagation algorithms, for important constraints.

2.1.1 Generalized Precedence Constraints

A *binary inequality propagator* f for a precedence constraint $x + d \leq y$ updates the domains of x and y in constant time as follows

$$f(D)(u) = \begin{cases} D(x) \cap (-\infty, \max_D(y) - d] & \text{if } u = x, \\ D(y) \cap [\min_D(x) + d, \infty) & \text{if } u = y, \\ D(u) & \text{otherwise} \end{cases}$$

where $u \in V$. Hence, the propagator infers a new upper bound on x if $\max_D(y) - d < \max_D(x)$ and a new lower bound on y if $\min_D(x) + d > \min_D(y)$.

2.1.2 Cumulative Resource Constraint

Of particular interest to us in this work is the global cumulative constraint for cumulative resource scheduling.

The cumulative constraint introduced by Aggoun and Beldiceanu (1993) takes four arguments, *i.e.*, s, p, r, and c. Each of the first three arguments are lists of the same length n and indicate information about a set of *activities*. $s[i]$ is the *variable start time* of the i^{th} activity, $p[i]$ is the fixed *duration* of the i^{th} activity, and $r[i]$ is the fixed *resource usage* (per time unit) of the i^{th} activity. The last argument c is the fixed *resource capacity*.

The cumulative constraints represent cumulative resources with a constant capacity over the considered project duration applied to non-preemptive activities, *i.e.*, if they are started they cannot be interrupted. Without loss of generality we assume that all values are integral and non-negative and there is a *maximal*

project duration t_{max} which is the latest time any activity can finish. Thus the cumulative constraint, denoted by `cumulative`, imposes the constraints:

$$\texttt{cumulative}(s, p, r, c) \equiv \sum_{i \in \mathcal{V}: s[i] \leq t < s[i]+p[i]} r[i] \leq c \qquad \forall t \in [0, t_{max}) \ ,$$

where $\mathcal{V}$ is the set of activities.

Example 2 Consider the five activities a, b, c, d, e from Example 1 with durations $2, 5, 3, 1, 2$ and resource requirements $3, 2, 1, 2, 2$ and a resource capacity of 4. This is represented by the cumulative constraint:

$$\texttt{cumulative}([s_a, s_b, s_c, s_d, s_e], [2, 5, 3, 1, 2], [3, 2, 1, 2, 2], 4) \ .$$

Imagine each task must start at time 0 or after and finish before time 8. The cumulative problem corresponds to scheduling the activities shown in Figure 2(a) into the Gantt chart shown to the left. □

There are many propagation algorithms for the cumulative constraint, but the most widely used for project scheduling problems is based on timetable propagation (see, *e.g.* Le Pape, 1994), because the precedence constraints usually mean the problem is not highly cumulative, *i.e.*, not many activities can be run concurrently.

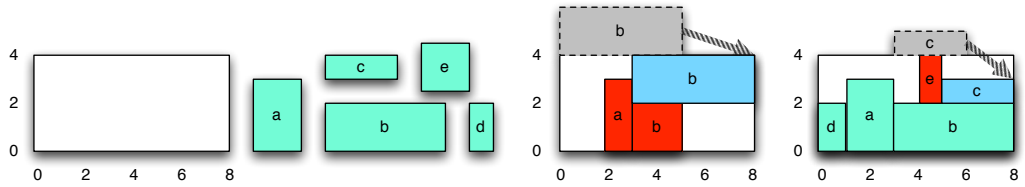
An activity i has a *compulsory part* given domain D from $[\max_D s[i] .. \min_D s[i] + p[i])$, that requires that activity i makes use of $r[i]$ resources at each of the times in $[\max_D s[i] .. \min_D s[i] + p[i] - 1]$ if the range is non-empty. The timetable propagator for `cumulative` first determines the *resource usage* profile $ru[t]$ which sums for each time t the resources required for all compulsory parts of activities at that time. If at some time t the profile exceeds the resource capacity, *i.e.*, $ru[t] > c$, the constraint is violated and failure detected. If at some time t the resources used in the profile are such that there is not enough left for an activity i, *i.e.*, $ru[t] + r[i] > c$, then we can determine that activity i cannot be scheduled to run during time t. If the earliest start time $\min_D s[i]$ of activity i, is such that the activity cannot be scheduled completely before time t, *i.e.*, $\min_D s[i] + p[i] > t$, we can update the earliest start time to be $t + 1$, similarly if the latest start time of the activity is such that the activity cannot be scheduled completely after t, *i.e.*, $\max_D s[i] \leq t$, then we can update the latest start time to be $t - p[i]$. For a full description of timetable propagation for `cumulative` see Schutt *et al.* (2011).

Example 3 Consider the cumulative constraint of Example 2. We assume that the domains of the start times are $D(s_a) = [1 .. 2]$, $D(s_b) = [0 .. 3]$, $D(s_c) = [3 .. 5]$, $D(s_d) = [0 .. 2]$, $D(s_e) = [0 .. 4]$. Then there are compulsory parts of activities a and b in the ranges [2..3) and [3..5) respectively shown in Figure 2(b) in red (dark). No other activities have a compulsory part. Hence the red contour illustrates the resource usage profile. Since activity b cannot be scheduled in parallel with activity a, and the earliest start time of activity b, which is 0, means that the activity cannot be scheduled before activity a we can reduce the domain of the start time for activity b to the singleton $[3 .. 3]$. This is illustrated in Figure 2(b). The opposite holds for activity a that cannot be run after activity b, hence the domain of its start time shrinks to the singleton range $[1 .. 1]$. Once we make these changes the compulsory parts of the activities a and b increase to the ranges [1..3) and [3..8) respectively. This in turn causes the start times of activities d and e to become $[0 .. 0]$ and $[3 .. 4]$ respectively, creating compulsory parts in the ranges [0..1) and [4..5) respectively. The latter causes the start time of activity c to become fixed at 5 generating the compulsory part in [5..8) which causes that the start time of activity e becomes fixed at 3. This is illustrated in Figure 2(c). In this case the timetable propagation results in a final schedule in the right of Figure 1. □

2.2 Boolean Satisfiability Solving

Let $\mathcal{B}$ be a set of Boolean variables. A *literal* l is either a Boolean variable $b \in \mathcal{B}$, *i.e.*, $l \equiv b$, or its negation, *i.e.*, $l \equiv \neg b$. The negation of a literal $\neg l$ is defined as $\neg b$ if $l \equiv b$ and b if $l \equiv \neg b$. A *clause* C is a set of literals understood as a disjunction. Hence clause $\{l_1, \ldots, l_n\}$ is satisfied if at least one literal l_i is true. An *assignment* A is a set of Boolean literals that does not include a variable and its negation, *i.e.*, $\nexists b \in \mathcal{B} : \{b, \neg b\} \subseteq A$. An assignment can be seen as a partial valuation on Boolean variables, $\{b \mapsto \mathit{true} \,|\, b \in A\} \cup \{b \mapsto \mathit{false} \,|\, \neg b \in A\}$. A theory T is a set of clauses. A SAT *problem* (T, A) consists of a set of clauses T and an assignment A over (some of) the variables occurring in T. Thus, the assignment A can be a partial, possible empty, assignment. A solution for the theory T is an assignment A containing each Boolean variable in either positive or negative context, and satisfying all clauses in T. Consequently, a solution for a SAT problem (T, A) is a solution A' of the theory T that is a superset of A, *i.e.*, $A' \supseteq A$.

A SAT solver based on the DPLL procedure (Davis *et al.*, 1962; Davis and Putnam, 1960) is a form of FD propagation solver specialized for Boolean clauses. Each clause is propagated by so-called *unit propagation*. Given an assignment A, unit propagation detects failure



(a) (b) (c)

Fig. 2: Figure illustrates the timetable propagation of the cumulative constraint for activities b and c where red (dark) boxes describe compulsory parts of unfixed activities.

using clause C if $\{\neg l \mid l \in C\} \subseteq A$, and unit propagation detects a new unit consequence l if $C \equiv \{l\} \cup C'$ and $\{\neg l' \mid l' \in C'\} \subseteq A$, in which case it adds l to the current assignment A. Unit propagation continues until failure is detected, or no new unit consequences can be determined.

SAT solvers exhaustively apply unit propagation to the current assignment A to generate all the consequences resulting in a new assignment A'. They then choose an unfixed variable b and create two equivalent problems $(T, A' \cup \{b\})$, $(T, A' \cup \{\neg b\})$ and recursively search these subproblems. The literals added to the assignment by choice are termed *decision literals*.

Modern DPLL-based SAT solving is a powerful approach to solving combinatorial optimization problems because it records nogoods that prevent the search from revisiting similar parts of the search space. The SAT solver records an explanation for each unit consequence discovered (the clause that caused unit propagation), and on failure uses these explanations to determine a set of mutually incompatible decisions, a *nogood*, which is added as a new clause to the theory of the problem. These nogoods drastically reduce the size of the search space needed to be examined. Another advantage of SAT solvers is that they track which variables are involved in the most failures (called *active* variables), and use a powerful autonomous search procedure which concentrates on the variables that are most active. The disadvantages of SAT solvers are the restriction to Boolean variables and the sometime huge models that are required to represent a problem because the only constraints expressible are clauses.

2.3 Lazy Clause Generation

LCG is a hybrid of FD propagation and SAT solving. The key idea in LCG is to run a FD propagation solver, but to build an explanation of the propagations made by the solver by recording them as clauses on a Boolean variable representation of the problem. Hence, as the FD search progresses, we lazily create a clausal representation of the problem. The hybrid has the advantages of FD solving, but inherits the SAT solvers ability to create nogoods to drastically reduce search, and uses activity based search.

2.3.1 Variable Representation

An LCG problem is stated as an FD problem, but each integer variable has a clausal representation in the SAT solver. In the remainder of this work, we use $[\![.]\!]$ as the *names* of Boolean variables. An integer variable $x \in \mathcal{X}$ with the initial domain $D_{init}(x) = [\, l \,..\, u \,]$ is represented by $2(u-l)+1$ Boolean variables $[\![x = l]\!]$, $[\![x = l+1]\!]$, $\ldots$, $[\![x = u]\!]$ and $[\![x \leq l]\!]$, $[\![x \leq l+1]\!]$, $\ldots$, $[\![x \leq u-1]\!]$. The variable $[\![x = d]\!]$ is *true* if x takes the value d, and *false* if x takes a value different from d. Similarly, the variable $[\![x \leq d]\!]$ is true if x takes a value less than or equal to d and false for a value greater than d.

We use the notation $[\![d \leq x]\!]$ to refer to the literal $\neg[\![x \leq d-1]\!]$.

Not every assignment of Boolean variables is consistent with the integer variable x, for example $\{[\![x = 3]\!], [\![x \leq 2]\!]\}$ (*i.e.*, both Boolean variables are true) requires that x is both 3 and ≤ 2. In order to ensure that assignments represent a consistent set of possibilities for the integer variable x we add to the SAT solver the clauses $DOM(x)$ that encode

$$[\![x \leq d]\!] \rightarrow [\![x \leq d+1]\!], \qquad l \leq d < u-1, \tag{1}$$

$$[\![x = l]\!] \leftrightarrow [\![x \leq l]\!], \tag{2}$$

$$[\![x = d]\!] \leftrightarrow ([\![x \leq d]\!] \wedge \neg[\![x \leq d-1]\!]), \quad l < d < u, \tag{3}$$

$$[\![x = u]\!] \leftrightarrow \neg[\![x \leq u-1]\!] \tag{4}$$

where $D_{init}(x) = [\, l \,..\, u \,]$. This equates to $u-l-1$ clauses for (Eq. 1) and $3(u-l-1)+4$ clauses for (Eq. 2–4). Note that clauses in (Eq. 2–4) are generated lazily on demand when propagation needs to express something using the literal $[\![x = d]\!]$ (see (Feydy, 2010) for details).

Any assignment A on these Boolean variables can be converted to a domain: $domain(A)(x) = \{d \in D_{init}(x) \mid \forall [\![c]\!] \in A, vars([\![c]\!]) = \{x\} : x = d \models c\}$, *i.e.*, the domain includes all values for x that are consistent with all the Boolean variables related to x. It should be noted that the domain may assign no values to some variable.

Example 4 Consider Example 1 and assume $D_{init}(s_i) = [\,0\,..\,15\,]$ for $i \in \{a, b, c, d, e\}$. The assignment $A = \{\neg[\![s_a \leq 1]\!], \neg[\![s_a = 3]\!], \neg[\![s_a = 4]\!], [\![s_a \leq 6]\!], \neg[\![s_b \leq 2]\!], [\![s_b \leq 5]\!], \neg[\![s_c \leq 4]\!], [\![s_c \leq 7]\!], \neg[\![s_e \leq 3]\!]\}$ is consistent with $s_a = 2$, $s_a = 5$, and $s_a = 6$. Therefore $domain(A)(s_a) = \{2, 5, 6\}$. For the remaining variables $domain(A)(s_b) = [\,3\,..\,5\,]$, $domain(A)(s_c) = [\,5\,..\,7\,]$, $domain(A)(s_d) = [\,0\,..\,15\,]$, and $domain(A)(s_e) = [\,4\,..\,15\,]$. Note that for brevity A is not a fixed point of unit propagation for $DOM(s_a)$ since we are missing many implied literals such as $\neg[\![s_a = 0]\!]$, $\neg[\![s_a = 8]\!]$ etc. □

2.3.2 Explaining Propagators

In Lcg, a propagator is extended from a mapping from domains to domains to a generator of clauses describing propagation. When $f(D) \neq D$ we assume the propagator f can determine a clause C to explain each domain change. Similarly, when $f(D)$ is a false domain the propagator must create a clause C that explains the failure.

Example 5 Consider the binary inequality propagator f for the precedence constraint $s_a + 2 \leq s_b$ from Example 1. When applied to the domains $D(s_i) = [\,0\,..\,15\,]$ for $i \in \{a, b\}$ it obtains $f(D)(s_a) = [\,0\,..\,13\,]$, and $f(D)(s_b) = [\,2\,..\,15\,]$. The clausal explanation of the change in domain of s_a is $[\![s_b \leq 15]\!] \rightarrow [\![s_a \leq 13]\!]$, similarly the change in domain of s_b is $\neg[\![s_a \leq -1]\!] \rightarrow \neg[\![s_b \leq 1]\!]$ ($[\![0 \leq s_a]\!] \rightarrow [\![2 \leq s_b]\!]$). These become the clauses $\neg[\![s_b \leq 15]\!] \vee [\![s_a \leq 13]\!]$ and $[\![s_a \leq -1]\!] \vee \neg[\![s_b \leq 1]\!]$. □

The explaining clauses of the propagation are passed to the Sat solver on which unit propagation is performed. Because the clauses will always have the form $C \rightarrow l$ where C is a conjunction of literals true in the current assignment, and l is a literal not true in the current assignment, the newly added clause will always cause unit propagation, adding l to the current assignment.

Example 6 Consider the propagation from Example 5. The clauses $\neg[\![s_b \leq 15]\!] \vee [\![s_a \leq 13]\!]$ and $[\![s_a \leq -1]\!] \vee \neg[\![s_b \leq 1]\!]$ are added to the Sat theory. Unit propagation infers that $[\![s_a \leq 13]\!]$ is *true* and $\neg[\![s_b \leq 1]\!]$ is *true* since $\neg[\![s_b \leq 15]\!]$ and $[\![s_a \leq -1]\!]$ are *false*, and adds these literals to the assignment. Note that the unit propagation is not finished, since for example the implied literal $[\![s_a \leq 14]\!]$, can be detected *true* as well. □

The unit propagation on the added clauses C is guaranteed to be as strong as the propagator f on the original domains. This means if $domain(A) \sqsubseteq D$ then $domain(A') \sqsubseteq f(D)$ where A' is the resulting assignment after addition of C and unit propagation (see Ohrimenko *et al.*, 2009).

Note that a single new propagation could be explained using different set of clauses. In order to get maximum benefit from the explanation we desire a "strongest" explanation as possible. A set of clauses C_1 is *stronger* than a set of clauses C_2 if C_2 implies C_1. In other words, C_1 restricts the search space at least as much as C_2.

Example 7 Consider explaining the propagation of the start time of the activity c described in Example 3 and Figure 2(c). The domain change $[\![5 \leq s_c]\!]$ arises from the compulsory parts of activity b and e as well as the fact that activity c cannot start before time 3. An explanation of the propagation is hence $[\![3 \leq s_c]\!] \wedge [\![3 \leq s_b]\!] \wedge [\![s_b \leq 3]\!] \wedge [\![3 \leq s_e]\!] \wedge [\![s_e \leq 4]\!] \rightarrow [\![5 \leq s_c]\!]$. We can observe that if $2 \leq s_c$ then the same domain change $[\![5 \leq s_c]\!]$ follows due to the compulsory parts of activity b and e. Therefore, a stronger explanation is obtained by replacing the literal $[\![3 \leq s_c]\!]$ by $[\![2 \leq s_c]\!]$.

Moreover, the compulsory parts of the activity b in the ranges $[\,3\,..\,3\,]$ and $[\,5\,..\,8\,]$ are not necessary for the domain change. We only require that there is not enough resources at time 4 to schedule task c. Thus the refined explanation can be further strengthened by replacing $[\![3 \leq s_b]\!] \wedge [\![s_b \leq 3]\!]$ by $[\![s_b \leq 4]\!]$ which is enough to force a compulsory part of s_b at time 4. This leads to the stronger explanation $[\![2 \leq s_c]\!] \wedge [\![s_b \leq 4]\!] \wedge [\![3 \leq s_e]\!] \wedge [\![s_e \leq 4]\!] \rightarrow [\![5 \leq s_c]\!]$. □

In this example the final explanation corresponds to a pointwise explanation defined in Schutt *et al.* (2011). In this work, we use the timetable propagation, as earlier described in this section, that generates pointwise explanations. The maximum size of these explanations ($maxLenCumu$) is bounded by $2 \times \max\{T \subseteq \{1, 2, \ldots, n\} \mid \sum_{i \in T} r[i] > c \text{ and } \forall j \in T : \sum_{i \in T - j} r[i] \leq c\}$ literals where n is the number of activities requiring some resource units of the cumulative resource and c is the resource capacity. For a full discussion about the best way to explain propagation of `cumulative` see Schutt *et al.* (2011).

2.3.3 Nogood generation

Since all propagation steps in Lcg have been mapped to unit propagation on clauses, we can perform nogood generation just as in a Sat solver. Here, the nogood generation is based on an *implication graph* and the

first unique implication point (1UIP). The graph is a directed acyclic graph where nodes represent fixed literals and directed edges reasons why a literal became *true*, and is extended as the search progresses. Unit propagation marks the literal it makes true with the clause that caused the unit propagation. The true literals are kept in a stack showing the order that they were determined as true by unit consequence or decisions.

For brevity, we do not differentiate between literals and nodes. A literal is fixed either by a search decision or unit propagation. In the first case, the graph is extended only by the literal and, in the second case, by the literal and incoming edges to that literal from all other literals in the clause on that the unit propagation assigned the *true* value to the literal.

Example 8 Consider the strongest explanation $[\![2 \leq s_c]\!] \wedge [\![s_b \leq 4]\!] \wedge [\![3 \leq s_e]\!] \wedge [\![s_e \leq 4]\!] \rightarrow [\![5 \leq s_c]\!]$ from Example 7. It is added to the SAT database as clause $\neg[\![2 \leq s_c]\!] \vee \neg[\![s_b \leq 4]\!] \vee \neg[\![3 \leq s_e]\!] \vee \neg[\![s_e \leq 4]\!] \vee [\![5 \leq s_c]\!]$ and unit propagation sets $[\![5 \leq s_c]\!]$ *true*. Therefore the implication graph is extended by the edges $[\![2 \leq s_c]\!] \rightarrow [\![5 \leq s_c]\!]$, $[\![s_b \leq 4]\!] \rightarrow [\![5 \leq s_c]\!]$, $[\![3 \leq s_e]\!] \rightarrow [\![5 \leq s_c]\!]$, and $[\![s_e \leq 4]\!] \rightarrow [\![5 \leq s_c]\!]$. □

Every node and edge is associated with the search level at which they are added to the graph. Once a conflict occurs, a nogood which is the 1UIP in LCG is calculated based on the implication graph. A conflict is recognized when the unit propagation reaches a clause where all literals are false. This clause is the starting point of the analysis and builds a first tentative nogood. Literals in the tentative nogood are replaced one by one by the literals from their incoming edges, in the reverse order that these literals were added to the implication graph. This process continues until the tentative nogood contains exactly one literal associated with the current conflict level. Thus, the time complexity of the nogood computation is bounded by the size of the extension of the implication graph at the conflict level. Given that in our case the timetable propagation of `cumulative` creates the largest explanations, the time complexity is bounded by $nprop \times maxLenCumu$ where $nprop$ is the number of domain reductions performed in the conflict level and $maxLenCumu$ is maximal length of an explanation for `cumulative`, which is described earlier. The resulting nogood is called the 1UIP nogood (Moskewicz *et al.*, 2001).

Example 9 Consider the RCPSP/max instance from Example 1 on page 2. Assume an initial domain of $D_{init} = [0..15]$ then after the initial propagation of the precedence constraints the domains are $D(s_a) = [0..8]$, $D(s_b) = [2..10]$, $D(s_c) = [3..12]$, $D(s_d) = [0..10]$, and $D(s_b) = [3..13]$. Note that no tighter bounds can be inferred by the cumulative propagator.

Assume search now sets $s_a \leq 0$. This sets the literal $[\![s_a \leq 0]\!]$ as true, and unit propagation on the domain clauses sets $[\![s_a = 0]\!]$, $[\![s_a \leq 1]\!]$, $[\![s_a \leq 2]\!]$, etc. In the remainder of the example, we will ignore propagation of the domain clauses and concentrate on the "interesting propagation."

The precedence constraint $s_c - 6 \leq s_a$ forces $s_c \leq 6$ with explanation $[\![s_a \leq 0]\!] \rightarrow [\![s_c \leq 6]\!]$. The precedence constraint $s_b + 1 \leq s_c$ forces $s_b \leq 5$ with explanation $[\![s_c \leq 6]\!] \rightarrow [\![s_b \leq 5]\!]$.

The timetable propagator for `cumulative` uses the compulsory part of activity a in [0..2) to force $s_d \geq 2$. The explanation for this is $[\![s_a \leq 0]\!] \rightarrow [\![2 \leq s_d]\!]$. The the precedence $s_d + 3 \leq s_e$ forces $s_e \geq 5$ with explanation $[\![s_d \geq 2]\!] \rightarrow [\![5 \leq s_e]\!]$.

Suppose next that search sets $s_b \leq 2$. It creates a compulsory part of b from [2..7) but there is no propagation from precedence constraints or `cumulative`.

Suppose now that the search sets $s_d \leq 2$. Then the precedence constraint $s_e - 3 \leq s_d$ forces $s_e \leq 5$ with explanation $[\![s_d \leq 2]\!] \rightarrow [\![s_e \leq 5]\!]$. This creates a compulsory part of d in [2..3) and a compulsory part of e in [5..7). In fact all the activities a, b, d and e are fixed now. Timetable propagation sees, since all resources are used at time 5, that activity c cannot start before time 6. A reason for this is $[\![2 \leq s_b]\!] \wedge [\![s_b \leq 5]\!]$ (which forces b to use 2 resources in [5..7)), plus $[\![5 \leq s_e]\!] \wedge [\![s_e \leq 5]\!]$ (which forces e to use 2 resources in [5..7)), plus $[\![3 \leq s_c]\!]$ (which forces c to overlap this time). Hence an explanation is $[\![2 \leq s_b]\!] \wedge [\![s_b \leq 5]\!] \wedge [\![5 \leq s_e]\!] \wedge [\![s_e \leq 5]\!] \wedge [\![3 \leq s_c]\!] \rightarrow [\![6 \leq s_c]\!]$.

This forces a compulsory part of c at time 6 which causes a resource overload at that time. An explanation of the failure is $[\![2 \leq s_b]\!] \wedge [\![s_b \leq 5]\!] \wedge [\![5 \leq s_e]\!] \wedge [\![s_e \leq 5]\!] \wedge [\![6 \leq s_c]\!] \wedge [\![s_c \leq 6]\!] \rightarrow fail$. The edges are shown in the implication graph of Figure 3 as dashed (for clarity).

The nogood generation process starts from this original explanation of failure. It removes the last literal in the nogood by replacing it by its explanation. Replacing $[\![6 \leq s_c]\!]$ by its explanation creates the new nogood $[\![2 \leq s_b]\!] \wedge [\![s_b \leq 5]\!] \wedge [\![5 \leq s_e]\!] \wedge [\![s_e \leq 5]\!] \wedge [\![3 \leq s_c]\!] \wedge [\![s_c \leq 6]\!] \rightarrow fail$. Since this nogood has only one literal that was made true after the last decision level $[\![s_e \leq 5]\!]$ this is the 1UIP nogood. Rewritten as a clause it is $[\![s_b \leq 1]\!] \vee \neg[\![s_b \leq 5]\!] \vee [\![s_e \leq 4]\!] \vee \neg[\![s_e \leq 5]\!] \vee [\![s_c \leq 2]\!] \vee \neg[\![s_c \leq 6]\!]$.

Now the solver backtracks to the previous decision level, undoing the decision $s_d \leq 2$ and its consequences. The newly added nogood unit propagates to force $s_e \geq 6$ with explanation $[\![2 \leq s_b]\!] \wedge [\![s_b \leq 5]\!] \wedge [\![5 \leq s_e]\!] \wedge [\![3 \leq s_c]\!] \wedge [\![s_c \leq 6]\!] \rightarrow [\![6 \leq s_e]\!]$, and the precedence

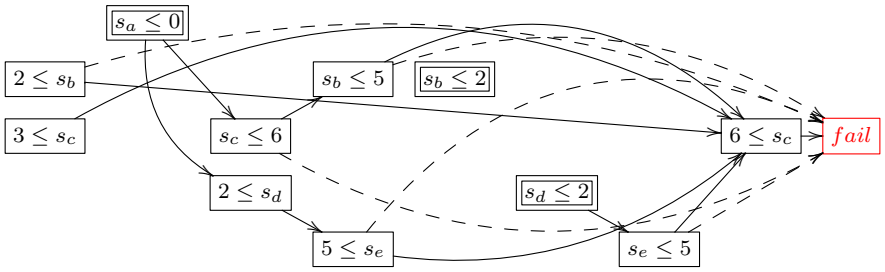


Fig. 3: (Part of) The implication graph for the propagation of Example 9. Decision literals are shown double boxed, while literals set by unit propagation are shown boxed.

constraint $s_e - 3 \leq s_d$ forces $s_d \geq 3$ with explanation $[\![6 \leq s_e]\!] \rightarrow [\![3 \leq s_d]\!]$. Search proceeds looking for a solution. □

Nogoods generated by this process can have a size as large as the number of possible Boolean variables in the SAT representation, except that there can be at most two non-redundant inequality literals for each integer variable involved. Note that all generated nogoods encode redundant information, and we could delete any of them at any time, but also lose the search reduction that their propagation creates. In this paper, all generated nogoods are kept permanently which requires space bounded by the maximal size of a nogood times the number of conflicts during the search.

3 Models for RCPSP/max

In this section, a basic model for RCPSP/max is first presented and then a number of model improvements.

An RCPSP/max can be represented as follows: A set of activities $\mathcal{V} = \{1, \ldots, n\}$ is subjected to generalized precedence relations in $\mathcal{E} \subset \mathcal{V}^2 \times \mathbb{Z}$ between two activities, and scarce resources in $\mathcal{R}$. The goal is to find a schedule $s = (s_i)_{i \in \mathcal{V}}$ that respects the precedence constraints and resource constraints, and minimizes the project duration where s_i is the *start time* of the activity i.

Each activity i has a finite *duration* p_i and requires (non-negative) r_{ik} units of resource k, $k \in \mathcal{R}$ for its execution, where r_{ik} is the *resource requirement* or *usage* of activity i for resource k. A resource $k \in \mathcal{R}$ has a constant capacity R_k over the planning period which cannot be exceeded at any point in time. The *planning period* is given by $[0, t_{max})$ where t_{max} is the maximal project duration. Each resource k is modeled by a single cumulative constraint:

$\texttt{cumulative}(s, p, [r_{ik} | i \in \mathcal{V}], R_k)$

Generalized precedence relations $(i, j, d_{ij}) \in \mathcal{E}$ between the activities i and j are subject to the constraint $s_i + d_{ij} \leq s_j$. If a minimal time lag d^+_{ij}, *i.e.*, $0 \leq d^+_{ij}$, and a maximal time lag d^-_{ji},*i.e.*, $0 > d^-_{ji}$, exist for an activity j concerning to i then the start time s_j is restricted to $[s_i + d^+_{ij} .. s_i - d^-_{ji}]$. In the case of $d^+_{ij} = -d^-_{ji}$, the activity j must start exactly d^+_{ij} time units after i.

The RCPSP/max problem can be stated as follows:

$$\text{minimize } MS \tag{5}$$

subject to

$$s_i + d_{ij} \leq s_j \qquad \forall (i, j, d_{ij}) \in \mathcal{E}, \tag{6}$$

$$\texttt{cumulative}(s, p, [r_{ik} | i \in \mathcal{V}], R_k) \qquad \forall k \in \mathcal{R}, \tag{7}$$

$$s_i + p_i \leq MS \qquad \forall i \in \mathcal{V}, \tag{8}$$

$$0 \leq s_i \leq t_{max} - p_i \qquad \forall i \in \mathcal{V}. \tag{9}$$

The objective is to minimize the project duration MS (Eq. 5) which is subjected to the generalized precedence constraints (Eq. 6), the resource constraints (Eq. 7), and the objective constraints (Eq. 8). All start times must be non-negative and all activities must be scheduled in the planning period (Eq. 9).

A basic constraint model uses a binary inequality propagator for each precedence relation (Eq. 6), one `cumulative` propagator for each resource constraint, and a binary inequality propagator for each objective constraint (Eq. 8). Since none of the propagators either generate or use equality literals of the form $[\![x = d]\!]$ they and their defining clauses in $DOM(x)$ are never generated during execution. Hence only $(n+1) \times t_{max} - \sum_{i \in \mathcal{V}} p_i$ Boolean variables and $(n+1) \times (t_{max} - 1) - \sum_{i \in \mathcal{V}} p_i$ clauses are needed for the domain representation of the start time variables and the objective variable. Thus, the number of literals in a nogood is bounded to $2 \times n + 2$.

For the remainder of this section, let an RCPSP/max instance be given with activities $\mathcal{V} = \{1, 2, \ldots, n\}$, generalized precedences $\mathcal{E}$, resources $\mathcal{R}$, and a planning period $[0, t_{max})$.

This basic (constraint) model has a number of weaknesses: first the initial domains of the start times are

large, second each precedence constraint is modeled as one individual propagator, and finally the SAT solver in LCG has no structural information about activities in disjunction.

A smaller initial domain reduces the size of the problem because fewer Boolean variables are necessary to represent the integer domain in the SAT solver. It can be computed in a preprocessing step by taking into account the precedences in $\mathcal{E}$ as described in the next subsection. Individual propagators for precedence constraints may not be so bad for a small number of precedence constraints, but for a larger number of propagators, their queuing behavior may result in long and costly propagation sequences. A global propagator can efficiently adjust the time-bounds in $\mathcal{O}(n \log n+m)$ time (see Feydy *et al.*, 2008) if the set of precedence constraints is feasible. Since the solver used herein does not offer this global propagator, an individual propagator is used for each precedence constraint.

3.1 Initial Domain

A smaller initial domain can be obtained for the start time variables by applying the Bellman-Ford single source shortest path algorithm (see Bellman, 1958; Ford and Fulkerson, 1962) on the digraph $G = (\mathcal{V}', \mathcal{E}')$ where $\mathcal{V}' = \mathcal{V} \cup \{v_0, v_{n+1}\}$, $\mathcal{E}' = \{(i, j, -d_{ij}) \mid (i, j, d_{ij}) \in \mathcal{E}\} \cup \{(v_0, i, 0), (i, v_{n+1}, -p_i) \mid i \in \mathcal{V}\}$, v_0 is the source node, and v_{n+1} is the sink node. The digraph is referred to as the activity-on-node network in the literature (*e.g.* Bartusch *et al.*, 1988; Neumann and Schwindt, 1997). If the digraph contains a negative-weight cycle then the RCPSP/max instance is infeasible. Otherwise the shortest path $v_0 \rightsquigarrow i$ from the source v_0 to an activity i determines the earliest possible start time for i, *i.e.*, $-w(v_0 \rightsquigarrow i)$ where $w(.)$ is the length of the path. Similarly the shortest path from an activity i to the sink v_{n+1} determines the latest possible start time for i in any schedule, *i.e.*, $t_{max}+w(i \rightsquigarrow v_{n+1})$. The Bellman-Ford algorithm has a runtime complexity of $\mathcal{O}(|\mathcal{V}|\times|\mathcal{E}|)$.

These earliest and latest start times can not only be used for initial smaller domains, but also to improve the objective constraints by replacing them with

$$s_i - w(i \rightsquigarrow v_{n+1}) \leq MS \qquad \forall i \in \mathcal{V}$$

since the start time will push back the minimum project duration by at least this much. These smaller domains result in $\sum_{i\in\mathcal{V}}(-w(v_0 \rightsquigarrow i)-w(i \rightsquigarrow v_{n+1}))$ less Boolean variables and clauses for the variable representation in the SAT solver. The maximal size of a nogood is not affected.

Preliminary experiments confirmed that starting the solution process with a smaller initial domain offers major improvements in the runtime for solving an instance and generating a first solution, especially on larger instances.

3.2 Activities in Disjunction

Two activities i and $j \in \mathcal{V}$ are in *disjunction*, if they cannot be executed at the same time, *i.e.*, the sum of their resource requirements for at least one resource $k \in \mathcal{R}$ is greater than the available capacity: $r_{ik}+r_{jk} > R_k$. Activities in disjunction can be exploited in order to reduce the search space.

The simplest way to model two activities i and j in disjunction is by two half-reified constraints (Feydy *et al.*, 2011) sharing the same Boolean variable B_{ij}.

$$B_{ij} \rightarrow s_i + p_i \leq s_j \qquad \forall i < j \text{ in disjunction} \qquad (10)$$

$$\neg B_{ij} \rightarrow s_j + p_j \leq s_i \qquad \forall i < j \text{ in disjunction} \qquad (11)$$

If B_{ij} is *true* then i must end before j starts (denoted by $i \ll j$), and if B_{ij} is *false* then $j \ll i$. The literals B_{ij} and $\neg B_{ij}$ can be directly represented in the SAT solver, consequently B_{ij} represents the relation (structure) between these activities. The propagator of such a reified constraint can only infer new bounds on left hand side of the implication if the right hand side is false, and on the start times variables if the left hand side is true. For example, the right hand side in the second constraint is false if and only if $\max_D s_i - \min_D s_j < p_j$. In this case the literal $\neg B_{ij}$ must be false and therefore $i \ll j$.

We add these redundant constraints to the model which allows the propagation solver to determine information about start time variables more quickly. For each constraint, the Boolean variable is directly modeled in the SAT solver and the maximal size of a nogood increases by one.

The detection of which activity runs before another can be further improved by considering the domains of the start times, and the minimal distances in the activity-on-node-network (see Dorndorf *et al.*, 2000). This requires keeping track of the minimal distance between each pair of activities in the network. Since such a propagator is not offered by the LCG solver, we do not use their improvement for the benchmarks.

4 The Branch-and-Bound Algorithm

Our branch-and-bound algorithms are based on start-time and conflict-driven branching strategies. We use them alone or in combination. After each branch all

constraints from the model, including constraints for activities in disjunctive, are propagated until a fixpoint is reached or the inconsistency for the partial schedule or the instance is proven. In the first case a new node is explored and in the second case an unexplored branch is chosen if one exists or backtracking is performed.

The propagation solver uses a priority queue in which the unit propagation in the SAT solver has the highest priority followed by propagators for precedence constraints, objective constraints, and the constraints for activities in disjunction. The propagators for cumulative constraints have the lowest priority. Thus, they are executed after no further domain reduction can be made by any other propagators.

4.1 Start-time Branching

The start-time branching strategy selects an unfixed start time variable s_i with the smallest possible start time $\min_D s_i$. If there is a tie between several variables then the variable with the biggest size, *i.e.*, $\max_D s_i - \min_D s_i$, is chosen. If there is still a tie then the variable with the lowest index i is selected. The binary branching is as follows: left branch $s_i \leq \min_D s_i$, and right branch $s_i > \min_D s_i$. In the remainder this branching is denoted by MSLF.

This branching creates a time-oriented branch-and-bound algorithm similar to Dorndorf *et al.* (2000), but it is simpler and does not involves any dominance rules. Hence, it is weaker than their algorithm.

4.2 Conflict-driven Branching

The conflict-driven branching is a binary branching over literals in the SAT solver. In the left branch, the literal is set to *true* and, in the right branch, to *false*. As described in Subsection 2.3.1 on page 6 the Boolean variables in the SAT solver represent values in the integer domain of a variable x (*e.g.* $\neg[\![x \leq 3]\!]$ and $[\![x \leq 10]\!]$) or a disjunction between activities. Hence, it creates a branch-and-bound algorithm that can be considered as a mixture of time oriented and conflict-set oriented.

As a branching heuristic, an activity-based heuristic is used which is a variant of the Variable-State-Independent-Decaying-Sum (VSIDS) (Moskewicz *et al.*, 2001). This heuristic is embedded in the SAT solver. In each branch, it selects the literal with the highest activity counter where an *activity counter* is assigned to each literal, and is increased during conflict analysis if the literal is related to the conflict. The analysis results in a nogood which is added to the clause data base. Here, we use the 1UIP as a nogood. Once in a while all counters are decreased by the same factor not only to avoid a (possible) overflow, but also to give literals in recent conflicts more weight.

In order to accelerate the finding of solutions and increase the robustness of the search on hard instances, VSIDS can be combined with restarts, which has been shown beneficial in SAT solving. On restart the set of nogoods and the activity counters have changed, so that the search will explore a very different part of the search tree. In the remainder VSIDS with restart is denoted by RESTART. Different restart policies can be applied. Here a geometric restart on failed nodes with an initial limit of 250 and a restart factor of 2.0 is used (Walsh, 1999; Huang, 2007).

4.3 Hybrid Branching

At the beginning of a search, the activity counters of the variables have to be initialized somehow. By default they are all initialized to the same value. With no useful information in this initial setting, these activities can mislead VSIDS resulting in poor performance. To avoid this, we consider a hybrid search that uses MSLF to search initially, which has the effect of modifying the activity counts to reflect some structure of the problem, and then switch to VSIDS after the first restart. Here, we switch the searches after exploration of the first 500 nodes unless otherwise stated. The strategy is denoted by HOT START, and HOT RESTART when VSIDS is combined with restart.

5 Computational Results

We carried out experiments on RCPSP/max instances available from Project Duration Problem RCPSP/max (2010) and accessible from the PSPLib (2010). Our approach is compared to the best known exact and non-exact methods so far on each testset. The detailed results can be found in the electronic supplementary material and at `http://www.cs.mu.oz.au/~pjs/rcpsp`.

Our methods are evaluated on the following testsets which were systematically created using the instance generator ProGen/max (Schwindt, 1995):

CD: C, and D: each consisting of 540 instances with 100 activities and 5 resources.

UBO: UBO10, UBO20, UBO50, UBO100, and UBO200: each containing 90 instances with 5 resources and 10, 20, 50, 100, and 200 activities resp. (cf. Franck *et al.*, 2001).

SM: J10, J20, and J30: each containing 270 instances with 5 resources and 10, 20, and 30 activities resp. (cf. Kolisch *et al.*, 1998).

Note that although the testset SM consists of small instances they are considerably harder than, *e.g.*, UBO10 and UBO20.

The experiments were run on an Intel(R) Xeon(R) CPU E54052 processor with 2 GHz clock running GNU Linux. The code was written in Mercury using the G12 Constraint Programming Platform and compiled with the Mercury Compiler using grade hlc.gc.trseg. All clauses created during propagation and all nogoods inferred during conflict analysis were permanently added to the original problem, *i.e.*, there is no garbage collection of clauses. Each run was given a 10 minute runtime limit.

5.1 Setup and Table Notations

In order to solve each instance, a two-phase process was used. Both phases used the basic model with the two described extensions (cf. Subsections 3.1 and 3.2).

In the first phase, a HOT START search was run to determine a first solution or to prove the infeasibility of the instance. The feasibility runs were set up with the trivial upper bound on the project duration $t_{max} = \sum_{i \in \mathcal{V}} \max(p_i, \max\{d_{ij} \mid (i, j, d_{ij}) \in \mathcal{E}\})$. The feasibility test was run until a solution was found or infeasibility proved. If a solution was found, we use UB to denote the project duration of the resulting solution. In the first phase, the search strategy should be good at both finding a solution or proving infeasibility, but not necessarily at finding and proving the optimal solution. Hence, it could be exchanged with methods that might be more suitable than HOT START. To improve HOT START for finding a solution, when we used it in the first phase we explored $5 \times n$ nodes using the start-time branching strategy before switching to VSIDS.

In the second optimization phase, each feasible instance was set up again this time with $t_{max} = UB$. The tighter bound is highly beneficial to lazy clause generation since it reduces the number of Boolean variables required to represent the problem. The search for optimality was performed using one of the various search strategies defined in the previous section.

The execution of the two-phased process leads to the following measurements.

rt_{max}: The runtime limit in seconds (for both phases together).

rt_{avg}: The average runtime in seconds (for both phases).

fails: The average number of infeasible nodes encountered in both phases of the search.

feas: The percentage of instances for which a solution was found.

infeas: The percentage of instances for which the infeasibility was proven.

opt: The percentage of instances for which an optimal solution was found and proven.

Δ_{LB}: The average relative deviation (as a percentage) from the best known lower bounds of feasible instances given in Project Duration Problem RCPSP/max (2010). The relative deviation is $(best_k - LB_k)/LB_k$ for an instance k where $best_k$ and LB_k are the best found and best known lower bound of the instance k respectively.

cmpr(i): Columns with this header give measurements only related to those instances that were solved by each procedure where i is the number of these instances.

all(i): Columns with this header compare measurements for all instances examined in the experiment where i is the number of these instances.

Entries with the symbol "-" indicate no comparable number was available. Entries with two numbers indicate that the corresponding procedure was applied several times to the instance and the first number gives the average over all runs and the number in parentheses gives the best number from all runs. Entries marked by a star "*" indicate that a procedure was not able to find a solution for all feasible instances and therefore the corresponding number may not be comparable with the number from other procedures.

5.2 Comparison of the different strategies

In the first experiment, we compare all of our search strategies against each other on all testsets. The strategies are compared in terms of rt_{avg} and fails for each testset.

The results are summarized in the Table 1. Similar to the results for RCPSP in Schutt *et al.* (2011) all strategies using VSIDS are superior to the start-time methods (MSLF), and similarly competitive. HOT RESTART is the most robust strategy, solving the most instances to optimality and having the lowest Δ_{LB}. Restart makes the search more robust for the conflict-driven strategies, whereas the impact of restart on MSLF is minimal.

In contrast to the results in Schutt *et al.* (2011) for RCPSP the conflict-driven searches were not uniformly superior to MSLF. The three instances 67, 68, and 154 from J30 were solved to optimality by MSLF and MSLF with restart, but neither RESTART and HOT RESTART could prove the optimality in the given time limit, whereas VSIDS and HOT START were not even able to find an optimal solution within the time limit. Furthermore, none our methods could find a first solution for the UBO200 instances 2, 4, and 70 nor prove

Table 1: Comparison on the testsets CD, UBO, and SM.

Procedure	feas	opt	infeas	Δ_{LB}	cmpr(2230)		all(2340)	
					rt_{avg}	fails	rt_{avg}	fails
MSLF	85.0	80.60	15.0	3.96785	7.73	6804	35.96	23781
MSLF with restart	85.0	80.60	15.0	3.96352	7.80	6793	36.04	23787
VSIDS	85.0	82.26	15.0	3.76928	2.16	1567	22.91	13211
RESTART	85.0	82.26	15.0	3.73334	**2.02**	**1363**	22.38	**12212**
HOT START	85.0	82.31	15.0	3.84003	2.22	1684	22.71	12933
HOT RESTART	85.0	**82.35**	15.0	**3.73049**	2.04	1475	**22.36**	12341

the infeasibility for the UBO200 instance 40 within 10 minutes. Only for these instances we let our methods run until a first solution was found or infeasibility was proven. The corresponding numbers are included in Table 1. A detailed discussion of these instances follows in Subsection 5.4.

5.3 Results on the testset CD

Table 2 presents the results for the testset CD where 98.1% (1.9%) of the instances are feasible (infeasible). Here, we compare RESTART and HOT RESTART with the time-oriented branch-and-bound procedure (denoted by B&B$_{\text{D00}}$) from Dorndorf *et al.* (2000) and the evolutionary algorithm EVA from Ballestín *et al.* (2011). The method B&B$_{\text{D00}}$ performs better on this testset than the methods proposed by De Reyck and Herroelen (1998); Schwindt (1998a); Fest *et al.* (1999)[3]. Moreover, B&B$_{\text{D00}}$ is the best published exact method on this testset so far. Their B&B$_{\text{D00}}$ method was implemented in C++ using ILOG SOLVER and ILOG SCHEDULER. Their experiments were run on a Pentium Pro/200 PC with NT 4.0 as operating system, thus their results were obtained on a machine approximately ten times slower.

We compare our results achieved with a runtime limit of 1 second to their results with a limit of 100 seconds which should be clearly in favor of them. While B&B$_{\text{D00}}$ can prove feasibility and infeasibility of all instances, the first-phase HOT START search with one second was unable to prove infeasibility of four infeasible instances or find solutions to two feasible instances. It does prove infeasibility of these four infeasible instances in less than 2.1 seconds and finds a first solution for these two feasible instances in 4.80 seconds and 5.04 seconds respectively. Within one second both our methods RESTART and HOT RESTART were able to prove the optimality of substantially more instances than B&B$_{\text{D00}}$. With more time our methods are able to prove optimality of almost all instances in the testsets.

One reason for the first-phase results at one second may simply be that there is a reasonable set up time required for lazy clause generation to generate all the Boolean variables and hence there is not much time for search. Another reason for the weakness of proving infeasibility is that our model only contains propagators that determine the order of activities in disjunction concerning their domains, but not also their minimal distance in the transitive closure of all precedences.[4] Dorndorf *et al.* (2000) show that these propagators are important for a fast detection of infeasibility. That HOT START is not so good in finding a first solution is not surprising, since the search is not as problem specific as that of B&B$_{\text{D00}}$. In order to overcome these problems, we could replace our first phase with, *e.g.*, the method of B&B$_{\text{D00}}$ to prove infeasibility and generate a first solution, and then use our second phase approach to find and prove optimality.

The method EVA is the best published metaheuristic on this testset. Their results were obtained on a Samsung X15 Plus computer with Pentium M processor with 1400 MHz clock speed. This means that our machine is at least 1.46 times faster than theirs. Their limits are a maximum of 5000 schedules, halting the process at any time after 10 generations where the best schedule could not be improved. Our methods generate better schedules within 10 seconds than their approach, which can be seen from the lower Δ_{LB} 3.20 which is less than 3.24.

Overall, our methods are able to close 310 open problems and improve the upper bound for all 21 remaining open problems in testset CD, according to the results recorded in Project Duration Problem RCPSP/-max (2010).

5.4 Results on testset UBO

Table 3 compares our procedures RESTART and HOT RESTART with the truncated branch-and-bound methods FBS$_{\text{F01}}$, the heuristic DM$_{\text{F01}}$, and the genetic algo-

[3] Results from Schwindt (1998a) are taken from Dorndorf *et al.* (2000)

[4] The missing propagators are not available in the G12 Constraint Programming Platform.

Table 2: Results on the testset CD.

Procedure	rt_{max}	rt_{avg}	feas	opt	infeas	Δ_{LB}
B&B$_{\text{D00}}$	100	-	**98.1**	71.7	**1.9**	4.6▲
Eva	-	0.62	98.1	$\geq$ 65.9	-	**3.24 (3.16)**
Restart	1	0.38	97.9	**78.1**	1.6	4.73⋆
	10	1.39	98.1	**89.8**	1.9	**3.20**
	100	6.17	98.1	**94.0**	1.9	**2.86**
	600	19.32	98.1	95.8	1.9	2.81
Hot Restart	1	0.44	97.9	76.8	1.6	4.87⋆
	10	1.49	98.1	89.6	1.9	**3.20**
	100	6.27	98.1	93.9	1.9	**2.86**
	600	19.42	98.1	**96.0**	1.9	**2.79**

The Δ_{LB} entry marked by ▲ is based on the lower bounds presented in Schwindt (1998b) which were not accessible for us.

rithm GA$_{\text{F01}}$ all proposed by Franck *et al.* (2001) on the UBO instances with up to 100 activities where 81.7% (18.3%) of the instances are feasible (infeasible). In this table, we add the column feas + infeas showing the sum of percentage of feas and infeas because the corresponding numbers for FBS$_{\text{F01}}$ are not available. Their results were obtained on personal computer PII with a 333MHz processor running NT 4.0 as operating system, *i.e.*, our machine is about 6.2 times faster. They imposed a time limit of n seconds, *e.g.*, an instance with 100 activities was given at most 100 seconds. We compare our methods with 10 (100) times lower time limit, which should be favorable to the other methods.

Their methods were able to prove the feasibility or infeasibility for all instances (except one instance for the method FBS$_{\text{F01}}$). Indeed DM$_{\text{F01}}$ is extremely fast requiring just 0.03 seconds on average, but it does not necessarily find very good solutions, as shown by the high Δ_{LB}.

In contrast, our first-phase was not always able to find a first solution or prove infeasibility with the time limit $n/100$. No solution was found for 6 instances with 100 activities and the infeasibility was not shown for 11 (1) instances with 100 (50) activities. Once the time limit was extended to $n/10$ then the first phase was always able to find a solution or prove infeasibility. If we compare Δ_{LB} achieved with a time limit $n/10$ (note for a time limit $n/100$ the data is not comparable, since our methods could not find a solution for all feasible instances) then our methods have a substantially better Δ_{LB} than their approaches, *i.e.*, our methods are quicker in improving the project duration. Our approaches could prove optimality for a substantial fraction of these problems even with time limit $n/100$.

In the Table 4, we compare our results with the best metaheuristic Eva from Ballestín *et al.* (2011) on the UBO instances with up to 100 activities. Our methods create better schedules within 100 seconds than the evolutionary algorithm Eva leading to a smaller lower bound deviation.

The Table 5 presents the results on ubo200 which are compared to the iterative flattening searches Ifs, Ifs-Fr, and Ifs-Mcsr from Oddi and Rasconi (2009).[5] The table contains the extra column Δ_{UB} that reports the percentage of the average relative deviation from the best known upper bounds of feasible instances given in Project Duration Problem RCPSP/max (2010). It is calculated similarily to Δ_{LB}, but using the best known upper bound. Here, 88.9% (11.1%) instances are feasible (infeasible). Note Franck *et al.* (2001) also run their methods on ubo200, but the presented results are accumulated with the results on instances with 500 and 1000 activities, so that a comparison is not possible.

Within the given time limit Hot Start was not able to find a solution for the instances 2, 4, and 70 nor to prove the infeasibility for the instance 40. In order to compare the results with Oddi and Rasconi, we let our solution method run until a solution was found or infeasibility proven. The runtimes for the instances 2, 4, 40, and 70 are 1030, 1478, 1139, and 3103 seconds, respectively. Interestingly, the first obtained solutions have a better upper bound by 62, 46, and 37 for the instances 2, 4, and 70, respectively, than the previously best known upper bound recorded in Project Duration Problem RCPSP/max (2010).

Comparing these results on Δ_{UB} with Oddi and Rasconi clearly shows our procedures achieve better schedules. Restart and Hot Restart perform comparably. The ubo200 instances clearly show that Hot Start as the search strategy in the first phase can have difficulties in finding a first solution or proving infeasibility.

In total our approaches close 178 open instances and improve the upper bound for 27 instances of 31 remaining open instances with 200 activities or less in the testset UBO, according to the results recorded in Project Duration Problem RCPSP/max (2010).

[5] No machine details are given in Oddi and Rasconi (2009).

Table 3: Results on the testset UBO for UBO10, UBO20, UBO50 and UBO100 instances in comparison with FBS_{F01}, DM_{F01}, and GA_{F01}.

Procedure	rt_{max}	rt_{avg}	feas + infeas	feas	opt	infeas	Δ_{LB}
FBS_{F01}	n	12.4	99.66	-	-	-	6.82*
DM_{F01}	n	0.03	100	81.7	-	18.3	10.72
GA_{F01}	n	3.16	100	81.7	-	18.3	6.93
RESTART	$n/100$	0.21	95.0	80.0	70.8	15.0	5.73*
	$n/10$	0.78	100	81.7	75.3	18.3	4.99
HOT RESTART	$n/100$	0.25	95.0	80.0	69.7	15.0	5.73*
	$n/10$	0.81	100	81.7	75.3	18.3	5.04

Table 4: Results on the testset UBO for UBO10, UBO20, UBO50 and UBO100 instances in comparison with EVA.

Procedure	rt_{max}	rt_{avg}	feas	opt	infeas	Δ_{LB}
EVA	-	0.38	81.7	-	-	**4.82 (4.79)**
RESTART	1	0.22	80.0	**71.4**	15.3	5.60*
	10	0.89	81.7	**75.3**	18.3	4.92
	100	5.32	81.7	**77.2**	18.3	**4.51**
	600	24.47	81.7	**78.1**	18.3	**4.40**
HOT RESTART	1	0.26	80.0	70.6	15.3	5.65*
	10	0.92	81.7	**75.3**	18.3	5.01
	100	5.26	81.7	**77.2**	18.3	4.55
	600	24.14	81.7	**78.1**	18.3	4.43

Table 5: Results on the testset UBO for UBO200 instances.

Procedure	rt_{max}	rt_{avg}	feas	opt	infeas	Δ_{LB}	Δ_{UB}
IFS	-	2148.7	88.9	-	-	-	2.06
IFS-FR	-	2024.7	88.9	-	-	-	1.81
IFS-MCSR	-	1716.7	88.9	-	-	-	1.65
RESTART	100	29.55	81.1	67.8	7.8	7.37*	-0.41*
	600	139.0	85.6	**68.9**	10.0	10.11*	-1.110*
	600+$^\blacktriangle$	187.5	88.9	**68.9**	11.1	11.88	-1.249
HOT RESTART	100	29.9	81.1	**68.9**	7.8	**7.22***	**-0.48***
	600	139.0	85.6	**68.9**	10.0	**10.10***	**-1.111***
	600+$^\blacktriangle$	186.9	88.9	**68.9**	11.1	**11.87**	**-1.250**

$^\blacktriangle$ For comparison purposes the instances 2, 4, 40, and 70 were run until a first solution was found or infeasibility proven.

5.5 Results on testset SM

Finally for the testset SM we compare our approaches MSLF, RESTART, and HOT RESTART with the method $\text{B\&B}_{\text{S98}}$ from Schwindt (1998a)[6], ISES from Cesta *et al.* (2002), and SWO(BR) from Smith and Pyle (2004). The method $\text{B\&B}_{\text{S98}}$ (Schwindt, 1998a) is a branch-and-bound algorithm that resolves resource conflicts by adding precedence constraints between activities and has been run on a Pentium 200 with a 100 seconds time limit. ISES is a heuristic that also adds precedence constraints between activities in order to resolve/avoid resource conflicts, uses restarts and has been run on a SUN UltraSparc 30 (266 MHz) with the same time limit. The method SWO(BR) (Smith and Pyle, 2004) is a squeaky wheel optimization. Their method is divided into two stages: schedule generation and prioritization where the schedule is created by a heuristic with priority scheme and the latter changes the priorities on variables depending on how "well" it is handled in the former stage. Their benchmarks were performed on a 1700 Mhz Pentium 4. Note that ISES and SWO(BR) are not exact methods, *i.e.*, they cannot prove infeasibility unless the precedence graph contains a positive weight cycle, and optimality is only proven if the project duration of the solution found equals the known lower bound.

Table 6 presents the results for the 270 instances from SM with 30 activities. From these instances 185, *i.e.*, 68.5% are feasible and 85, *i.e.*, 31.5% infeasible. All our approaches could prove feasibility and infeasibility of all instances within one second whereas $\text{B\&B}_{\text{S98}}$ could not find a solution for a few feasible instances. Moreover, our methods could prove optimality significantly more often than the exact method $\text{B\&B}_{\text{S98}}$ (and clearly also the incomplete methods). All our methods were, on average, able to find on better solutions in one second than these approaches as indicated by a lower Δ_{LB}. For these harder benchmarks, our methods clearly

[6] The paper (Schwindt, 1998a) was not accessible for us, so that here the reported results are taken from Cesta *et al.* (2002).

Table 6: Results on the J30.

Procedure	rt_{max}	rt_{avg}	feas	opt	infeas	Δ_{LB}
B&B$_{S98}$	100	-	67.7	42.6	-	9.56▲
ISES	100	22.68	68.5	33.9 (35.6)	-	10.99 (10.37)
SWO(BR)	10	1.07	68.5	35.0	-	10.3
MSLF	1	0.16	68.5	58.1	31.5	8.91
	10	0.82	68.5	61.9	31.5	8.40
	100	4.90	68.5	**64.8**	31.5	8.23
	600	21.61	68.5	**65.5**	31.5	8.20
RESTART	1	0.12	68.5	**61.5**	31.5	8.38
	10	0.57	68.5	**64.1**	31.5	8.19
	100	3.92	68.5	**64.8**	31.5	8.17
	600	21.34	68.5	65.2	31.5	**8.12**
HOT RESTART	1	0.12	68.5	**61.5**	31.5	**8.37**
	10	0.59	68.5	**64.4**	31.5	**8.18**
	100	3.93	68.5	**64.8**	31.5	**8.16**
	600	21.47	68.5	65.2	31.5	8.13

The Δ_{LB} entry marked by ▲ is based on the lower bounds presented in Schwindt (1998b) which were not accessible for us.

outperform the competition. One reason could be that constraint propagation over `cumulative` has a greater benefit than on other testsets because here more activities can be run simultaneously.

Our approaches each give similar results: RESTART and HOT RESTART are superior to MSLF up to 10 seconds, and all are similar to each other with longer time limits. For this problem set, MSLF could prove optimality for three instances where RESTART and HOT RESTART only found the optimal solution. On the other hand, MSLF could not find an optimal solution for two instances where RESTART and HOT RESTART could. It seems that MSLF may better suit problems where more activities can be executed in parallel, but this needs further investigation.

Experiments were also carried out on the instances with 10 or 20 activities. All our methods could solve all 270 instances with 10 activities within 0.05 seconds. And all our methods could solve all 270 instances with 20 activities within 30 seconds. Moreover, for the instances with 20 activities, an optimal solution was found within 1 second for all feasible instances.

Here, our approaches close 85 open problems and improve the upper bound for 3 problems of the 6 remaining open problems in the testset SM, according to the results recorded in Project Duration Problem RCPSP/max (2010).

6 Conclusion

In this paper, we minimize the project duration of RCPSP/max using a generic constraint programming solver that includes nogood learning facilities and conflict-driven search. Experiments on three well-established benchmark suites show that our solver is able to find better solutions quicker than competing approaches, and prove optimality for many more instances than competing approaches.

We use a two-phase process. In the first phase, a solution is generated or infeasibility is proven and, in the second phase, a branch-and-bound algorithm is used for optimization where the problem is set up with an upper bound on the project duration found from the first solution. In contrast to some previous approaches, we use individual propagators for precedence constraints instead of a propagator taking all precedence constraints into account at once. This yields not only weaker propagation, but also slower detection of infeasibility, in particular for instances with a large number of precedence constraints. Hence, our generic search used in the first phase is sometimes slower in finding a first solution than some other problem-specific approaches in the literature. However, the first-phase generic search could be replaced by one of these methods.

Overall, our method closes 573 open problems and improves a further 51 upper bounds on the project duration of the 58 remaining open problems, according to the best known results given in Project Duration Problem RCPSP/max (2010). We note though that the methods from Ballestín *et al.* (2011), and Oddi and Rasconi (2009) may have found better upper bounds on some of these problems, but we could not find any record of them. Note that our method is highly robust: our method proves the optimal value for each already closed instance in every test. Furthermore, for every open instance in every test set we either close the instance or improve the upper bounds, except for 7 instances, in 4 of which we still regenerate the best known upper bound.

Acknowledgements We are thankful to the reviewers for their helpful comments and suggestions. NICTA is funded by the Aus-

tralian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

Aggoun A, Beldiceanu N (1993) Extending CHIP in order to solve complex scheduling and placement problems. Mathematical and Computer Modelling 17(7):57–73

Ballestín F, Barrios A, Valls V (2011) An evolutionary algorithm for the resource-constrained project scheduling problem with minimum and maximum time lags. Journal of Scheduling 14:391–406, DOI 10.1007/s10951-009-0125-9

Bartusch M, Möhring RH, Radermacher FJ (1988) Scheduling project networks with resource constraints and time windows. Annals of Operations Research 16(1):199–240

Bellman R (1958) On a routing problem. Quarterly of Applied Mathematics 16(1):87–90

Blazewicz J, Lenstra JK, Rinnooy Kan AHG (1983) Scheduling subject to resource constraints: classification and complexity. Discrete Applied Mathematics 5(1):11–24

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

Cesta A, Oddi A, Smith SF (2002) A constraint-based method for project scheduling with time windows. Journal of Heuristics 8(1):109–136

Davis M, Putnam H (1960) A computing procedure for quantification theory. Journal of the ACM 7:201–215

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

De Reyck B, Herroelen W (1998) A branch-and-bound procedure for the resource-constrained project scheduling problem with generalized precedence relations. European Journal of Operational Research 111(1):152–174

Dorndorf U, Pesch E, Phan-Huy T (2000) A time-oriented branch-and-bound algorithm for resource-constrained project scheduling with generalised precedence constraints. Management Science 46(10):1365–1384

Fest A, Möhring RH, Stork F, Uetz M (1999) Resource-constrained project scheduling with time windows: A branching scheme based on dynamic release dates. Tech. Rep. 596, Technische Universität Berlin

Feydy T (2010) Constraint programming: Improving propagation. PhD thesis, The University of Melbourne

Feydy T, Schutt A, Stuckey PJ (2008) Global difference constraint propagation for finite domain solvers. In: Antoy S, Albert E (eds) Proceedings of Principles and Practice of Declarative Programming – PPDP 2008, ACM, pp 226–235

Feydy T, Somogyi Z, Stuckey PJ (2011) Half reification and flattening. In: Lee JHM (ed) Proceedings of Principles and Practice of Constraint Programming – CP 2011, Springer, Lecture Notes in Computer Science, vol 6876, pp 286–301

Ford LR, Fulkerson DR (1962) Flows in Networks. Princeton University Press

Franck B, Neumann K, Schwindt C (2001) Truncated branch-and-bound, schedule-construction, and schedule-improvement procedures for resource-constrained project scheduling. OR Spectrum 23(3):297–324

Herroelen W, De Reyck B, Demeulemeester E (1998) Resource-constrained project scheduling: A survey of recent developments. Computers & Operations Research 25(4):279–302

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

Huang J (2007) The effect of restarts on the efficiency of clause learning. In: Veloso MM (ed) Proceedings of Artificial Intelligence – IJCAI 2007, pp 2318–2323

Kolisch R, Schwindt C, Sprecher A (1998) Project Scheduling: Recent Models, Algorithms and Applications, Kluwer Academic Publishers, chap Benchmark instances for project scheduling problems, pp 197–212

Le Pape C (1994) Implementation of resource constraints in ILOG Schedule: A library for the development of constraint-based scheduling systems. Intelligent Systems Engineering 3(2):55–66

Marriott K, Stuckey PJ (1998) Programming with Constraints: An Introduction. The MIT Press, Cambridge, MA, USA

Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an efficient SAT solver. In: Design Automation Conference, ACM, New York, NY, USA, pp 530–535

Neumann K, Schwindt C (1997) Activity-on-node networks with minimal and maximal time lags and their application to make-to-order production. OR Spectrum 19:205–217

Oddi A, Rasconi R (2009) Iterative flattening search on rcpsp/max problems: Recent developments. In: Oddi A, Fages F, Rossi F (eds) Recent Advances in

Constraints, Lecture Notes in Computer Science, vol 5655, Springer Berlin / Heidelberg, pp 99–115

Ohrimenko O, Stuckey PJ, Codish M (2009) Propagation via lazy clause generation. Constraints 14(3):357–391

Project Duration Problem RCPSP/max (2010) URL `http://www.wior.uni-karlsruhe.de/LS_Neumann/Forschung/ProGenMax/rcpspmax.html`

PSPLib (2010) Project Scheduling Problem Library. URL `http://129.187.106.231/psplib/`

Schutt A, Feydy T, Stuckey PJ, Wallace MG (2009) Why cumulative decomposition is not as bad as it sounds. In: Gent IP (ed) Proceedings of Principles and Practice of Constraint Programming – CP 2009, Lecture Notes in Computer Science, vol 5732, Springer Berlin / Heidelberg, pp 746–761

Schutt A, Feydy T, Stuckey PJ, Wallace MG (2011) Explaining the cumulative propagator. Constraints 16(3):250–282

Schwindt C (1995) ProGen/max: A new problem generator for different resource-constrained project scheduling problems with minimal and maximal time lags. WIOR 449, Universität Karlsruhe, Germany

Schwindt C (1998a) A branch-and-bound algorithm for the resource-constrained project duration problem subject to temporal constraints. WIOR 544, Universität Karlsruhe, Germany

Schwindt C (1998b) Verfahren zur Lösung des ressourcenbeschränkten Projektdauerminimierungsproblems mit planungsabhängigen Zeitfenstern. Shaker-Verlag

Smith TB, Pyle JM (2004) An effective algorithm for project scheduling with arbitrary temporal constraints. In: McGuinness DL, Ferguson G (eds) Proceedings of Artifical intelligence – AAAI 2004, AAAI Press / The MIT Press, pp 544–549

Stuckey PJ, García de la Banda MJ, Maher MJ, Marriott K, Slaney JK, Somogyi Z, Wallace MG, Walsh T (2005) The G12 project: Mapping solver independent models to efficient solutions. In: Gabbrielli M, Gupta G (eds) Proceedings of Logic Programming – ICLP 2005, Lecture Notes in Computer Science, vol 3668, Springer Berlin / Heidelberg, pp 9–13

Tsang E (1993) Foundations of Constraint Satisfaction. Academic Press

Walsh T (1999) Search in a small world. In: Proceedings of Artificial intelligence – IJCAI 1999, Morgan Kaufmann, pp 1172–1177