ECONSTOR
Make Your Publications Visible.

A Service of

ZBW Leibniz-Informationszentrum Wirtschaft
Leibniz Information Centre for Economics

Horbach, Andrei

Working Paper

A boolean satisfiability approach to the resource-constrained project scheduling problem

Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel, No. 644

Provided in Cooperation with:
Christian-Albrechts-University of Kiel, Institute of Business Administration

Suggested Citation: Horbach, Andrei (2009) : A boolean satisfiability approach to the resource-constrained project scheduling problem, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel, No. 644, Universität Kiel, Institut für Betriebswirtschaftslehre, Kiel

This Version is available at:
https://hdl.handle.net/10419/147562

Standard-Nutzungsbedingungen:

Die Dokumente auf EconStor dürfen zu eigenen wissenschaftlichen Zwecken und zum Privatgebrauch gespeichert und kopiert werden.

Sie dürfen die Dokumente nicht für öffentliche oder kommerzielle Zwecke vervielfältigen, öffentlich ausstellen, öffentlich zugänglich machen, vertreiben oder anderweitig nutzen.

Sofern die Verfasser die Dokumente unter Open-Content-Lizenzen (insbesondere CC-Lizenzen) zur Verfügung gestellt haben sollten, gelten abweichend von diesen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Terms of use:

Documents in EconStor may be saved and copied for your personal and scholarly purposes.

You are not to copy documents for public or commercial purposes, to exhibit the documents publicly, to make them publicly available on the internet, or to distribute or otherwise use the documents in public.

If the documents have been made available under an Open Content Licence (especially Creative Commons Licences), you may exercise further usage rights as specified in the indicated licence.

WWW.ECONSTOR.EU

Mitglied der Leibniz-Gemeinschaft

Manuskripte
aus den
Instituten für Betriebswirtschaftslehre
der Universität Kiel

No. 644

A Boolean Satisfiability Approach to the Resource-Constrained Project Scheduling Problem

Andrei Horbach[1]

März 2009

[1]: Andrei Horbach
Christian-Albrechts-Universität zu Kiel,
Institut für Betriebswirtschaftslehre,
Olshausenstr. 40, 24098 Kiel, Germany
http://www.bwl.uni-kiel.de/bwlinstitute/Prod
horbach@bwl.uni-kiel.de

Abstract

We formulate the resource-constrained project scheduling problem as a satisfiability problem and adapt a satisfiability solver for the specific domain of the problem. Our solver is lightweight and shows good performance both in finding feasible solutions and in proving lower bounds. Our numerical tests allowed us to close several benchmark instances of the RCPSP that have never been closed before by proving tighter lower bounds and by finding better feasible solutions. Using our method we solve optimally more instances of medium and large size from the benchmark library PSPLIB and do it faster compared to any other existing solver.

1 Introduction

The resource-constrained project scheduling problem (RCPSP) consists in scheduling activities with predefined durations and demands on each of the given renewable resources subject to partial precedence and resource constraints. The objective is to minimize the makespan. Due to its general formulation and practical importance the problem has been the subject of intensive research since the last decades, see for review Brucker et al. [4] and Herroelen [16].

In this paper we propose a fast and robust method for the RCPSP that is implemented without use of any proprietary optimization software. This method is general and can be applied to many other combinatorial problems. At the same time, it is very efficient. Our solver outperforms all other complete RCPSP solvers on hard test instances of medium and large size. For many of the test instances our solver is the first one to solve them to optimality.

Our method is based on the technique originating from the field of artificial intelligence. Its key element is a reduction to the conjunctive normal form satisfiability problem (SAT). The decision version of SAT has been historically the first problem proven to be NP-complete (Cook [5] and Levin [20]).

The SAT solving techniques have been developed to such an extent that for many combinatorial problems a reduction to SAT and the use of a SAT solver can be a good alternative to a problem specific solver running on the original problem formulation.

Surprisingly, there are a few if any competitive applications of a SAT solver to optimization problems of operations research described in the literature. One possible explanation could be some misinterpretation of the famous negative result of Haken [14]. He shows the non-existence of a complete SAT solver based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm that can prove in polynomial time the infeasibility of a trivial variant of the bipartite matching problem formulated as follows: place $n-1$ pigeons into n holes such that in each hole sits exactly one pigeon. Without contradicting the inefficiency for this trivial problem, our numerical tests clearly demonstrate the efficiency of an adapted complete SAT solver for a far less trivial one.

A SAT solver works on a feasible set described by means of a Boolean formula in conjunctive normal form (CNF). A clause in such a formula can be considered as a 'low-level' combinatorial constraint. At the same time, many 'high-level' combinatorial constraints can be represented as the conjunction of such clauses. A difficulty for practical use, however, can be the number of clauses needed for such representation. This number can be huge and can grow very fast with the size of the original combinatorial constraint.

Many combinatorial problems contain knapsack constraints in their formulations. Such constraints can bee seen as 'high-level' ones compared to CNF clauses. Various reductions of linear constraints over Boolean variables to CNF have been proposed in the literature, see for instance Eén and Sörensson [12]. We do not know any polynomial size reduction of linear constraints that could be shown to be efficient in the numerical tests. If a linear inequality over Boolean variables contains only positive coefficients, it can be replaced by a number of cover inequalities. Each of the cover inequalities can be easily transformed to a CNF clause. The number of cover inequalities and therefore the number of clauses needed for their representation can grow exponentially with the size of the original linear constraint. One possible way to overcome this size problem is to generate the clauses 'on the fly', dynamically adding them to the database while the SAT solver is running, where and if they are needed. This can be done in a very similar way as it has been done for a linear solver embedded into a cutting plane framework, where the solver obtains new inequalities from a separation routine.

Modern integer linear solvers have an interface for adding cuts and variables generated by a procedure which has knowledge about the specific problem domain. A similar interface to the SAT solver is needed for the implementation of our method. Since there is no SAT solver providing such an interface, we show here on the example of an open code SAT solver how it can be adapted in order to obtain cover clauses whenever they are needed.

One of the simplest but very fast and robust implementations of the SAT solver is MiniSat developed by Eén and Sörensson [11]. MiniSat is an extended C++ implementation of the DPLL algorithm. In this paper we adapt MiniSat to achieve efficiency in the specific domain of the RCPSP. Our method can be used in combination with any SAT solver based on the DPLL algorithm.

The paper is organized as follows. Section 2 presents a rigorous formulation of the RCPSP and briefly discusses relevant complete methods used by other researches. Section 3 considers the SAT problem together with the DPLL algorithm. Section 4 formulates the feasibility version of the RCPSP as series of SAT problems. Section 5 is devoted to our adaption of the SAT solver for these specific SAT problems. Section 6 presents our numerical results and compares them with results of other authors. Section 7 contains summary and suggestions for future research.

2 The Resource-Constrained Project Scheduling Problem

The resource-constrained project scheduling problem (RCPSP) can be formulated as follows. Given:

- a set R of renewable resources with limited capacities R_k for each $k \in R$,
- a set $V = \{0, 1, \ldots, n+1\}$ of activities with integer duration d_i for each $i \in V$,
- resource demand r_{ik} for each activity i and resource k,
- a subset E of $V \times V$ defining precedence constraints,
- the maximum makespan T, for the *feasibility* version of the RCPSP.

Activity 0 represents the start of the project and is a predecessor of each other activity. Activity $n+1$ represents the end of the project and is a successor of each other activity. They both have zero durations and zero resource demands. If S_i is the start time of activity i, we say that i is *in process* in periods $S_i, S_i+1, \ldots, S_i+d_i-1$. To determine is a vector of start times $(S_0, S_1, \ldots, S_{n+1})$ satisfying the following constraints:

- precedence constraints for all $(i,j) \in E$: $S_i + d_i \leq S_j$,
- resource constraints for each period and for each resource k: the cumulated demand of activities that are in process in the period does not exceed the resource capacity R_k,
- for the *feasibility* version of RCPSP: $S_{n+1} \leq T$,
- for the *optimization* version of RCPSP: the completion time S_{n+1} is to be minimized.

There are a great number of publications devoted to various extensions of the RCPSP, see e.g. the review of Hartmann and Briskorn [15]. The variant of the problem we consider here is apparently the most interesting one because of

- its simple but rather general formulation,
- the existence of challenging instances even of small dimension not solved to optimality until now,
- the fact that the power of the most methods developed for extensions of the RCPSP can be evaluated also on the instances of the RCPSP.

To determine the optimum of an instance of the RCPSP one needs to solve two problems: construct a feasible solution and prove, that no better feasible solution exists. Methods that are good for the first problem are not necessarily good for the second one and vice versa.

Good-quality solutions can be obtained by heuristics. Apparently the best existing results have been reported by Debels and Vanhoucke [8] for their genetic algorithm.

Branch and bound algorithms are suitable both for searching and proving optimality. They can be enforced by methods that prune the search space. These can be techniques of constraint propagation, see Dorndorf et al. [10] and Liess and Michelon [21]. The method of Demeulemeester and Herroelen [9] seems to be a very efficient one at least for the instances of small size. This method uses so called *cut sets* that help to filter out dominated nodes of the search tree, if a better node has been already considered.

The search space can be reduced, if an algorithm for computing lower bounds is available. Such an algorithm tries to find lower bounds for the distance between a pair of activities. Relaxing the resource constraints we obtain the *critical path lower bounds*, which can be calculated in $O(n^2 \log n)$ iterations of a shortest path algorithm. Practically, the Floyd-Warshall algorithm with the runtime of $O(n^3)$ is usually used. Tighter lower bounds can be obtained, if the resource constraints are respected at least to some extent. Klein and Scholl [18] review distinct lower bounds and provide a metastrategy which combines them in order to produce even better bounds. Klein and Scholl [17] use this method in a *scattered* branch and bound framework and report competitive results.

Tight lower bounds can be obtained from a linear relaxation of the problem. Mingozzi et al. [22] propose various integer programming formulation of the RCPSP. They obtain lower bounds by relaxing some of the constraints in these formulations.

One of their relaxations is based on the replacement of the precedence constraints by *disjunctive* constraints and the allowance of *preemptions*. They define *feasible subsets* as subsets of activities that can be processed in parallel. Then for V as the base set and for the collection of all feasible subsets they formulate the minimum cardinality *set multicover problem* with d_i as the coverage requirement for each activity i. Relaxing the integrality in the standard integer linear programming formulation of this set multicover problem, they obtain a lower bound for the RCPSP.

Brucker et al. [3] make use of this idea in order to compute lower bounds for their branch and bound scheme. Since the linear problem appearing here has a huge number of variables, they implement the delayed column generation to solve it. Brucker and Knust [2] deal further with this formulation and strengthen it by dividing the time horizon $[0, T]$ into subintervals. Then they consider for each subinterval only the feasible subsets of activities that can be in process within the subinterval. They use constraint propagation techniques to tighten the time windows of the activities. Baptiste and Demassey [1] propose valid inequalities that together with a constraint propagation technique lead to a significant improvement of the lower bounds for many of hard test instances.

Most of the solvers for the RCPSP have been evaluated on the so called KSD instances generated by Kolisch et al. [19]. These instances are included in the library PSPLIB (http://webserver.wi.tum.de/psplib/). The library supports an automatical submission and validation of solutions and contains values for upper and lower bounds. It is apparently the most extensively searched benchmark library for the RCPSP. There are four sets of KSD instances with 30, 60, 90 and 120 activities. The instances are generated with the problem generator ProGen under diverse parameter settings. Each group contains both easy and hard instances. Despite all efforts applied to solve the instances of the library in the last decade, up to now only the group with 30 activities has been reported completely solved to optimality.

3 The Satisfiability Problem

A propositional formula is a formula that is defined over variables that take values in the set $\{true, false\}$. A propositional formula is in conjunctive normal form (CNF) if it is a conjunction (AND, $\wedge$) of *clauses*, where each clause is a disjunction (OR, $\vee$) of *literals*. A literal is either a variable, then it is called a positive literal, or its negation (NOT, $\neg$), then it is called a negative literal.

The conjunctive normal form satisfiability problem (CNF SAT or just SAT) is defined as follows. Given a CNF, does there exist an assignment of the variables, such that the CNF evaluates to *true* under such assignment? If such a satisfying assignment exists, we say that the formula is satisfiable and the assignment is called a *model*. Otherwise the formula is unsatisfiable.

Many researches contributed to the development of efficient algorithms for SAT making a great advance in the ability to solve problem instances involving over a million of variables and several millions of clauses. The results of the annual international SAT competitions

can be found at www.satcompetition.org.

The basic algorithm for SAT has been proposed by Davis and Putnam [6] and Davis et al. [7], see Algorithm 1. This algorithm not only answers the question of satisfiability but also finds a satisfying assignment if it exists. Surprisingly, this algorithm happened to be so good that it is still the basis of the most modern complete SAT solvers. A version of this algorithm is implemented by Eén and Sörensson [11] in their solver MiniSat.

Algorithm 1 DPLL-Algorithm

```
decision_level = 0
while true do
  propagate()
  if not conflict then
    if all variables assigned then
      return Satisfiable
    else
      ++decision_level, decide()
  else
    analyze()
    if top-level conflict found then
      return Unsatisfiable
    else
      backtrack()
```

The algorithm starts at decision level zero, chooses a literal (a variable and the *true* or the *false* direction) to branch on, makes the *true* assignment of the literal and completes it by *unit propagation* which is done in `propagate()`. If a variable is chosen for branching it is marked as *decision* variable. If the value is assigned to the variable at unit propagation, it is a *propagated* variable. If `propagate()` returns no conflict, the algorithm steps to the next decision level and chooses a next literal to branch on. If under the current partial assignment `propagate()` determines a conflict (a clause can not be satisfied under this partial assignment), the conflict is returned. The conflict is analyzed. The result of `analyze()` is the backtracking level, the largest level up to that all assignments have to be canceled such that the last conflict is not *false* under the current partial assignment. In the state-of-the-art solvers `analyze()` returns also a *learnt clause*, which is a consequence of the other clauses in the database and, hence, must be satisfied by each satisfying assignment and can be considered as a *reason* of the last conflict. The learnt clause is added to the database and propagation goes on.

The components of DPLL are:

- branching procedure `decide()`, which makes choice of the next literal to branch on;
- procedure `propagate()`, which makes unit propagation and determines conflicts;
- procedure `analyze()`, which analyzes conflicts, decides about the level of backtracking, and generates learnt clauses;
- backtrack procedure, which unwinds all made assignments up to the backtracking level returned by `analyze()`.

Two techniques should be noted as making special contribution to the efficiency of modern SAT solvers. *Clause learning*, being involved in unit propagation, speeds up the recognition of future conflicts and may lead to an impressive rise of performance. *The watched literals scheme* proposed by Moskewicz et al. [23] and first implemented in their solver zChaff is now a standard method for efficient constraint propagation. The key idea behind this scheme is to maintain two special literals for each not yet satisfied clause that are not $false$ under the current partial assignment. As long as the clause has two such literals, it cannot be involved in unit propagation. Only if one of these two literals is assigned to $false$, `propagate()` assigns *true* to the other literal, if the clause is not satisfied yet. More details on state-of-the-art SAT solvers can be found in Gomes et al. [13].

4 Reduction of the feasibility RCPSP to SAT

Assume we are given an instance of the feasibility RCPSP. We determine for each activity i its earliest (es_i) and its latest (ls_i) start times, its earliest (ef_i) and its latest (lf_i) finish times, given that activity 0 starts at 0 and activity $n+1$ starts at T. These times are calculated in a standard way using the critical-path method by Floyd-Warshall algorithm. As a by-product of this algorithm we obtain for each pair of activities (i,j) the shortest time interval between them, denoted by $dist_{ij}$ (if i is not a predecessor of j, $dist_{ij} := -\infty$, $dist_{ii} := -d_i$).

Then, given these start intervals, we define in a straightforward way two types of Boolean variables of the SAT problem:

- For each $i \in V$ and $t \in \{es_i, \ldots, ls_i\}$ a *start* variable

$$s_{it} = \begin{cases} true & \text{if activity } i \text{ starts at period } t, \\ false & \text{otherwise.} \end{cases}$$

- For each $i \in V$ and $t \in \{es_i, \ldots, lf_i\}$ a *process* variable

$$u_{it} = \begin{cases} true & \text{if activity } i \text{ is in process at period } t, \\ false & \text{otherwise.} \end{cases}$$

Next we formulate a CNF over these variables, such that each satisfying assignment of this CNF encodes a feasible schedule and vice versa. We define the following three classes of clauses.

To provide a correct combination of start and process variables for each activity we need the *consistency clauses*:

$$\neg s_{it} \vee u_{il} \qquad i \in V, t \in \{es_i, \ldots, ls_i\}, l \in \{t, \ldots, t + d_i - 1\} \tag{1}$$

If i starts at period t, it has to be in process in the following d_i periods.

To provide the satisfaction of the precedence constraints we need *precedence clauses*:

$$\neg s_{it} \bigvee_{l=es_j,\ldots,es_i-d_j} s_{jl} \qquad (j,i) \in E, t \in \{es_i,\ldots,ls_i\} \tag{2}$$

If i starts at t, all its predecessors start early enough to allow this start of i.

To complete the formulation we adapt the definition of a *cover* for the knapsack problem. We call a subset of activities $C \in V$ a *cover* if there is a resource $k \in R$ such that $\sum_{i \in C} r_{ik} > R_k$. A cover C is *minimal* if $C \setminus \{i\}$ is a cover for no $i \in C$.

To guarantee the satisfaction of the resource constraints we need the *cover clauses*:

$$\bigvee_{i \in C} \neg u_{it} \qquad t \in \{0,\ldots,T-1\}, C \subset V, C \text{ is a minimal cover.} \tag{3}$$

The cover clauses guarantee, that no subset of activities with total demand exceeding the capacity of any resource is in process in any period.

The first two classes are of polynomial size (given the maximum activity duration is limited and this limit is not a part of the input) and can be easily treated by any modern SAT solver. The number of clauses in (3) can grow exponentially with the size of the problem. Therefore, it does not seem to be a very promising approach to generate all of them in advance. We make use of a technique, which is similar to the cut generation approach in linear programming. We maintain a database of clauses which contains all of (1) and (2) and some of (3) clauses and extend it by clauses of (3) that are violated or can be violated by the current (partial) assignment. In line with the notation used in linear programming we call this process *separation*. We consider the details of the technical implementation of this method in the next section.

5 Adaption of the DPLL-Algorithm for the RCPSP

As mentioned above, we have to implement separation of cover clauses. One option is to generate all cover inequalities and then to start the SAT solver. Our initial experiments showed, that this approach works only for small instances. In particular, we could generate all minimal cover clauses without memory overflow for KSD instances with at most 30 activities.

The second option is to adapt the standard separation strategy of the integer linear programming by generating unsatisfied cover clauses each time as a feasible assignment is found. If some violated cover clauses have been found, add them to the database and restart the SAT solver. This way is easy to implement, since all necessary interfaces are provided and the SAT solver can be used as a black box. This method has, however, one essential drawback: the violated cover clauses are found late and the SAT solver usually has to cancel most of the variable assignments. Our numerical experiments showed, that this approach is acceptable for easy instances, but stays inefficient comparing to our third approach.

The approach we develop in this paper separates cover clauses as early as possible. We do it every time when a process variable is fixed to *true*. Usually, a SAT solver does not have an interface to do this, so we have to adapt function `propagate()`. This function is

called for the current assignment and does some work for each assigned variable that is not propagated yet. The work is to fix all not assigned variables whose values can be inferred from the current partial assignment. If a clause is $false$ under the current assignment (conflict clause), it is returned to the overall search procedure.

We adapt `propagate()` in such a way, that it tests whether the current literal is a process variable. If it is the case, it looks which other process variables corresponding to the same period t are assigned to *true*, and, if the cumulated demand on any resource at t exceeds the supply, it generates a minimal cover clause and returns the clause to the search procedure as a conflict. If such conflicting cover clause is not found, `propagate()` updates the residual value of each resource. If the residual value of some resource is less than the demand of activity j, it infers $u_{jt} = false$ and adds the corresponding cover clause to the database.

The behavior of the solver can be influenced by various means. First of all, we can use different SAT formulation of the problem. The formulation considered here is apparently one of the most straightforward ones. We only performed minor changes of our initial formulation using the results of some numerical tests. However, we succeeded to find out a possibility to strengthen this formulation by adding some valid clauses.

Our numerical tests have shown, that the following class of valid clauses improves the runtime on medium and hard KSD instances of all sizes:

$$\neg u_{it} \vee u_{i,t+1} \vee s_{i,t-d_i+1} \quad i \in V,\ t \in \{ef_i, \ldots, lf_i - 1\} \tag{4}$$

The introduction of these clauses has an effect that the period t is fixed as the end period of activity i, when the solver fixes two variables u_{it} and u_{it+1} to *true* and $false$, correspondingly. Then the start time of i, i.e. $it - d_i + 1$, is inferred immediately by `propagate()`.

It is a well-known fact, that a SAT solver can perform much faster on some formulas if it succeeds to choose the right variables to branch on early. A right order of branch variables can significantly improve the runtime also for several optimization problems, see e.g. Pesch and Teitzlaff [24]. If we deal with formulas of some specific problem domain, it may be expected, that early branching on variables of some type would lead faster to a solution. We may then indicate a higher branching priority for such variables. Our initial tests have shown, that branching on process variables leads faster to a feasible solution or infeasibility.

6 Numerical Experiments

In this section we present the results of our numerical tests on the KSD instances generated by Kolisch et al. [19], which are a part of the library PSPLIB. The KSD instances contain four instance sets j30, j60, j90, j120, where e.g. j30 corresponds to the set with 30 activities in each instance. Each of the first three sets contains 480 instances generated by ProGen (see Kolisch et al. [19]) with different parameter settings. Some of the instances (exactly 120 in each set j30, j60 and j90) are trivial in the meaning that an optimal schedule can be obtained by fixing the start time of each activity i to its earliest start time es_i. Set j120 contains 600 instances without trivial ones.

We compare our results with results reported by other authors:

1. Demeulemeester and Herroelen [9] (DH), obtained on IBM PS/2 Model P75 with a 80486 processor running at 25 MHz under Windows NT.
2. Klein and Scholl [17] (KS), obtained on a computer with a Pentium 166 processor running under Windows 95.
3. Sprecher [25] (Sp), obtained on a computer with a Pentium 166 processor running under Linux.
4. Brucker et al. [3] (BKST), obtained on SUN/Sparc 20/801 running at 80 MHz under Unix Solaris 2.5.
5. Dorndorf et al. [10] (DPP), obtained on a computer with a Pentium Pro 200 under Windows NT.
6. Liess and Michelon [21] (LM), obtained on NEC PowerMate running at 2GHz under Debian GNU/Linux.

We also take as a reference the mixed integer solver Ilog CPLEX version 11.0 (www.ilog.com/products/cplex/) running on Compaq 8710 with Intel Core2Duo T7700 2.4 GHz under Windows XP with default parameter settings. The integer programming model used here can be found in the Appendix.

To make a comparison with the first five solvers we used a Pentium II computer with 64 MB RAM running at 266 MHz under Windows 98. This computer is approximately 40% faster than the Pentium Pro used for testing DPP solver, 140% faster than the computers Sp and KS solvers were tested on, and 20 times faster than the computer used for testing DH solver. These estimations are based on the results of SPECint95 test (www.specbench.org/cpu95/results/cint95.html).

Further results were obtained on Dell Precision with Intel Core2Duo T6400 2.2GHz with 1 GB of RAM running under Windows XP. Our solver uses only one of two available processor kernels. We can roughly estimate that this hardware is two times faster than the computer used for testing LM solver. Our code is written in C++ and compiled with Microsoft Visual C++ 2005 compiler.

We followed two test scenarios. We exploited the upper bounds listed in PSPLIB as the start values for the makespan. Each time our solver finds a feasible solution, we decrease the makespan by one and try to determine a better solution or to prove that no solution with this new makespan exists. We consider an instance as solved optimally, if the solver finds a feasible solution and proves that no feasible solution with the makespan decreased by one exists. The same test method and the same upper bounds were used by Liess and Michelon [21]. In the following, we will refer to this scenario as the *good UB scenario*.

In other experiments, the authors did not use any given upper bounds. To make a fair comparison, we also conducted other tests in which the initial upper bounds were set to the best known upper bound from PSPLIB increased by ten percent. This increase is not small: the heuristic of Debels and Vanhoucke [8] provides within the average time of less than 0.2 sec upper bounds that exceed those listed in PSPLIB on average at most by 0.46% for instance sets j30, j60 and j90. In the following, we will refer to this scenario as the *bad UB scenario*.

We expect, that combining our complete solver with a good heuristic in order to determine feasible solutions, would slightly improve the overall results compared to the *good UB scenario*. In this case there would be no need to prove the upper bound determined by the heuristic, which in many cases is the optimal makespan value.

The results of the experiments are summarized in Tables 1 – 4. The tables contain a column for each solver, test scenario, and hardware configuration. Each column contains the percent of the test instances of the respective KSD instance set solved to optimality within the *per instance* time limit indicated in the first column. A dash means that for the corresponding solver and time limit no results have been reported or no experiments have been made.

time limit	this solver	LM	this solver		DPP	KS	DH	CPLEX
sec.	good UB	good UB	bad UB	good UB				good UB
	Core2Duo		Pentium II	Pentium II	Pentium Pro	Pentium	80486	Core2Duo
1	94.0	–	78.5	86.0	80.2	–	–	69.2
10	97.5	–	90.2	94.0	88.3	–	–	74.4
30	98.3	96.0	94.0	96.3	92.7	–	–	77.5
300	100	–	97.3	98.5	95.4	–	–	84.0
360		97.7	97.5	98.5	–	100	–	84.4
600		–	98.5	98.8	–		–	–
1800		–	99.4	99.4	97.3		–	–
3600		98.1	99.8	100	–		99.8	–

Table 1: Percent of j30 instances solved optimally

time limit	this solver	LM	this solver		DPP	KS	Sp	BKST
sec.	good UB	good UB	bad UB	good UB				
	Core2Duo		Pentium II	Pentium II	Pentium Pro	Pentium	Pentium	Sparc 20
1	80.0	–	44.0	73.5	73.5	–	–	–
5	**82.3**	–	75.8	79.6	–	–	–	–
10	83.1	–	77.7	80.0	75.4	69.6	–	–
30	85.2	79.4	79.4	81.0	76.2	–	–	–
60	85.6	–	81.0	**82.9**	–	–	–	–
120	86.9	–	**82.1**	84.4	–	–	–	–
300	88.1	81.2	84.0	85.2	78.5	76.0	72.7	–
1800	89.6	81.9	–	–	80.0	80.2	75.8	–
3600	89.6	82.1	–	–	–	81.9	–	67.9

Table 2: Percent of j60 instances solved optimally

Utilizing the PSPLIB upper bounds (the good UB scenario) our solver needed 1326 seconds on the Intel Core2Duo to solve optimally all instances of j30. Most of the time (87.3%) was spent for proving the optimality and only 12.7% for initializing the solver and determining feasible solutions.

time limit	this solver	LM	this solver		DPP	Sp
sec.	good UB	good UB	bad UB	good UB		
	Core2Duo		Pentium II	Pentium II	Pentium Pro	Pentium
1	78.3	–	2.1	53.1	71.2	–
5	78.6	–	69.4	**76.0**	–	–
10	**79.0**	–	75.0	77.1	74.2	–
30	79.6	78.3	**77.3**	77.5	–	–
60	80.2	–	77.9	79.0	75.0	–
300	81.5	78.5	79.0	79.8	76.0	61.5
1800	82.1	78.8	–	–	–	–
3600	82.5	–	–	–	–	–

Table 3: Percent of j90 instances solved optimally

time limit	this solver	LM	this solver		DPP
sec.	good UB	good UB	bad UB	good UB	
	Core2Duo		Pentium II	Pentium II	Pentium Pro
10	**40.8**	–	24.8	**34.3**	31.0
30	42.5	39.2	32.0	35.7	32.0
60	43.2	–	**33.8**	39.3	–
300	44.7	39.8	41.2	43.2	33.3
1800	46.0	40.0	–	–	–

Table 4: Percent of j120 instances solved optimally

For any instance solved by CPLEX within the time limit of 360 sec our solver needed at most 0.11 seconds. The standard integer programming software seems not to be competitive with any of these solvers on these benchmark instances (at least for our integer programming model).

The solvers KS and DH outperform our solver on this instance set. DH solver needed for its most challenging instance j3013-1 (the only one not solved within an hour and hence not considered in the table) 7209 sec. This instance was solved by our solver within 39 sec on Pentium II processor. No results of applying the DH solver to larger instances have been reported. We can, however, compare the results of our solver on instance set j60 with those of KS solver, see Table 2.

We solved within the time limit of five seconds on the Intel Core2Duo at least as many instances of this set as any other solver did within any reported time limit (half an hour or one hour). The results on the other two instance sets can be found in Tables 3 and 4.

We can not compare the runtimes of the solvers on each particular instance. We can, however, conclude that for each solver each of the sets j60, j90, j120 contains instances solved by our solver within by one to two orders of magnitude shorter equivalent CPU time. Moreover, we solve more instances form each of these sets within the time limit of 10 or more sec. on Pentium II than any other of the solvers does within an equivalent time limit.

Although the runtime on some instances shows considerable fluctuations depending on the solver parameters, the empirical evidence remains valid: there are hard KSD instances, which are hard for any parameter settings and there are easy ones, which are easy for any settings. The solver could solve optimally (find a feasible solution and prove its optimality) all j30 instances within the time limit of 300 sec.

In the second part of our numerical tests we tried to close instances still marked as unclosed in PSPLIB by finding better upper and lower bounds without proving the upper bounds indicated in PSPLIB. The results are presented in Tables 5 and 6. They were obtained on Dell Precision with Intel Core2Duo T6400 2.2GHz with 1 GB of RAM running under Windows XP. The tables contain the instances closed for the first time by our solver or by LM solver. The optimal makespan and the time needed by each of the two solvers is indicated for each instance. A dash means that the instance was not closed by the solver or no result was reported.

Table 5 provides the results for instance set j60. Our solver improved here the upper bound only for instance j609-10 but could not find out if it is optimal even after 20 hours of computation, so the instance remains open. Several instances were closed by proving the tight lower bound. Instance j6014-3 is mistakenly reported by Liess and Michelon [21] to have a lower bound of 62, however, its optimal value is 61 according to PSPLIB. This value was also confirmed by our solver. In total, our solver was able to close 80 of such unclosed instances with 60 activities within a 'soft' time limit of ten hours, while LM solver closed 41 ones. Our solver was slower only for five easy-to-close instances (runtime under 0.1 sec) and much faster for the remaining ones. In total, our solver needed factor 298 less time to close these 41 instances than LM solver did.

Similar results were obtained for instance set j90. Our solver could close within the soft time limit of half an hour 44 instances, which were marked as not closed in PSPLIB. We found that the results reported by Liess and Michelon [21] contain some inconsistencies. The lower bounds indicated for instances j9026-8, j9030-5 and j9030-7 conflict with the upper bounds 82, 83 and 84 listed for them in PSPLIB. These upper bounds are also confirmed by our solver for instances j9026-8 and j9030-5 after 82 sec and 22 min of runtime, correspondingly. Although our solver is slower for some easy instances (runtime under 0.2 sec), it is significantly faster for more challenging ones and does not show any contradictory results. Assuming the correctness of all other results of LM solver we conclude that it closed 24 new instances and compare the results in Table 6. LM solver closed only one instance of set j90 for which our solver needed more than one second.

Our solver found a better feasible solution for one instance of set j60 (runtime 6.6 hours) and for four instances of set j120 (within the runtime of 3.6 sec till 4.4 min). These and many other solutions can be found on the web page of PSPLIB (http://webserver.wi.tum.de/psplib/).

instance	optimal	runtime sec.		instance	optimal	runtime sec.	
	makespan	this solver	LM		makespan	this solver	LM
j601-7	72	0.06	0.01	j6026-4	67	0.11	225.35
j603-8	55	0.03	0.01	j6026-6	74	0.03	0
j605-1	76	9.28	–	j6026-9	65	0.16	98.81
j605-2	106	16.55	–	j6030-3	82	0.02	8.5
j605-3	80	2.13	–	j6030-5	76	155.64	–
j605-4	72	16.67	–	j6030-7	86	12.44	–
j605-5	108	8.28	4539.4	j6030-10	86	95.30	–
j605-6	74	0.25	13.57	j6033-6	75	0.03	1.31
j605-7	75	93.33	–	j6037-1	97	2.28	83.21
j605-8	78	0.09	27.16	j6037-2	95	6.30	7121.78
j605-9	83	0.08	0.00	j6037-3	139	2.44	288.9
j605-10	81	615.33	–	j6037-4	101	0.25	9.67
j606-6	55	0.02	1.77	j6037-5	98	0.08	2.16
j609-2	82	37.09	–	j6037-6	102	155.77	–
j609-3	100	5243	–	j6037-7	110	31.13	172.16
j609-4	87	1.74	–	j6037-8	93	0.19	8.36
j609-8	96	19960	–	j6037-9	96	0.27	34.23
j609-9	99	6228	–	j6037-10	96	0.08	0.09
j6014-1	61	10.75	–	j6038-2	76	0.31	652.27
j6014-10	72	1.41	–	j6038-8	71	0.02	0.03
j6017-8	85	0.09	–	j6038-10	66	0.03	3.38
j6021-1	103	1.02	186.42	j6041-1	122	117.27	–
j6021-2	108	0.16	0.53	j6041-2	113	147.16	–
j6021-3	87	0.22	1.28	j6041-4	133	16.98	2972.55
j6021-4	95	2.56	–	j6041-6	134	592.45	–
j6021-5	89	1.72	456.92	j6041-7	132	22.16	5726.95
j6021-6	84	0.16	18.8	j6041-8	135	200.39	–
j6021-7	103	1.12	9.92	j6041-9	131	92.02	–
j6021-8	110	1.33	46.23	j6042-3	78	1.63	6360.82
j6021-9	89	52.09	–	j6042-4	103	0.09	4.66
j6021-10	80	0.34	3.37	j6042-7	59	0.19	–
j6022-4	73	0.03	14	j6042-8	82	0.10	602.74
j6025-1	114	9499	–	j6046-1	79	0.03	0
j6025-3	113	481.08	–	j6046-4	74	6.44	–
j6025-4	108	50861	–	j6046-5	91	8.72	–
j6025-5	98	11113	–	j6046-6	90	0.05	–
j6025-9	99	446.64	–	j6046-7	78	7.58	–
j6025-10	108	16616	–	j6046-8	75	0.08	78.72
j6026-2	66	0.03	0.33	j6046-9	69	250.31	–
j6026-3	76	0.22	–	j6046-10	88	72.64	–

Table 5: Closing j60 instances: results compared with the reported results of LM solver

instance	optimal makespan	runtime sec. this solver	LM	instance	optimal makespan	runtime sec. this solver	LM
j901-1	73	0.09	0.06	j9026-5	85	395.31	–
j901-3	66	0.09	–	j9033-1	99	0.17	0.45
j901-4	86	0.28	–	j9033-4	92	0.09	0.15
j901-6	74	0.09	8.41	j9033-7	109	0.14	0.05
j901-7	91	0.11	0.03	j9033-8	110	0.17	0.27
j901-8	95	0.24	6.4	j9033-9	95	0.13	–
j901-9	72	0.09	138.5	j9034-5	83	0.06	0.03
j901-10	90	0.13	0.04	j9037-1	110	133.30	–
j905-1	78	4.13	–	j9037-3	132	6.20	–
j905-2	93	5.63	–	j9037-4	123	85.25	–
j906-3	77	0.09	1.08	j9037-5	126	1948.89	–
j906-8	68	0.11	0.00	j9037-7	123	24.31	–
j9017-1	92	0.14	0.03	j9037-9	123	23.64	–
j9017-2	100	0.5	–	j9038-1	85	0.05	0.15
j9017-3	89	0.09	0.02	j9038-3	89	0.06	0.00
j9017-9	96	0.16	0.13	j9038-5	86	0.06	0.02
j9017-10	89	0.17	320.73	j9042-2	102	18.19	–
j9021-3	124	120	–	j9042-7	87	64.70	–
j9021-4	106	11.75	–	j9042-9	83	0.06	0.01
j9021-6	106	1448.08	–	j9042-10	90	58.22	–
j9021-10	109	429.84	–	j9046-1	104	0.09	0.01
j9026-4	97	32.25	0.00	j9046-3	113	0.16	0.01

Table 6: Closing j90 instances: results compared with the reported results of LM solver

7 Further Research

Besides the fact that our solver efficiently finds optimal solutions and proves their optimality, even more important features for its possible practical use are its *generality* and *extensibility*. Without much effort we can implement in our SAT model various kinds of extensions of the RCPSP, for instance:

- different resource availabilities for each period,
- different modes for each activity resulting in different duration and resource demands (Multi-Mode RCPSP),
- maximum time lags between activities,
- restrictions of the type 'only one of two selected activities must be scheduled'.

In our opinion, such extensions (at least the last two) would demand a significant change in any specific heuristic algorithm developed for the RCPSP.

Algorithms hybridizing SAT and linear solvers seem also to be promising for various combinatorial optimization problems. This is a proven fact, that no complete SAT solver

can efficiently deal with several kinds of infeasibilities appearing in combinatorial problems. The pigeonhole problem is only one example of such infeasibility. At the same time, much more general problems can be efficiently solved by a linear solver.

Therefore, it can be useful to combine a SAT solver and an LP solver in one search framework with the SAT solver on the top and the LP solver on the lower levels. The propagate function of the SAT solver should decide if it is worth to call the linear solver for the current partial assignment. The results of the linear solver can then be used in various ways:

- if the linear solver detects infeasibility (or the lower bound obtained by the linear solver is bad), a conflict clause (or a set of clauses) forbidding the current partial assignment are generated and the backtrack is called,
- variable values and dual prices can be used when making branching decision in the SAT solver,
- the learnt clauses can be used to generate cut inequalities for the linear solver.

Appendix: Integer Programming Model

Here we describe the integer programming model for the feasibility version of RCPSP used in our numerical experiments with CPLEX. We use here the notation of Section 2. For each $i \in V$ and for each $t \in \{es_i, \ldots, ls_i\}$ we define a start variable x_{it} such that

$$x_{it} = \begin{cases} 1 & \text{if activity } i \text{ starts at period } t, \\ 0 & \text{otherwise.} \end{cases}$$

For each $i \in V$ and for each $t \in \{es_i, \ldots, lf_i\}$ we define a process variable u_{it} such that

$$u_{it} = \begin{cases} 1 & \text{if activity } i \text{ is in process at period } t, \\ 0 & \text{otherwise.} \end{cases}$$

The following integer linear problem possesses the desirable properties:

$$\begin{aligned}
&\min x_{n+1,T} \\
&\sum_{t \in \{es_i, \ldots, ls_i\}} x_{it} = 1 && i \in V \\
&u_{il} - x_{it} \geq 0 && i \in V,\ t \in \{es_i, \ldots, ls_i\},\ l \in \{t, \ldots, t + d_i - 1\} \\
&\sum_{l \in \{es_j, \ldots, t - d_j\}} x_{jl} \geq x_{it} && (j,i) \in E,\ t \in \{es_i, \ldots, ls_i\} \qquad (5) \\
&\sum_{i \in V, es_i \leq t \leq ls_i} r_{ik} u_{it} \leq R_k && t \in \{0, 1, \ldots, T\},\ k \in R \\
&x_{00} = 1 \\
&x_{it} \in \{0, 1\}
\end{aligned}$$

If one is the optimal value of (5), then T is the optimal makespan for the RCPSP. If the optimal value is zero, $T-1$ is an upper bound for the RCPSP. If (5) has no feasible solution, $T+1$ is a lower bound for the RCPSP.

References

[1] P. Baptiste and S. Demassey. Tight LP bounds for resource constrained project scheduling. *OR Spectrum*, 26:251–262, 2004.

[2] P. Brucker and S. Knust. A linear programming and constraint propagation-based lower bound for the RCPSP. *European Journal of Operational Research*, 127:355–362, 2000.

[3] P. Brucker, S. Knust, A. Schoo, and O. Thiele. A branch and bound algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 107:272–288, 1998.

[4] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch. Resource-constrained project scheduling: notation, classification, models, and methods. *European Journal of Operational Research*, 112:3–41, 1999.

[5] S. A. Cook. *Conf. Record of 3rd STOC*, chapter The complexity of theorem proving procedures, pages 151–158. Shaker Height, OH, May 1971.

[6] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960. ISSN 0004-5411.

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

[8] D. Debels and M. Vanhoucke. A decomposition-based genetic algorithm for the resource-constrained project-scheduling problem. *Operations Research*, 55(3):457–469, 2007.

[9] E. L. Demeulemeester and W. S. Herroelen. New benchmark results for the resource-constrained project scheduling problem. *Management Science*, 43(11):1485–1492, 1997. ISSN 0025-1909.

[10] U. Dorndorf, E. Pesch, and T. Phan-Huy. A branch-and-bound algorithm for the resource-constrained project scheduling problem. *Mathematical Methods of Operations Research*, 52:413 – 439, 2000.

[11] N. Eén and N. Sörensson. *Theory and Applications of Satisfiability Testing*, chapter An extensible SAT-solver, pages 502–518. Springer Berlin / Heidelberg, 2004.

[12] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[13] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. *Handbook of Knowledge Representations*, chapter Satisfiability Solvers, pages 89–132. Elsevier, 2008.

[14] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[15] S. Hartmann and D. Briskorn. A survey of deterministic modeling approaches for project scheduling under resource constraints. Technical Report 2, Hamburg School of Business Administration, 2008.

[16] W. Herroelen. Project scheduling – theory and practice. *Production & Operations Management*, 14(4):413 – 432, 2005.

[17] R. Klein and A. Scholl. Scattered branch and bound: an adaptive search strategy applied to resource-constrained project scheduling. *Central European Journal of Operations Research*, 7:177–201, 1999.

[18] R. Klein and A. Scholl. Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. *European Journal of Operational Research*, 112(2):322–346, January 1999.

[19] R. Kolisch, A. Sprecher, and A. Drexl. Characterization and generation of a general class of resource-constrained project scheduling problems. *Management Science*, 41: 1693–1703, 1995.

[20] L. Levin. Universal sequential search problem. *Problems of Information Transmission*, 9:265–266, 1973.

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

[22] A. Mingozzi, V. Maniezzo, S. Ricciardelli, and L. Bianco. An exact algorithm for the resource-constrained project scheduling problem based on a new mathematical formulation. *Management Science*, 44:714–729, 1998.

[23] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *DAC*, pages 530–535, 2001.

[24] E. Pesch and U. A. W. Teitzlaff. Constraint propagation based scheduling of job shops. *INFORMS Journal on Computing*, 8:144–157, 1996.

[25] A. Sprecher. Scheduling resource-constrained projects competitively at modest memory requirements. *Management Science*, 46:710–723, 2000.