

An Adaptive Large Neighborhood Search Algorithm for the Resource-constrained Project Scheduling Problem

Laurent Flindt Muller*

*Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen, Denmark
laurent@diku.dk

Abstract

We present an application of an *Adaptive Large Neighborhood Search* (ALNS) algorithm to the *Resource-constrained Project Scheduling Problem* (RCPSP). The ALNS framework was first proposed by Pisinger and Røpke [19] and can be described as a large neighborhood search algorithm with an adaptive layer, where a set of destroy/repair neighborhoods compete to modify the current solution in each iteration of the algorithm. Experiments are performed on the well-known J30, J60 and J120 benchmark instances, which show that the proposed algorithm is competitive and confirms the strength of the ALNS framework previously reported for different variants of the *Vehicle Routing Problem*.

1 Introduction

In many situations, such as industrial production and software development, one needs to plan a number of interdependent activities on a scarce number of resources, such that the time to complete all the activities is minimized. These kind of problems can be modelled as a *Resource-constrained Project Scheduling Problem* (RCPSP), which can be described as follows (cf. Brucker et al. [4]): A project consists of a set $\mathcal{A} = \{1, \dots, n\}$ of activities, which must be performed on a set $\mathcal{R} = \{1, \dots, m\}$ of resources. An activity $j \in \mathcal{A}$ requires $r_{jk} \geq 0$ units of resource $k \in \mathcal{R}$ throughout its non-preemptible processing time $p_j \geq 0$. Each resource $k \in \mathcal{R}$ has a limited capacity $R_k \geq 0$. There exists precedence relations between the activities, such that one activity $j \in \mathcal{A}$ can not be started before all its predecessors, P_j , have completed (symmetrically S_j denotes the set of successors). The objective is to find a precedence and resource-capacity feasible schedule which minimizes the makespan.

The RCPSP was first described by Pritsker et al. [20] and as a generalization of the *Job Shop Scheduling Problem* it is \mathcal{NP} -hard (cf. Blażewicz et al [2]). A large number of solutions methods have been applied to the RCPSP, see for instance the surveys by Herroelen et al. [10], Kolisch and Hartmann [13, 12] and Kolisch and Padman [15]. The RCPSP is notoriously hard and only instances with up to 30 activities can consistently be solved to optimality. It is thus of interest to consider heuristics as an alternate approach. Among the most successful heuristics are the (hybrid) genetic

Hamburg, Germany, July 13–16, 2009

algorithms devised by Debels and Vanhoucke [6], Hartmann [9, 8] and Valls et al. [30, 29, 28], the local search algorithms devised by Fleszar and Hindi [7], Kochetov and Stolyar [11] and Palpant et al. [18], the simulated annealing algorithm devised by Boulemain and Lecocq [3], the tabu search algorithms devised by Nanobe and Ibaraki [17] and Valls et al. [27], the sampling based algorithms devised by Tormos and Lova [25, 26] and the scatter search based algorithms devised by Debels et al. [5] and Ranjbar et al. [21].

We propose to solve the RCPSP heuristically by using an *Adaptive Large Neighborhood Search* (ALNS) algorithm, where – exploiting the flexibility of the ALNS framework – we unify techniques from other algorithms proposed for the RCPSP and let the adaptive layer of the algorithm select among the best during execution. The computational experiments show that the algorithm is competitive with state-of-the-art algorithms. To the of best our knowledge, this is the first application of the ALNS framework to the RCPSP.

In Section 2, a general description of the ALNS framework is given, in Section 3, a description of the adaption of the ALNS framework to the RCPSP is given, in Section 4, a description of the different destroy/repair neighborhoods is given, in Section 5, the results of running the algorithm on benchmark instances is presented and finally we conclude in Section 6.

2 Adaptive Large Neighborhood Search

The ALNS framework was first proposed by Pisinger and Røpke [19] for different variants of the *Vehicle Routing Problem*, where good results were reported by Røpke and Pisinger [22, 23] and Pisinger and Røpke [19]. It is a general framework which can be applied to a large class of optimization problems and can be described as follows: ALNS is a local search framework in which a number of simple neighborhoods compete to modify the current solution. In each iteration a *destroy neighborhood* is chosen to destroy the current solution, and an *repair neighborhood* is chosen to repair the solution. The new solution is accepted if it satisfies some criteria defined by the local search framework applied at the master level. The neighborhoods used are typically neighborhoods, who can reach a large part of the solution space. An adaptive layer stochastically controls which neighborhoods to choose based on their past performance (score). The more a neighborhood has contributed to the solution process, the larger score it obtains, and hence it has a larger probability of being chosen. The adaptive layer uses roulette wheel selection for choosing a destroy and repair neighborhood. If the past score of a neighborhood i is π_i and we have ω neighborhoods, then we choose neighborhood j with probability $\pi_j / \sum_{i=1}^{\omega} \pi_i$. ALNS can be based on any local search framework, e.g., simulated annealing, tabu search or guided local search. For a more detailed description we refer to this paper by Pisinger and Røpke [19].

3 Algorithm

In the following we give a presentation of the different components of the proposed algorithm.

Representation There exists a number of different solution representations for the RCPSP (cf. Kolisch and Hartmann [14]). The proposed algorithm employs list representation, where a schedule

Hamburg, Germany, July 13–16, 2009

is represented as an precedence-ordered list of activities, i.e., if $i \in P_j$ then i comes before j in the list. When using list representation, one additionally needs a scheme for converting the list into a schedule. The two most commonly used are the serial and parallel schedule generation schemes (SGS) (cf. Kolisch and Hartmann [14]). The serial SGS can be described as follows: in the order defined by the list, schedule each activity in turn, at the earliest precedence and resource feasible point in time. The parallel SGS can be described as follows: Time is incremented starting at zero. At each time the unscheduled precedence and resource feasible activities are scheduled in the order defined by the list. When no more activities can be scheduled, the time is incremented. It has been shown by Sprecher et al. [24] that the serial SGS produces so-called *active* schedules, that parallel SGS produces so-called *non-delay* schedules, that an optimal solution exists within the set of active schedules but not necessarily within the set of non-delay schedules and that the list representation (with either parallel or serial SGS) is not unique, i.e., two different list may represent the same schedule. Based on experiments the serial SGS is used for the proposed algorithm.

Precedence augmentation In the variable neighborhood search algorithm proposed by Fleszar and Hindi [7] the concept of *precedence augmentation* is introduced. It can be described as follows: Let the *head*, h_j , of an activity $j \in \mathcal{A}$ be the time that must pass before activity j can be started and let the *tail*, t_j , be the time that must pass from the completion time of activity j until the project can be completed. The process of precedence augmentation is the process of permanently adding new precedence relations based on heads, tails and an upper bound on the current problem, such that (not necessarily all) solutions which are not better than the upper bound are rendered infeasible. This has the effect of narrowing the search space. We employ two of the precedence augmentation rules described by Fleszar and Hindi [7]: Assume that a new better solution with makespan T has been found. From this point on, only solutions with a makespan of at most $UB = T - 1$ are interesting. Consider all pairs of activities $i, j \in \mathcal{A}$ which are not in any precedence relation (direct or indirect). A new precedence relation from i to j is added if one of the following holds:

1. $h_j + t_i \geq UB$
2. $\exists k \in \mathcal{R} : r_{ik} + r_{jk} > R_k \wedge h_j + p_j + p_i + t_i > UB$.

When new precedence rules are added, the current solution may become infeasible and needs to be repaired before the search can go on. This may results in the repaired solution having a worse makespan than the solution on the basis of which precedence relations were added, which is inconvenient since the search will continue from this worse solution. It is thus worthwhile to spend some time repairing the solution, such that it is at least as good as the original one. To this end Fleszar and Hindi [7] construct a special repair algorithm. We take a different approach and use the repair neighborhoods already part of the algorithm. The activities to be reinserted are the ones which now violate the precedence-ordering of the activity list. Each repair neighborhood is given a chance to repair the solution until either a solution which is at least as good as the original is found or there are no neighborhoods left, in which case the repaired solution with the best makespan is used. Each time a new best solution is found during execution of the algorithm, precedence augmentation is performed.

Double justification Valls et al. [30] shows that a simple technique denoted *justification* can improving the quality of a solution with little extra computational effort. One speaks of *left-*

justification, *right-justification* and *double-justification*. Left-justification is essentially pulling all activities of a schedule as far towards time zero (left) as possible, while right-justification is essentially pulling all activities as far towards the time corresponding to the makespan (right) as possible. Double-justification is doing first a right-justification followed by a left-justification. Often a double-justified schedule is better than the original one. Since this is a simple technique, which have been shown to produce good results, all schedules produced during the course of the ALNS algorithm are double-justified. This means that in each iteration at least three schedules are generated (more may be generated if the current solution must be repaired after precedence augmentation).

Master level search framework As mentioned earlier one needs to select a local search framework at the master level. After some experimenting the choice fell on one with the following properties: In each iteration a destroy and repair neighborhood is selected based on the current scores. Given the parameter Q , which governs how large a part of the solution should be destroyed, a new solution is created. Only solutions which are as good as, or better than the current solution is accepted. A tabu list is maintained such that the same activity list is not visited twice. The value Q is progressively reduced from its initial value toward a final value, Q_{end} , such that if Q_i is the value of Q in the i -th iteration then $Q_i = \max\{c^i \cdot Q, Q_{end}\}$, where $c \in [0; 1]$. This has the effect that the algorithm will initially look at large neighborhoods, but these get progressively reduced as the search progresses to good solutions. This will result in a diversified search in the beginning and an intensified search at the end.

Scoring scheme As part of an ALNS algorithm, a scoring scheme needs to be chosen. As in the paper by Pisinger and Røpke [19] the scores are updated at certain intervals rather than in each iteration. Thus scoring information is collected during a certain number of iterations before the scores of each neighborhood is updated and the collection restarts. We call the number of iterations which must pass between each score update the *score interval*. Let π_j be the current score and $\bar{\pi}_j$ the scoring information collected during the last scoring interval, then the score π_j is updated as follows: $\pi_j = \max\{\rho \cdot \frac{\bar{\pi}_j}{a_j} + (1 - \rho) \cdot \pi_j, \pi_{min}\}$, where a_j is the number of times the neighborhood has been chosen during the last score interval (if $a_j = 0$, the score is unchanged), ρ is the *score reaction* and π_{min} is the *minimum score*.

Let T be the makespan of the current solution and T' the makespan of the current candidate created by applying the destroy and repair neighborhood N^+ and N^- . Let $\bar{\pi}_j$ be the collected score so far for N^+ (the procedure is equivalent for N^-), $\bar{\pi}_j$ is updated as follows: $\bar{\pi}_j = \bar{\pi}_j + b^{(T-T')/T}$, where b is some real number, experimentally chosen to 7. This scoring scheme differs from the one used by Pisinger and Røpke [19], where one of three fixed scores are attributed depending on whether the current solution was improved globally, locally or a worse solution was accepted. The reason for this difference is that the proposed algorithm only accepts equal or better solutions, which can results in many iterations where there is no improvement at all. If a fixed scoring scheme was used all neighborhoods would score equally bad, while for the employed scheme it is possible to differentiate the neighborhoods who produce solutions which are (almost) as good as the current and the ones that produce solutions which are far worse.

Hamburg, Germany, July 13–16, 2009

4 Neighborhoods

An important part of an ALNS algorithm is the destroy and repair neighborhoods. Even though there is an adaptive layer, one should remain careful about adding too many neighborhoods, especially when only a limited number of iterations is allowed. The reason is that a number of iterations will be needed before any poorly performing neighborhoods will have been filtered out and these neighborhoods will end up taking time from the good ones. It is also important to have a good mix of neighborhoods, which are good at search intensification and diversification. For the proposed algorithm the neighborhoods have been selected by running the algorithm with all conceived neighborhoods enabled, then in turn each neighborhood was disabled, if this resulted in a better solution, the neighborhood was permanently removed. In the following we describe the destroy and repair neighborhoods employed.

Destroy neighborhoods Given the parameter Q and an activity list, L , a destroy neighborhood must remove Q activities from L . All destroy neighborhoods share the same structure: For $j \in \mathcal{A}$ the *predecessor-cluster* of j is defined as $C^p(j) = \{i \in \mathcal{A} | i \in P_j \vee s_i + p_i = s_j\}$, and the *cluster* of j as $C^c(j) = \{i \in \mathcal{A} | i \in C^p(j) \vee i \in S_j \vee s_j + p_j = s_i\}$, where s_j denotes the starting time of activity j . Let $p : \mathcal{A} \rightarrow \{0, 1\}$ be some predicate, then a *core removal candidate set*, C , is constructed as $C = \{j \in \mathcal{A} | p(j) = 1\}$. C is given some ordering on the basis of which each activity j from C is removed from the list along with possibly elements from either $C^p(j)$ or $C^c(j)$ (depending on the neighborhood) until either the set C is empty or Q elements have been removed from L . The idea behind clusters is that one wants as much flexibility as possible for the repair neighborhood, e.g., if all the predecessors and successors of an activity are left in place there is potentially little room for inserting the activity in new positions. There are in total 10 destroy neighborhoods, which are described below.

- **random** This neighborhood ensures diversification by randomly removing Q activities from the current solution. It comes in two flavors, one where predecessor-clustering is used and one where clustering is used.
- **most-mobile** Let $j \in \mathcal{A}$. As Fleszar and Hindi [7], we define the *left limit* $LL(j)$ and the *right limit* $RL(j)$ as $LL(j) = \max\{\gamma_i | i \in P_j\} + 1$ and $RL(j) = \min\{\gamma_i | i \in S_j\} - 1$, where γ_i is the position of i within the activity list. Now the *mobility*, $m(j)$, of j is defined as $m(j) = RL(j) - LL(j)$.

This neighborhood selects the Q activities with the highest mobility from the current activity list and also ensures diversification but in a different way than the one above. It ensures that the neighborhood explored is large by selecting activities which have many reinsertion possibilities.

- **non-peak** In the hybrid genetic algorithm proposed by Valls et al. [28] the peak crossover operator employed passes on to its children the parts of the schedules with high utilization, so-called *peaks*. Similarly we define a *non-peak* predicate. A peak is defined in the same way as by Valls et al.: Let S be the current schedules at let $S(t) = \{j \in \mathcal{A} | s_j \leq t \wedge t \leq s_j + p_j\}$, where s_j is the starting time of activity j and t is some time instant. We define the *Resource*

Utilization Ratio as follows

$$RUR(t) = \frac{1}{m} \cdot \sum_{j \in S(t)} \sum_{k=1}^m \frac{r_{jk}}{R_k}$$

Given some $\delta \in [0; 1]$ we say that an time instant t is of *high utilization* if $RUR(t) \geq \delta$. Similarly we say that a time interval I is of *high utilization* if $\forall t \in I : RUR(t) \geq \delta$. Let \mathcal{I} be the set of disjunctive maximal intervals of high utilization for the schedule S , then a *peak activity* $j \in \mathcal{A}$ is an activity which satisfies $\exists I \in \mathcal{I} : [s_j; s_j + p_j] \cap I \neq \emptyset$, i.e., all activities which are active during some interval of high utilization. A *non-peak activity* is an activity which is not a peak activity. The non-peak predicate selects all activities which are non-peak activities.

This neighborhood uses the non-peak predicate and tries to preserve the structure of the solution where the utilization is good, and destroy the parts where it is not. The neighborhood comes in two flavors one where predecessor-clustering is used and one where clustering is used. In both cases the activities are chosen at random from the removal candidate set.

- **critical-path** Given the current schedule S we construct a weighted directed graph $G = (\mathcal{A}, E)$, where $E = \{(i, j) \in \mathcal{A} \times \mathcal{A} | s_i + p_i = s_j\}$ and the weight of a vertex j is p_j . Since the schedule S contains no point in time (before the end) where no activity is scheduled, there must exist at least one path p with weight equal to the makespan of S . In order to improve the makespan, at least one of the activities on this path must be moved elsewhere. The *critical-path* predicate selects all activities which are part of a critical path.

Let $j \in \mathcal{A}$, we define the *volume*, $v(j)$, of j as $v(j) = p_j \cdot \prod_{r \in \{r_{jk} | r_{jk} > 0, k \in \mathcal{R}\}} r$. The *largest-vol* (*smallest-vol*) ordering is the ordering, where the removal candidate set is sorted non-decreasingly (non-increasingly) w.r.t. volume.

This neighborhood uses the critical-path predicate to break the critical path of the current schedule. Three orderings of the removal candidate set are used: **largest-vol**) this ensures that a big part of the critical path is destroyed by removing the activities with the biggest volume **smallest-vol**) this ensures a certain intensification by removing the activities with the smallest volume, which should be easy to insert in other locations **random**) activities are picked at random. For the largest-vol and smallest-vol orderings only predecessor-clustering is used, while both predecessor-clustering and clustering is employed for the random ordering.

- **segment** This neighborhood ensures a certain intensification by selecting a subsequence of length Q from the activity list. This subsequence corresponds to a sub-schedule, which can hopefully be improved.

Repair neighborhoods Given a partial activity list and a set of activities to be reinserted a repair neighborhood must construct a new precedence ordered activity list, which will hopefully lead to a better solution. Again each repair neighborhood shares some structure: Given an ordering of the set of activities to be reinserted, the activities one at a time are inserted into the activity list in such a way that the partial activity list is still precedence ordered. A repair neighborhood can be seen as a pairing between an insertion algorithm and an ordering.

For the variable neighborhood search algorithm proposed by Fleszar and Hindi [7] a possible move for an activity $j \in \mathcal{A}$ is defined as any positions in the interval defined by $LL(j)$ and $RL(j)$.

Hamburg, Germany, July 13–16, 2009

Similarly we define a *random-insert* algorithm which reinserts each activity randomly within this interval.

There are in total 11 repair neighborhoods, where the difference between each neighborhood is the order in which the random-insert algorithm is applied to the activities to be reinserted. For each of the the following well-known priority-rules for the RCPSP (cf. Kolisch and Hartmann [14]) there is an equivalent repair neighborhood : *Shortest processing time (SPT)*, *most total successors (MTS)*, *earliest start time (EST)*, *minimum latest finish time (LFT)*, *minimum slack (MSLK)*, *greatest rank positional weight (GRPW)* and *minimum latest start time (LST)*. These neighborhoods ensure that the solution will be a (hopefully) good mix of these priority rules and is similar to multi-pass methods, where the priority rule is changed between passes. The remaining 4 neighborhoods use the following orderings: *random*, *largest-vol*, *smallest-vol* and *reverse*, where the reverse ordering reverses the order of the activities compared to the current activity list.

5 Computational results

In order to evaluate the performance of the proposed algorithm, tests have been run on the well-known benchmarks, J30, J60 and J120 created by Kolisch and Sprecher [16], which contain respectively 480, 480 and 600 instances. The algorithm has been coded in C++ and the tests have been run on a PC with an Intel Core i7 920 @ 2.67 Ghz. To be able to compare the algorithm with other algorithms we have used the same maximum schedule counts as the one used in a recent survey paper by Kolisch and Hartmann [12], that is 1,000, 5,000 and 50,000. The algorithms from the survey paper fall into two categories: Algorithms where it makes sense to count the number of schedules generated and algorithms where it does not (such as methods based on implicit enumeration). The proposed algorithm falls into the first category, since in each iteration the schedule corresponding to the current solution is destroyed and a new schedule is generated from a partial one by the repair neighborhood (actually 3 schedules, since double justification is employed). We therefore only compare our algorithm to algorithms within the first category and as commonly done for the RCPSP, use the quality of the solution as the measure of performance rather than running time. As a base of comparison we choose the 5 best algorithms from the survey paper along with two recently proposed algorithms (marked with a † in Table 1). Each test run has been repeated 10 times and the average taken. The following parameter values have been employed: score interval = 5, score reaction = 0.2, $Q = 10\%$ (for J120), $Q = 40\%$ (for J30 and J60). The parameter c is set such that Q reaches a value corresponding to the removal of 1 activity after the maximum allowed number of iterations.

Table 1 shows the critical path average deviation (smaller is better) for the different benchmark instances (for J30 it is the average deviation from the optimal solution). As can be seen, the algorithm is competitive, though it is not the best on any of the benchmark instances. Promisingly it ranks better as the instances gets larger (and harder), that is 5th for J30 benchmark instances, 4th for J60 benchmark instances and 3rd for J120 benchmark instances. If only 1,000 schedules is considered the algorithm actually ranks 2nd on both the J60 and the J120 benchmark instances, which indicates that the algorithm does not fully take advantage of the additional iterations.

Table 1: Average deviation from optimal makespan (%) – J30, average deviation from critical path lower bound (%) – J60 and J120

Benchmark	Reference	Algorithm	max. #schedules		
			1,000	5,000	50,000
J30	Ranjbar et al. [21] [†]	SS	0.10	0.03	0.00
	Kochetov and Stolyar [11]	GA, TS	0.10	0.04	0.00
	Debels et al. [5] [†]	SS	0.10	0.04	0.00
	Valls et al. [28]	GA	0.27	0.06	0.02
	ALNS	ALNS	0.18	0.07	0.02
	Alcaraz and Maroto[1]	GA	0.33	0.12	-
	Valls et al. [30]	GA	0.34	0.20	0.02
	Tormos and Lova [25]	sampling	0.25	0.13	0.05
J60	Ranjbar et al. [21] [†]	SS	11.59	11.07	10.64
	Debels et al. [5] [†]	SS	11.73	11.10	10.71
	Valls et al. [28]	GA	11.56	11.10	10.73
	ALNS	ALNS	11.58	11.12	10.73
	Kochetov and Stolyar [11]	GA, TS	11.71	11.17	10.74
	Valls et al. [30]	GA	12.21	11.27	10.74
	Hartmann [8]	GA	12.21	11.70	11.21
	Hartmann [9]	GA	12.68	11.89	11.23
J120	Valls et al. [28]	GA	34.07	32.54	31.24
	Ranjbar et al. [21] [†]	SS	35.08	33.24	31.49
	ALNS	ALNS	34.35	32.91	31.54
	Debels et al. [5] [†]	SS	35.22	33.10	31.57
	Valls et al. [29]	GA	35.39	33.24	31.58
	Kochetov and Stolyar [11]	GA, TS	34.74	33.36	32.06
	Valls et al. [29]	GA	35.18	34.02	32.81
	Hartmann [8]	GA	37.19	35.39	33.21

6 Conclusion

An application of the ALNS framework to the RCPSP has been presented, where a number of techniques from other algorithms are unified within the ALNS framework. Computational results from running the algorithm on the J30, J60 and J120 benchmark instances show that the algorithm is competitive with the state-of-the-art and confirms the strength of the ALNS framework. Interestingly the proposed algorithm is the only non-population based algorithm to rank within the top-5 algorithms and as such represents promising alternative approach to the usual population based algorithms. Another encouraging sign is that the algorithm ranks better on the larger and more difficult instances.

Hamburg, Germany, July 13–16, 2009

References

- [1] J. Alcaraz, C. Maroto A robust genetic algorithm for resource allocation in project scheduling. *Annals of Operations Research*, 102: 83–109, 2001.
- [2] J. Blažewicz, J. Lenstra, A.R. Kan Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics* 5: 11–24, 1983.
- [3] K. Bouleimen, H. Lecocq A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *European Journal of Operational Research*, 149(2): 268–281, 2003.
- [4] P. Brucker, A. Drexler, R. Möhring, K. Neumann and E. Pesch Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1): 3–41, 1999.
- [5] D. Debels, B.D. Reyck, R. Leus, M. Vanhoucke A hybrid scatter search/electromagnetism meta-heuristic for project scheduling. *European Journal of Operational Research*, 169(2): 638–653, 2006.
- [6] D. Debels, M. Vanhoucke A Decomposition-Based Genetic Algorithm for the Resource-Constrained Project-Scheduling Problem. *Operations Research*, 55(4): 457–469, 2007.
- [7] K. Fleszar, K.S. Hindi Solving the resource-constrained project scheduling problem by a variable neighbourhood search. *European Journal of Operational Research*, 155(2): 402–413, 2004.
- [8] S. Hartmann A self-adapting genetic algorithm for project scheduling under resource constraints. *Naval Research Logistics* 49: 433–448, 2002.
- [9] S. Hartmann A competitive genetic algorithm for resource-constrained project scheduling. *Naval Research Logistics*, 45: 733–750, 1998.
- [10] W. Herroelen, E. Demeulemeester, B.D. Reyck Resource-constrained project scheduling - a survey of recent developments. *Computers & Operations Research* 29(4): 279–302, 1998.
- [11] Y. Kochetov, A. Stolyar Evolutionary local search with variable neighborhood for the resource constrained project scheduling problem. In *Proceedings of the 3rd International Workshop of Computer Science and Information Technologies*, 2003.
- [12] R. Kolisch, S. Hartmann Experimental investigation of heuristics for resourceconstrained project scheduling: An update. *European Journal of Operational Research*, 174(1): 23–37, 2006.
- [13] R. Kolisch, S. Hartmann Experimental investigation of heuristics for resourceconstrained project scheduling. *European Journal of Operational Research*, 127: 394–407, 2000.
- [14] R. Kolisch, S. Hartmann Heuristic algorithms for solving the resource-constrained project scheduling problem: Classification and computational analysis. In J. Weglarz, editors, *Project Scheduling, Recent Models, Algorithms and Applications*, pages 147–178, Kluwer Academic Publishers, Boston, 1999.
- [15] R. Kolisch, R. Padman An integrated survey of deterministic project scheduling. *Omega* 29(3): 249–272, 2001.

- [16] R. Kolisch, A. Sprecher PSPLIB – A project scheduling library. *European Journal of Operational Research*, 96: 205–216, 1997.
- [17] K. Nonobe, T. Ibaraki Formulation and tabu search algorithm for the resource constrained project scheduling problem. In C.C. Ribeiro, P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 557–588, Kluwer Academic Publishers, 2002.
- [18] M. Palpant, C. Artigues, P. Michelon LSSPER: Solving the resource-constrained project scheduling problem with large neighbourhood search. *Annals of Operations Research*, 131(1-4): 237-257, 2004.
- [19] D. Pisinger and S. Røpke A general heuristic for vehicle routing problems. *Computers & Operations Research*, 34(8): 2403–2435, 2007.
- [20] A.A.B. Pritsker, L.J. Watters and P.M. Wolfe Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science* 16(Sep.): 93-108, 1969.
- [21] M. Ranjbar, B. De Reyck, F. Kianfar A hybrid scatter search for the discrete time/resource trade-off problem in project scheduling *European Journal of Operational Research*, 193(1): 35–48, 2009.
- [22] S. Røpke and D. Pisinger. A unified heuristic for a large class of vehicle routing problems with backhauls. *European Journal of Operational Research*, 171(3): 750–775, 2006.
- [23] S. Røpke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4): 455–472, 2006.
- [24] A. Sprecher, R. Kolisch and A. Drexl Semi-active, active and non-delay schedules for the resource-constrained project scheduling problem *European Journal of Operational Research*, 80: 94–102, 1995.
- [25] P. Tormos, A. Lova Integrating heuristics for resource constrained project scheduling: One step forward. Technical Report, Department of Statistics and Operations Research, Universidad Politcnica de Valencia, 2003.
- [26] P. Tormos, A. Lova An efficient multi-pass heuristic for project scheduling with constrained resources. *International Journal of Production Research*, 41: 1071–1086, 2003.
- [27] V. Valls, S. Quintanilla, F. Ballestín Resource-constrained project scheduling: A critical activity reordering heuristic. *European Journal of Operational Research*, 149(2): 282-301, 2003.
- [28] V. Valls, F. Ballestín, S. Quintanilla A hybrid genetic algorithm for the resourceconstrained project scheduling problem. *European Journal of Operational Research*, 185(2): 495–508, 2008.
- [29] V. Valls, F. Ballestín, S. Quintanilla Justification and RCPSP: A technique that pays. *European Journal of Operational Research*, 165: 375–386, 2005.
- [30] V. Valls, F. Ballestín, S. Quintanilla A population-based approach to the resource-constrained project scheduling problem. *Annals of Operations Research*, 131: 304–324, 2004.