Profit-oriented scheduling of resource-constrained projects with flexible capacity constraints

André Schnabel, Carolin Kellenbrink[a], Stefan Helber

Institute of Production Management
Leibniz Universität Hannover
Königsworther Platz 1, D-30167 Hannover, Germany

andre.schnabel@prod.uni-hannover.de, +49 511 7622538
carolin.kellenbrink@prod.uni-hannover.de, +49 511 76214468
stefan.helber@prod.uni-hannover.de, +49 511 7625650

[a] *Corresponding author*

Abstract

We consider a novel generalization of the resource-constrained project scheduling problem (RCPSP). Unlike many established approaches for the RCPSP that aim to minimize the makespan of the project for given static capacity constraints, we consider the important real-life aspect that capacity constraints can often be systematically modified by temporarily assigning costly additional production resources or using overtime. We furthermore assume that the revenue of the project decreases as its makespan increases and try to find a schedule with a profit-maximizing makespan. Like the RCPSP, the problem is $\mathcal{NP}$-hard, but unlike the RCPSP it turns out that an optimal schedule does not have to be among the set of so-called active schedules. Scheduling such a project is a formidable task, both from a practical and a theoretical perspective. We develop, describe, and evaluate alternative solution encodings and schedule decoding mechanisms to solve this problem within a genetic algorithm framework and we compare them to both optimal reference values and the results of a commercial local search solver called LocalSolver.

Keywords: Project scheduling, encodings, heuristics, local-search, genetic algorithm, RCPSP, overtime

1. Introduction

Many models and procedures for resource-constrained project scheduling problems (RCPSPs) assume that the capacities of the renewable resources that are required to perform the project's activities are exogenously given and that the objective is to find a (feasible) schedule with a minimal project makespan or duration. In reality, however, the renewable resources like human labor or machinery are often temporarily assigned to a project and decisions on additional resources or overtime are made in order to achieve a short project duration. A short project duration may be economically attractive for different reasons. Consider, for example, software development projects or construction projects for factories. In such cases, a shorter project duration may permit an earlier market entry. This desire to achieve a short project duration can, e.g., lead to contractual penalty clauses or other incentive schemes that relate actual payments to project durations. In such cases, the revenue from a project typically decreases as its duration increases. This immediately leads to the question how to use overtime and how to schedule such projects with flexible capacity constraints and makespan-dependent revenues in the most profitable way.

The remainder of this paper is organized as follows: In Section 2, we describe the assumptions of the resource-constrained project scheduling problem with makespan-specific revenues and option of overcapacity (RCPSP-ROC), give another real-world example, demonstrate basic problem and solution properties, and provide an overview of the related literature. In Section 3, we develop a formal mathematical decision model for the RCPSP-ROC and discuss properties of solutions that guide the development of solution procedures. The design rationales and detailed descriptions of different solution encodings for this problem are given in Section 4.2. On this basis, we propose various genetic algorithms and local search procedures in Sections 4.3 and 4.4. Section 5 is devoted to the test design and the results from a numerical study to evaluate the different proposed methods to solve the RCPSP-ROC. Section 6 concludes this paper by giving a short summary of the results and suggestions for future research.

2. Problem and literature

2.1. Projects with flexible capacity constraints, makespan-dependent revenues and overtime cost

The problem studied in this paper is a modification and extension of the well-established RCPSP, cf., e.g. Pritsker et al. (1969). In a project, there are given activities $j \in \mathcal{J} = \{0, 1, \ldots, J, J+1\}$ that have to be executed in order to complete the project. Activities 0 and $J+1$ denote dummy start and end activities, respectively. Each activity has to be executed exactly once. Between activities, there can be precedence restrictions preventing an activity j to start unless all its predecessor activities $i \in \mathcal{P}_j$ have been completed.

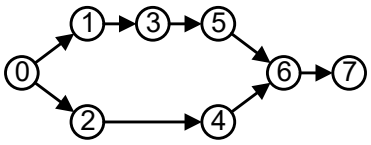


Figure 1: Activity on node graph

Activities and precedence relations can be visualized as an activity-on-node network like the one depicted in Figure 1.

Table 1: Activity durations and capacity requirements

Activity j	0	1	2	3	4	5	6	7
Duration d_j	0	3	2	2	3	1	2	0
Capacity requirement $k_{j,1}$	0	3	2	2	1	2	1	0

During the duration d_j of activity j, it requires k_{jr} units of a renewable resource r, see the data in Table 1 for the example project in Figure 1 using a single renewable resource $r = 1$.

A feasible schedule for such a project is completely characterized by the starting times ST_j and corresponding finishing times FT_j of all activities j so that all precedence constraints are respected and that the project is feasible with respect to the capacities of the resources required to perform these activities. The capacity K_r of resource r is often assumed to be exogenously given and constant over time, and one seeks a schedule that minimizes the project duration or makespan FT_{J+1}.

We extend this well-known problem setting by adding the possibility to use overtime capacity z_{rt} at resource r in period t, up to a limit $\overline{z}_r$, i.e., $z_{rt} \leq \overline{z}_r$ in all periods, at a cost of κ_r monetary units per period and capacity unit of overtime. If in some periods overtime z_{rt} is used, it may be possible to perform activities in parallel that would have to be scheduled sequentially if no overtime capacities were available. If overtime is used, it may hence be possible to achieve a shorter project makespan.

We now further assume that a project's economic value depends on the project's makespan. This value is reflected, for example, by a customer's willingness to pay. In particular, we assume that it is non-increasing as the makespan increases. In Table 2, we show the values for three such hypothetical customers for the project in Figure 1. This directly leads to the question how to schedule the project in a profit-maximizing way and how to use overtime most efficiently, taking the individual customer's perspective and sensitivities into account.

For the example project in Figure 1, we assume that the capacity of the single renewable resource amounts to $K_1 = 4$ units and that it is possible to use $\overline{z}_1 = 2$ units of overtime per period at an overtime cost of $\kappa_1 = 10$ monetary units per period and unit of overtime. For customer A in Table 2, the schedule presented in Figure 2 with a makespan of 10 periods and no overtime is profit-maximizing.

Table 2: Makespan-dependent willingness to pay (revenue) of different customers

Makespan	<8	8	9	10	11	>11
Customer A	10	10	10	10	10	0
Customer B	60	45	30	15	0	0
Customer C	20	20	15	0	0	0

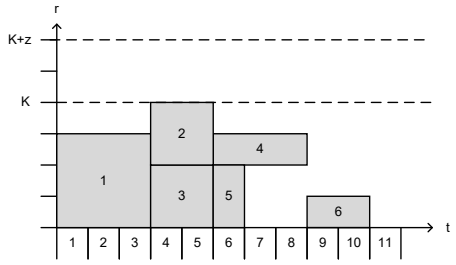


Figure 2: Optimal schedule without overtime usage for example customer A

Note that in this special case of customer A our problem setting bears a resemblance to the resource overload problem with "total overload cost function" as stated in Rieck & Zimmermann (2015, p. 364), because in this special case the objective is effectively reduced be to minimize the cost of overtime for a given deadline δ, which is shown to be $\mathcal{NP}$-hard by Neumann et al. (2003, p. 242).

If, by contrast, there are more strongly decreasing revenues as for the time-sensitive customer B, then the schedule in Figure 3 with a duration of only 8 periods is profit-maximizing. We use two units of the comparatively cheap overtime, but are able to decrease the makespan by two periods and therefore increase the revenue by 30 units and the profit from 15 to 25 units compared to the schedule in Figure 2.

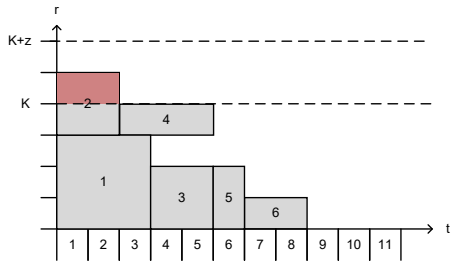


Figure 3: Schedule with maximum amount of overtime utilized for example project

For Customer C, it is neither optimal to minimize overtime as for Customer A nor to minimize the makespan as for Customer B. Instead, in this case the schedule in Figure 4 with a makespan of 9 periods, 1 unit of overtime and a resulting profit of 5 units is profit-maximizing.

In general, let $\underline{T}$ denote the shortest possible makespan making potentially ample use of overtime irrespective of overtime cost but within overtime bounds $\overline{z}_r$. In a similar way, let $\overline{T}$ denote the shortest possible makespan using only the regular capacity K_r, i.e., without any use of overtime. Then the potentially optimal, i.e., profit-maximizing, project

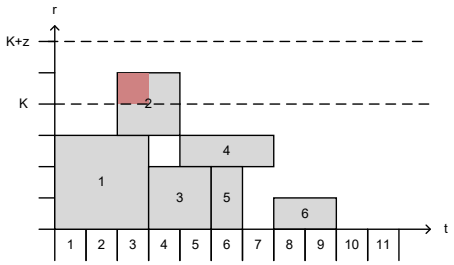


Figure 4: Schedule with some overtime usage for example project

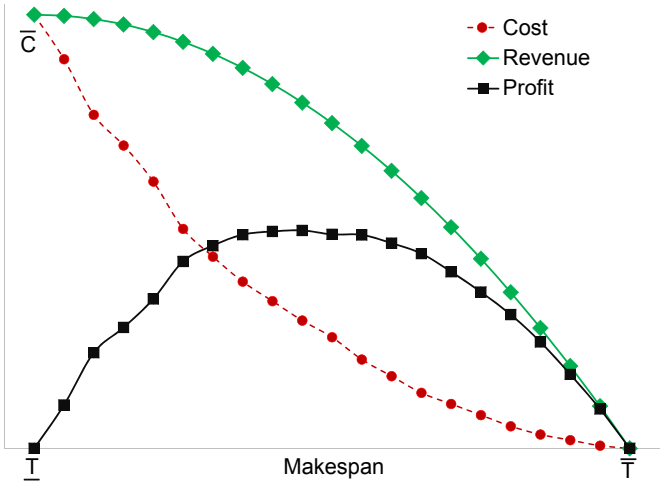


Figure 5: Relationship between makespan, overtime cost, revenue and profit

durations should lie in the time interval $[\underline{T}, \overline{T}]$, since a makespan below $\underline{T}$ is impossible and a makespan exceeding $\overline{T}$ unnecessarily leads to possibly decreasing revenues. In other words, we assume that the cost- and revenue structures are roughly as shown in Figure 5 in order to lead to a non-trivial problem.

Typical real-world examples for projects with these features include aircraft engine remanufacturing projects undertaken by a service contractor, software development projects, and construction projects. Jet engines of commercial aircraft are extremely valuable and durable goods that are routinely overhauled and remanufactured, often after significant wear and tear. Aircraft engine remanufacturing is executed by independent service contractors or original equipment manufacturers. In either case, this complex process typically has a project character, see Kellenbrink & Helber (2016) and Kellenbrink & Helber (2015), as the state of the engines as well as the chosen repair or replace options differ from case to case. The customers are typically airlines that are interested in short remanufacturing processes. For them, it is neither attractive to operate with a large number of reserve engines nor to reduce flight operations due to lengthy engine overhaul processes. The service provider may hence use overtime to increase his capacity for these overhaul processes. Similar situations can exist in software development projects when additional (freelance) programmers are temporarily hired to speed up software development processes. In construction projects, it is not unusual that companies obtain additional

capacities by temporal hiring additional manpower or by renting additional machinery to speed up projects. In many of these cases, the decision maker faces the fundamental problem outlined above to use these additional resources in the most profitable way. However, as far as we know, there is no solution approach available dealing with this problem setting in a systematic way.

2.2. Related literature

The problem described in Section 2.1 bears similarities to the well-known and widely researched single-mode single-project RCPSP without preemption, shortly characterized by $PS|prec|C_{max}$ using the notation introduced by Brucker et al. (1999). Recent overviews of the RCPSP and its extensions were given by Hartmann & Briskorn (2010) and Demeulemeester & Herroelen (2006). Additional literature surveying the state of the art in RCPSP research was published by Kolisch & Padman (2001), Brucker et al. (1999), Herroelen et al. (1997) and Özdamar & Ulusoy (1995). Kolisch & Hartmann (2006) evaluated and differentiated various heuristic solution approaches for the standard RCPSP. Artigues et al. (2015) list a plethora of different previously proposed linear programming formulations for the RCPSP and compare them from a theoretical and experimental perspective.

The most important difference between the standard RCPSP and the variant treated in this paper is the objective function. An objective function is *regular*, if and only if it is monotonically non-decreasing in the activity starting times, cf., e.g., Brucker & Knust (2012, p. 12). Minimizing the makespan is an example of such a regular objective function for the RCPSP. A detailed description and analysis of different objective functions for RCPSPs can be found in the extensive scheduling fundamentals book by Schwindt (2005).

In our paper, we consider a profit objective in which the project's profit is the difference between the makespan-dependent revenue and the associated overtime cost. The objective function of this problem is a linear combination of both a regular part and a non-regular part. Minimizing the makespan is equivalent to maximizing the revenue, since the revenue function is assumed to be monotonically decreasing in the makespan. For any strictly decreasing revenue u_t, revenue maximization even matches the makespan minimization objective. Hence, this part is a regular function. However, minimizing overtime cost is a non-regular objective, since decreasing the overtime typically leads to longer project durations as fewer activities can be performed in parallel.

Focusing on this behavior of the revenue and the cost function, two decomposition approaches involving different RCPSP aspects known from the literature seem to suggest themselves. On the one hand, the makespan can be minimized for the (potenially extremely large) set of all possible fixed overtime profiles. The remaining problems of revenue maximization are similar to many makespan-oriented objective functions, if we do not consider the possible difference between the given overtime profile and the actually used overtime in the resulting schedule. (This difference may lead to an overestimation of the overtime cost.) These subproblems are equivalent to the RCPSP with time-varying

capacities, which was introduced in Hartmann (2012) and more elaborately described in Hartmann (2015). A more general RCPSP extension which also includes time-varying capacities was introduced by Klein (2000). This is a well-researched problem for which many powerful algorithms are available. Unfortunately, the set of all the possible overtime profiles can be extremely large.

On the other hand, the overtime cost can be minimized for the set of all possible fixed deadlines. This set of fixed deadlines can also be very large depending on the structure and size of the problem instance. The remaining problem for any given deadline resembles the following problems with resource-oriented objective functions, which are usually non-regular:

- Resource investment problem: Peak resource utilization must be minimized, cf., e.g., Drexl & Kimms (2001).
- Resource leveling problems: Negative and positive deviations from a given resource usage threshold have to be minimized, cf., e.g., Easa (1989).
- Resource overload problem: Only positive deviations are minimized, cf., e.g., Brucker & Knust (2012, p. 13) and Neumann et al. (2003, p. 242).

If we treat the remaining problem for a given fixed deadline as a resource overload problem, the issue arises that the practically required makespan of a resulting schedule may be shorter than the given deadline. (This difference may lead to an underestimation of the revenues.)

Both ideas to relate our problem to those previously presented approaches in the literature appear to be problematic, given the potentially large number of subproblems that are themselves hard to solve. We are therefore not aware of any procedure to solve the problem type presented above. For this reason, we now state it formally and present newly developed algorithmic solution approaches.

3. Formal description and analysis of the RCPSP with revenues and overtime cost

3.1. Mathematical model

We now formally define the resource constrained project scheduling problem with revenues and overtime cost as described in Section 2.1. This formulation is based on the widely used RCPSP linear programming formulation, see Klein (1999, p. 79f). We use so-called dummy activities 0 and $J + 1$ with a duration of 0 periods and no resource consumption to represent the distinct start and end of the project. In a preprocessing step, we compute for each activity j earliest finish times EFT_j and latest finish times LFT_j by standard forward and backward pass calculations, see, e.g., Demeulemeester & Herroelen (2006, p. 96ff.). In this process, earliest possible finishing times EFT_j can easily

Table 3: Notation of the RCPSP-ROC

Indices and (ordered) sets	
$j \in \mathcal{J}$	activities $\mathcal{J} = \{0, 1, \ldots, J, J+1\}$
$t, \tau \in \mathcal{T}$	periods $\mathcal{T} = \{0, 1, \ldots, T\}$
$r \in \mathcal{R}$	renewable resources $\mathcal{R} = \{1, \ldots, R\}$
$\mathcal{P}_j \subseteq \mathcal{J}$	set of predecessors of activity j

Parameters	
d_j	duration of activity j
EFT_j	earliest finishing time of activity j
LFT_j	latest finishing time of activity j
k_{jr}	required units of resource r while executing activity j
K_r	capacity of resource r
$\overline{z}_r$	overtime limit of resource r
κ_r	per unit cost for overtime of resource r
u_t	revenue for project termination in period t

Decision variables	
x_{jt}	$= \begin{cases} 1, & \text{if activity } j \text{ ends in period } t \\ 0, & \text{otherwise} \end{cases}$
z_{rt}	amount of overtime of resource r used in period t

be determined by constructing an *earliest start schedule*, thereby ignoring any capacity constraints. In a similar way, a latest finishing time LFT_{J+1} for the dummy ending activity $J+1$ can be determined as the sum of activity durations $LFT_{J+1} = \sum_{j=1}^{J} d_j$, assuming that all activities are executed successively, which is always feasible. From this latest finishing time LFT_{J+1}, the other latest finishing times can then be derived in a backward pass.

The central binary decision variable x_{jt} of the discrete time model equals 1 if activity j is finished in period t and 0 otherwise. The implied amount of overtime used in period t at resource r is tracked in the derived decision variable z_{rt}. Using the notation as given in Table 3, we now define the RCPSP-ROC as follows:

Model RCPSP-ROC

$$\max F = \sum_{t=EFT_{J+1}}^{LFT_{J+1}} u_t \cdot x_{J+1,t} - \sum_{r \in \mathcal{R}} \sum_{t \in \mathcal{T}} \kappa_r \cdot z_{rt} \tag{1}$$

subject to

$$\sum_{t=EFT_j}^{LFT_j} x_{jt} = 1, \qquad j \in \mathcal{J} \tag{2}$$

$$\sum_{t=EFT_i}^{LFT_i} x_{it} \cdot t \leq \sum_{t=EFT_j}^{LFT_j} x_{jt} \cdot t - d_j, \qquad j \in \mathcal{J}, i \in \mathcal{P}_j \tag{3}$$

$$\sum_{j=1}^{J} \sum_{\tau=t}^{t+d_j-1} k_{jr} \cdot x_{j\tau} \leq K_r + z_{rt}, \qquad r \in \mathcal{R}, t \in \mathcal{T} \tag{4}$$

$$z_{rt} \leq \overline{z}_r, \qquad r \in \mathcal{R}, t \in \mathcal{T} \tag{5}$$

$$x_{jt} \in \{0, 1\}, \qquad j \in \mathcal{J}, t \in \mathcal{T} \tag{6}$$

$$z_{rt} \geq 0, \qquad r \in \mathcal{R}, t \in \mathcal{T} \tag{7}$$

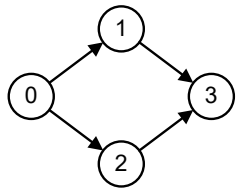
The objective function (1) maximizes the decision-dependent contribution to the profit. It is the revenue related to the finishing period of the dummy-activity $J + 1$ minus the total amount of overtime cost resulting from the schedule. Equations (2) enforce that each activity is finished exactly once between its earliest and latest finishing times. The precedence restrictions between activities are modeled via constraints (3). Capacity limits for the renewable resources are enforced and overtime usage is determined through constraints (4). An upper bound $\overline{z}_r$ for overtime usage is established via restrictions (5).

3.2. Complexity analysis, structural characteristics and algorithmic considerations

The RCPSP-ROC is a generalization of the RCPSP, which itself has been proven to be an $\mathcal{NP}$-hard problem by Blazewicz et al. (1983). Since there is a polynomial time reduction for RCPSP instances to RCPSP-ROC instances (that is RCPSP $\leq_p$ RCPSP-ROC), it follows that RCPSP-ROC is also an $\mathcal{NP}$-hard problem. The reduction can be achieved by setting the revenue function to any strictly monotonically decreasing function (e.g., $u_t = -t$) and by preventing any usage of overtime by setting the overtime limit to zero, i.e., $\overline{z}_r = 0, \forall\, r$. Given the $\mathcal{NP}$-hardness of the RCPSP-ROC, we do not expect to be able to develop an exact algorithm for the RCPSP-ROC that runs in polynomial time. We also observed that the computation time using the Gurobi MIP-Solver even for small RCPSP-ROC instances can be substantial. For this reason, we turned to heuristic methods to determine at least sub-optimal schedules in acceptable time.

In order to (hopefully) find good solutions of a scheduling problem in a systematic manner, the structural properties of good or even optimal solutions to this problem have to be identified. To this end, Schwindt (2005) introduced the term "characteristic points". Characteristic points of a specific scheduling problem form a set of feasible schedules which is guaranteed to include at least one optimal schedule for that problem, see Neumann et al. (2000), who already described the underlying idea. For scheduling problems with regular objective functions, e.g., makespan minimization, this is the set of so-called *active*

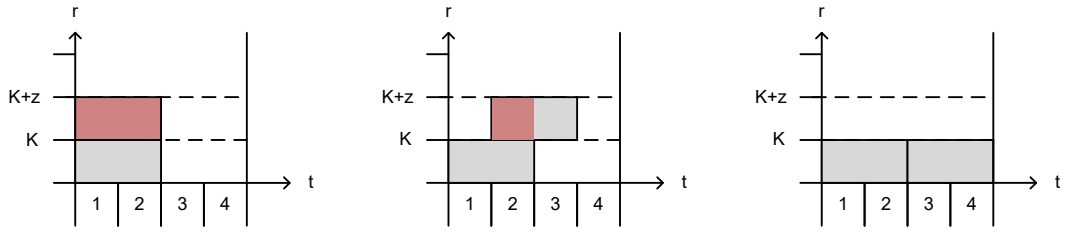
schedules $\mathcal{AS}$. A schedule is active if no activity can start earlier without delaying another one, i.e., no global left shift is possible in such a schedule. As a consequence, it is sufficient to consider "only" all active schedules in order to find the optimal solution. Many algorithmic approaches take this property into account. Unfortunately, for scheduling problems with objective functions that combine regular and non-regular components, such as the RCPSP-ROC with revenues and overtime cost, it is not sufficient to only consider the set of active schedules, see Ballestin & Blanco (2015, p. 418). It can be possible to improve a given schedule without changing its makespan by delaying an activity, thereby reducing overtime usage and cost. It is hence not sufficient or advisable to limit the search to the set of active schedules.



(a) Activity on node graph

t	1	2	3	4
u_t	2	2	2	1

(b) Revenue function



(c) $\pi = 1$ (d) $\pi = 1.5$ (e) $\pi = 1$

Figure 6: Example of optimal schedule being outside of $\mathcal{QSS}$

One might also consider to operate on the set of quasi-stable schedules $\mathcal{QSS}$ that have to be examined for problems in which resource usage deviations from a certain threshold are minimized, see Neumann et al. (2003, p. 210). However, one can find RCPSP-ROC instances for which none of the optimal schedules is quasi-stable and hence it is not even sufficient to limit the search to the set of quasi-stable schedules $\mathcal{QSS}$, see Figure 6. For the problem instance with two interchangeable symmetric non-dummy activities depicted in Figure 6a, with a revenue function shown in Figure 6b and per-unit overtime cost of $\kappa = \frac{1}{2}$ monetary units, the schedule in Figure 6c and the pair of schedules in Figure 6e are quasi-stable but only the non-quasi-stable schedule pair in Figure 6d is optimal.

As the set of quasistable schedules $\mathcal{QSS}$ is the largest set of characteristic points defined in literature, we are not able to classify our problem class into any known class of schedules. This implies that optimal schedules for the RCPSP-ROC have other and so far unknown properties than optimal schedules of established RCPSP variants. For this reason, it is not at all obvious how to systematically construct potentially optimal schedules.

4. Heuristic algorithmic approaches

4.1. General considerations: Genetic algorithms vs. LocalSolver

In order to develop algorithms for the RPCPSP-ROC, two different approaches appear to be very promising. On the one hand, population-based genetic algorithms (see Holland (1975)) turned out to be very powerful to solve RCPSPs, in particular with respect to the makespan minimization objective, see, e.g., the results reported in Kolisch & Hartmann (2006). If one follows this approach, the central question is how solutions are encoded and how schedules are derived from this encoding so that the operators of genetic algorithms can lead to new and still feasible schedules. (Note that a *direct* representation based on a possibly extremely large number of decision variables from the RCPSP-ROC model in Section 3 does *not* meet this fundamental requirement.) However, such a solution representation and corresponding schedule generation scheme can also be used within a (heuristic) local search algorithm. A commercial local search solver named *LocalSolver* has recently gained attention as it offers a flexible modeling interface to define in particular combinatorial optimization problems, for example, vehicle routing problems. It turned out to be relatively easy to use LocalSolver to solve our problem, given the solution representation and decoding mechanisms we developed for the genetic algorithms. Furthermore, LocalSolver easily bet the Gurobi MIP solver operating on the RCPSP-ROC formulation in Section 3. For those reasons, we used a relatively lightweight LocalSolver implementation based on the solution representations for the genetic algorithms as a surprisingly strong benchmark.

4.2. Alternative solution encodings and corresponding schedule generation schemes

4.2.1. The serial schedule generation scheme based on an activity list λ

In the context of the RCPSP, the serial schedule generation scheme (SSGS) is widely used to decode a solution representation based on an activity list λ into a schedule, see Kolisch & Hartmann (1999, p. 150ff). In an activity list λ, all jobs (including the dummy activities) are included once. If an activity i in the project has to precede another activity j, then this order has to be respected in the activity list as well. Hence the first and last entries of the activity list are always the (dummy) start and end activities. Using the SSGS, one iteratively schedules activities in the order implied by the activity list. Starting with the first activity on this list λ, one determines its earliest starting point that is feasible both with respect to capacity constraints and activity precedence relations.

Consider, for example, the project depicted in Figure 1 requiring a single resource $r = 1$ with a period capacity of $K_1 = 4$ capacity units (and no overtime capacity, that is $\overline{z}_r = 0, \forall\, r$). Then the activity list

$$\lambda_1 = (0, 1, 3, 2, 5, 4, 6, 7) \tag{8}$$

is decoded into the schedule in Figure 2. Note that the SSGS is not injective, meaning different activity lists can lead to the same schedule. For details on this established procedure see Kolisch & Hartmann (1999).

This serial schedule generation scheme operating on an activity list λ only generates active schedules $\mathcal{AS}$ which are a subset of the quasi-stable schedules, i.e., $\mathcal{AS} \subseteq \mathcal{QSS}$, when enumerating over all possible activity lists λ as input data. As mentioned before, it is not even sufficient to consider (only) the set of $\mathcal{QSS}$ schedules to find an optimal solution for the RCPSP-ROC. We are hence not aware of any established construction rule operating on an activity list λ and the SSGS to build promising schedules for the RCPSP-ROC, due to its specific objective function.

For this reason, we developed several *extended* solution representations and modified decoding mechanisms that can all be seen as generalizations of the established SSGS approach for the RCPSP. We describe these below in detail.

The basic reasoning is that when constructing a schedule, it is (with respect to revenues) essentially attractive to schedule activities as early as possible. This tends to be achieved by the SSGS. However, in the RCPSP-ROC there is the additional question of when or for which activities overtime should be used. We present below three different approaches in which this decision is directly determined by the solution representation.

In addition, we adapted the so called iterative forward backward improvement technique by Li & Willis (1992). It is an improvement step for a given schedule appended to all decoding schemes introduced in the following paragraphs. This technique tries to decrease the project duration by subsequently shifting and aligning all activities to the right and afterwards to the left. This is iteratively repeated until there is no improvement as opposed to the more widespread double justification variant with just two shifting passes. The approach often tends to decrease makespan maintaining the feasibility of the schedule. For a more detailed description of this procedure see Valls et al. (2005). We modified the procedure by allowing only such shifts that do not increase overtime consumption.

4.2.2. Solution encoding $(\lambda|\hat{z}_r)$

One possible solution encoding for the RCPSP-ROC is the representation $(\lambda|\hat{z}_r)$. Here $\hat{z}_r$ denotes a column vector specifying the maximum permissible overtime usage for resource r. When decoding a solution via the SSGS, we hence operate on a *modified* and time invariant period capacity $K_{rt}^{\text{mod}} = K_r + \hat{z}_r$ for each resource r.

For the project introduced in Figure 1 requiring a single resource $r = 1$ with a regular period capacity of $K_1 = 4$ capacity units, the representation

$$(\lambda|\hat{z}_r) = \ (0 \quad 1 \quad 3 \quad 2 \quad 5 \quad 4 \quad 6 \quad 7 \mid [0]) \tag{9}$$

(without overtime permission) is decoded into the schedule in Figure 2 whereas the representation

$$(\lambda|\hat{z}_r) = \ (0 \quad 1 \quad 3 \quad 2 \quad 5 \quad 4 \quad 6 \quad 7 \mid [2]) \tag{10}$$

leads to the schedule in Figure 3. In this case, the additional decision on a fixed upper bound $\hat{z}_1 = 2$ for overtime throughout the entire planning horizon is a component of the encoded solution. Please note that $\hat{z}_r$ only gives the maximum permissible overtime and that the amount of overtime actually used has to be derived from the schedule in order to compute the objective function value related to this schedule.

For a single resource r, the set of possible (integer) $\hat{z}_r$ values is rather small and contains $\overline{z}_r + 1$ elements $\{0, \ldots, \overline{z}_r\}$. However, even a full state space enumeration for this representation does not necessarily yield an optimal schedule, since this representation only explores a subset of the entire set of all possible overtime profiles. An obvious advantage of this representation is that it is quite lean.

4.2.3. Solution encoding $(\lambda|\hat{z}_{rt})$

The $(\lambda|\hat{z}_{rt})$ representation is quite similar to the $(\lambda|\hat{z}_r)$ representation. It generalizes the $(\lambda|\hat{z}_r)$ representation by deciding on a per-period basis on the allowed amount of overtime and therefore inducing a time-varying profile. Here $\hat{z}_{rt}$ denotes a matrix of permissible overtime capacities. When applying the SSGS, the *modified* and time-variant period capacity $K_{rt}^{\text{mod}} = K_r + \hat{z}_{rt}$ for each resource r is considered.

For the project introduced in Figure 1, the representation

$$(\lambda|\hat{z}_{rt}) = \ (0 \quad 1 \quad 3 \quad 2 \quad 5 \quad 4 \quad 6 \quad 7 \mid \begin{bmatrix} 0 & 0 & 0 & 2 & 2 & 2 & 0 & 0 & 0 & 0 \end{bmatrix}) \tag{11}$$

with permission of two units of overtime in periods 4, 5, and 6 is decoded into the schedule in Figure 2 (in which no overtime is actually used) whereas the representation

$$(\lambda|\hat{z}_{rt}) = \ (0 \quad 1 \quad 3 \quad 2 \quad 5 \quad 4 \quad 6 \quad 7 \mid \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}) \tag{12}$$

with permission of one unit of overtime in periods 3 and 4 leads to the schedule in Figure 4.

Since this representation can be used to explore the entire set of possible overtime profiles, it is in principle guaranteed to find an optimal solution when doing a full state space exploration. However, this is of course not practical since it is still an $\mathcal{NP}$-hard problem and the matrix of permissible overtime capacities $\hat{z}_{rt}$ can get very large very quickly, requiring a high-dimensional search process in combination with all possible activity lists.

4.2.4. Solution encoding $(\lambda|\beta)$

The main idea behind the $(\lambda|\beta)$ representation is that the need to use overtime could be tied to activities j as opposed to resources r as assumed in the $(\lambda|\hat{z}_r)$ representation or resource-period-combinations (r,t) in the $(\lambda|\hat{z}_{rt})$ representation. The $(\lambda|\beta)$ representation hence explicitly contains the information whether an activity is allowed to be scheduled with or only without additional overtime usage. When the activity list λ is being decoded

and an activity j is being treated given a partial schedule, the *original* capacity K_r is used with respect to resource r if $\beta_j = 0$, i.e., if activity j is *not* allowed to use overtime while being inserted into the partial schedule. However, if $\beta_j = 1$, i.e., if activity j is allowed to use overtime, the *modified* capacity $K_r^{\text{mod}} = K_r + \overline{z}_r$ is being used.

For the project depicted in Figure 1 requiring a single resource $r = 1$ with a regular period capacity of $K_1 = 4$ capacity units the representation

$$\begin{pmatrix} \lambda \\ \beta \end{pmatrix} = \begin{pmatrix} 0 & 1 & 3 & 2 & 5 & 4 & 6 & 7 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{13}$$

with permission of overtime for activities 2 and 6 is decoded into the schedule in Figure 3 whereas the representation

$$\begin{pmatrix} \lambda \\ \beta \end{pmatrix} = \begin{pmatrix} 0 & 1 & 3 & 2 & 5 & 4 & 6 & 7 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \tag{14}$$

with permission of overtime for activities 3 and 5 leads to the schedule in Figure 2. Note that even a full state space enumeration along this representation is not guaranteed to yield an optimal solution as it may be optimal to use overtime during only a *part* of the execution of an activity (see Figure 6d).

4.3. Genetic algorithms for the RCPSP-ROC

4.3.1. Basis solution approach of genetic algorithms

Genetic algorithms are nature-inspired meta-heuristics that have been successfully applied to many different combinatorial optimization problems. They were first presented by Holland (1975) and are now widely and successfully used in computer science and operations research. As shown in Kolisch & Hartmann (2006), they are the dominating heuristic method in the literature to solve the RCPSP as they can often find high-quality solutions for challenging problems very quickly.

Genetic algorithms for an optimization problem operate on a population of individuals or candidate solutions over a sequence of generations with reproduction and selection mimicking the "survival of the fittest". A candidate solution can in principle contain a complete set of numerical values as assignment for the decision variables of an underlying optimization model as the binary x_{jt} and the integer z_{rt} variables in the RCPSP-ROC model presented in Section 3. However, it is often more convenient and efficient to use a more compact encoding like the activity list λ introduced before. It is an established compact encoding for a schedule. Decoding an activity list into the starting times of a schedule is efficiently possible via the SSGS. The objective function value of the schedule then serves as a fitness value for the particular activity list of that individual.

The individuals of one generation are taken as parents, combined in the so-called crossover operation, and potentially mutated in order to hopefully create better child individuals. Out of the set of parents and children, a new generation of parents for

the next generation is selected. This process is repeated iteratively until a specified termination criterion is met.

To characterize our genetic algorithms, we hence have to specify the solution representation, the decoding scheme and the fitness computation as well as the structure of the initial population, the combination of solutions, and finally the mutation and selection operators. As representations we use the encodings introduced in Section 4.2 together with their corresponding schedule generation schemes. For all these considered representations, we specify the remaining elements of the genetic algorithm in the following subsections.

4.3.2. Generation of the initial population

All our solution representations contain activity lists as a substantial element. We construct each activity list in the initial population step by step, starting with the first position of that list. For each position, we determine the set of activities $j \in \mathcal{D}$ that have not yet been assigned to the activity list while all immediate predecessor activities $i \in P_j$ have already been assigned to that particular list. Each such activity $j \in \mathcal{D}$ can hence be selected for the currently considered position of the activity list without violating any precedence constraint between activities. One of these activities $j \in \mathcal{D}$ is chosen randomly following a distribution where the selection probability positively correlates with the priority value of that activity (biased sampling). The chosen activity is then appended to the activity list and this procedure repeats until all activities are included.

In our case, we are using the latest finishing times LFT_j of the activities as priority values, so that the priority of activities decreases with increasing latest finishing times. This is a useful priority rule, since delaying an activity j with a small LFT_j value is likely to postpone project completion. Based on these priorities, the weight $w_j = \max_{i \in \mathcal{D}} LFT_i - LFT_j$ is the relative regret of *not* selecting activity j, i.e., the difference between highest overall priority value for the assignable activities $i \in \mathcal{D}$ and priority value of activity j. To randomly select one of the schedulable activities $j \in \mathcal{D}$, we use non-uniform selection probabilities

$$Prob_j = \frac{(w_j + 1)}{\sum_{i \in \mathcal{D}} (w_i + 1)}$$

in a regret-based biased random sampling (RBBRS) as proposed in Tormos & Lova (2001).

In the $(\lambda|\hat{z}_r)$ representation, an additional initial limit on the permissible overtime $\hat{z}_r$ for each resource r has to be assigned for each individual. We draw it randomly from a uniform distribution over the integer values in the set $\{0, 1, 2, ..., \overline{z}_r - 1, \overline{z}_r\}$ for each resource r. In the case of the $(\lambda|\hat{z}_{rt})$ representation, we use this limit for resource r over all periods t so that initially we have $\hat{z}_{r,1} = \hat{z}_{r,2} = ... = \hat{z}_{r,T}$ for each resource r of an individual. In the $(\lambda|\beta)$ representation, the binary parameter β_j which indicates whether activity j may be scheduled using overtime is set to 0 or 1 with probabilities of 0.5 each.

4.3.3. Crossover

During each iteration of the genetic algorithm, we build pairs of individuals by randomly matching distinct individuals from the current parent set until each individual has been matched with one other individual from that set. Denote one individual from such a pair as the mother "M" and the other as the father "F".

Let λ^M be the mother's activity list and λ^F be the father's activity list. Following Hartmann (1998), we perform a one-point-crossover on those activity lists. We pick a random number q between 1 and J. A daughter is characterized by choosing the first q elements from the mother. The remaining elements, not yet chosen from the mother, are taken in the order of the father. The son is determined analogously by switching the roles of mother and father. With this approach, all precedence restrictions are always met, so that each activity list is feasible. For the $(\lambda|\beta)$ representation containing an additional overtime decision per job, the overtime decision for each job is linked to the overtime decision from the passing parent.

We describe this procedure using an example for the $(\lambda|\beta)$ representation of the sample project in Figure 1 (considering only the non-dummy activities) with the crossover point $q = 3$:

$$I_\lambda^M = \left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{array}\right) \quad I_\lambda^F = \left(\begin{array}{ccc|ccc} 1 & 3 & 5 & 2 & 4 & 6 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{array}\right)$$

$$I_\lambda^D = \left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 5 & 4 & 6 \\ 1 & 1 & 0 & 1 & 1 & 0 \end{array}\right) \quad I_\lambda^S = \left(\begin{array}{ccc|ccc} 1 & 3 & 5 & 2 & 4 & 6 \\ 1 & 0 & 1 & 1 & 0 & 0 \end{array}\right)$$

The first three positions on the activity list for the daughter "D" are taken in the sequence given by the mother. It is not possible to simply take the last three elements from the father as activity 2 would be implemented twice and activity 5 never. For this reason, we only take the sequence of the remaining activities from the father: 5, 4 and 6. The β_j values for the first three positions are inherited from the mother. For the other activities, the values are inherited from the father. The son "S" analogously inherits in the opposite direction.

In the $(\lambda|\hat{z}_r)$- and $(\lambda|\hat{z}_{rt})$-representations, a second one-point-crossover is performed. The crossover of the $\hat{z}_r$ and $\hat{z}_{rt}$ components is independent of the crossover of the activity lists. The reason is that there exists no direct relationship between the elements of an activity list and the entries of the matrix $\hat{z}_{rt}$ or vector $\hat{z}_r$. We show the procedure for the case of two resources r and six periods t and first assume that the crossover is performed *period-wise* between periods 4 and 5:

$$I_{\hat{z}_{rt}}^M = \left(\begin{array}{cccc|cc} 3 & 3 & 3 & 3 & 3 & 3 \\ 1 & 1 & 4 & 4 & 4 & 4 \end{array}\right) \quad I_{\hat{z}_{rt}}^F = \left(\begin{array}{cccc|cc} 2 & 2 & 2 & 2 & 6 & 6 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right)$$

$$I^D_{\hat{z}_{rt}} = \left(\begin{array}{cccc|cc} 3 & 3 & 3 & 3 & 6 & 6 \\ 1 & 1 & 4 & 4 & 1 & 1 \end{array}\right) \quad I^S_{\hat{z}_{rt}} = \left(\begin{array}{cccc|cc} 2 & 2 & 2 & 2 & 3 & 3 \\ 0 & 0 & 1 & 1 & 4 & 4 \end{array}\right)$$

The result shows, e.g., for the son individual and resource $r = 1$, that the permissible overtime is two units for the first four periods (as inherited from the father) and three units for the remaining two periods (as inherited from the mother). In a similar way, the crossover can be performed *resource-wise* between resources, in this case the only two resources 1 and 2:

$$I^M_{\hat{z}_{rt}} = \left(\begin{array}{cccccc} 3 & 3 & 3 & 3 & 3 & 3 \\ \hline 1 & 1 & 4 & 4 & 4 & 4 \end{array}\right) \quad I^F_{\hat{z}_{rt}} = \left(\begin{array}{cccccc} 2 & 2 & 2 & 2 & 6 & 6 \\ \hline 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right)$$

$$I^D_{\hat{z}_{rt}} = \left(\begin{array}{cccccc} 3 & 3 & 3 & 3 & 3 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right) \quad I^S_{\hat{z}_{rt}} = \left(\begin{array}{cccccc} 2 & 2 & 2 & 2 & 6 & 6 \\ \hline 1 & 1 & 4 & 4 & 4 & 4 \end{array}\right)$$

In the case of the $(\lambda|\hat{z}_{rt})$ representations, we either perform the crossover period-wise or resource-wise (with probabilities of 0.5 each). For the $(\lambda|\hat{z}_r)$ representations, it can only be performed resource-wise.

4.3.4. Mutation

Mutation is only applied with a certain small mutation probability P_{mutate}. For activity lists, we apply a so called adjacent pairwise interchange, see (Hartmann, 1999, p. 90) and Brucker & Knust (2012, p. 130). First, a list position is determined at random. Then the activity on the currently considered list position is exchanged with the activity on the next position unless this would result in an infeasible activity list. Therefore, feasibility-violating swaps are avoided.

In the case of the $(\lambda|\hat{z}_r)$- or the $(\lambda|\hat{z}_{rt})$-representation, we mutate the overtime capacities $\hat{z}_r$ and z_{rt} by randomly either increasing or decreasing all values by one capacity unit (unless this would violate the limits 0 and $\overline{z}_r$, respectively). For the $(\lambda|\beta)$ representation, we mutate by flipping a randomly selected bit, i.e., by setting $\beta_j \leftarrow 1 - \beta_j$.

4.3.5. Selection

In the selection step, individuals with low fitness values get discarded. For our implementation, we chose to discard the 50% worst individuals out of the set of all parents and all children of the current generation, i.e., we use an elite selection scheme.

4.4. Central elements of the local search implementation using LocalSolver

The commercial general-purpose local search solver *LocalSolver* offers as its front end a descriptive modeling language. It can be used to declare and define the elements of an optimization model such as variables, parameters, objective function, and constraints.

In principle, it is possible to use a direct solution representation based on the binary x_{jt} variables as defined in the RCPSP-ROC. However, this representation is neither suitable

for a local search algorithm nor for a genetic algorithm. Swapping two jobs in a schedule, for example, is conceptually rather simple and easily implemented in an activity list encoding, whereas in a direct binary encoding of a schedule potentially many different variables must be modified and it can be difficult to achieve feasibility again.

Fortunately, the descriptive LocalSolver language contains not only the modeling elements commonly used in MIP models like the RCPSP-ROC in Section 3, but also other language elements like if-then-clauses, maximum or minimum operators and so-called *list variables* that can be used to describe problems in a very compact way. In particular, this list variable can be used to directly model an activity list and perform a local search over this activity list using the black-box LocalSolver search engine, i.e., its back end.

Using a list variable to directly model the activity list, however, turned out to be very simple and effective. A list variable in LocalSolver holds a permutation of numbers in a certain range $[0, n-1] \cap \mathbb{N}_0$ of n elements. The upper limit $n-1$ and therefore the cardinality n of this range can be externally specified. For our problem setting, we set $n = J + 2$. LocalSolver explores possible solutions by permuting this list.

When using an indirect encoding via an activity list in a modeling language, a corresponding decoding procedure must be implemented. The current version of the LocalSolver language is well suited for descriptive programming but not yet suitable for highly efficient algorithmic procedural programming. However, the LocalSolver SDK offers an API for implementing parts of the model as function callbacks written in another general-purpose programming language. With these so-called *native functions*, a custom algorithm can be integrated in a LocalSolver model and the respective search process. This allows us to reuse and plug in the schedule-generating and decoding procedures described in Section 4.2 and already implemented in the genetic algorithm.

The additional effort to use LocalSolver within our C++ program is hence very limited. The main component for the case of the $(\lambda|\beta)$ representation is shown in Listing 1. This short code fragment is sufficient to declare the model's data object (code lines 1 and 2), embed the native decoding function (code lines 5 and 6), and establish the activity list augmented by the binary β_j vector (code lines 9 to 24) using a list variable. This way, additional data representing the decisions on overtime is integrated into LocalSolver.

Note that the list variable, i.e., the activity list, is not guaranteed to be topologically ordered after a position swap performed by LocalSolver. There are two ways to resolve this issue. First, one could add an additional constraint to the model, which enforces that list elements (i.e., activities) can only occur at a certain list position if all predecessors are at earlier positions on that list. In this case, the local search via the LocalSolver engine is forced to generate only feasible activity lists. Alternatively, one could omit this constraint and instead modify the decoding procedure to be usable with all possible permutations. To this end, it is sufficient to delay scheduling an activity on the list until all its immediate predecessors have be scheduled when decoding the activity list. Numerical experiments

Listing 1: Model definition in the LocalSolver language

```
LocalSolver ls;
LSModel model = ls.getModel();

// Plug in native C++ function into model as LocalSolver expression
auto nativeFunction = model.createNativeFunction(decoder);
LSExpression objective = model.call(nativeFunction);

// Declare list permutation variable with fixed length #jobs
LSExpression activityList = model.listVar(project.numJobs);
model.constraint(model.count(activityList) == project.numJobs);

// Generate handles for each list element, insert as objective function operands
vector<LSExpression> listElems(project.numJobs);
for (int i = 0; i < project.numJobs; i++) {
  listElems[i] = model.at(activityList, i);
  objective.addOperand(listElems[i]);
}

// Declare boolean decision vector for overtime and insert as objective function operands
vector<LSExpression> betaVar(project.numJobs);
for (int i = 0; i < project.numJobs; i++) {
  betaVar[i] = model.boolVar();
  objective.addOperand(betaVar[i]);
}

model.addObjective(objective, OD_Maximize);
model.close();
```

show that the second option is more efficient on average.

In summary, LocalSolver only needs a trivial model consisting of the list variable, setting its length to the number of jobs and possible additional decision variables for overtime. LocalSolver decides on the assigned values of the decision variables and then passes these variables to the native function facilities of the LocalSolver API. The called decoding procedure maps the list and overtime decisions to a schedule and an objective function value as described in Section 4.2. The local search stops when a pre-determined clock time has elapsed. Contrasting it with the genetic algorithm, we observe that we now operate on a single solution as opposed to a population of solution and that all the modification and selection is done by a black-box engine as opposed to the crossover, mutation and selection operators required in the genetic algorithms.

5. Numerical analysis of the different solution methods

5.1. Test design

We performed a set of numerical experiments in order to evaluate the relative quality of the different solution approaches presented in Section 4. To this end, the widely used PSPLIB problem library of heterogeneous and challenging RCPSP-instances presented in Kolisch & Sprecher (1996) was used and modified to match the specific characteristics of our problem. In particular, we defined additional parameters not already included in the classical RCPSP. These are resource-specific overtime cost of $\kappa_r = \frac{1}{2}$ monetary units per capacity unit and period, upper bounds for overtime $\overline{z}_r = \frac{1}{2}K_r$, and the makespan-dependent revenue function u_t. The revenue function has to be constructed carefully to

avoid that the optimal solutions either always have zero overtime or always use overtime whenever possible. In these two trivial cases, a standard RCPSP procedure to minimize the makespan would be sufficient after adjusting the capacities to $K'_r = K_r + \overline{z}_r$ or $K'_r = K_r$, respectively.

Thus, interesting problem instances have a certain structure with their optimal makespan being between those two extreme points $\underline{T}$ and $\overline{T}$ as shown in Figure 5. Ideally, $\underline{T}$ should be the shortest possible makespan that can be achieved within the overtime limits z_r. In a similar way, $\overline{T}$ should be the shortest possible makespan that is possible without any overtime. However, in order to determine these two values, two $\mathcal{NP}$-hard problems would have to be solved, which is entirely impractical. In a fast and simple (but admittedly crude) approximation, we took advantage of the fact that in the PSPLIB, the activities for each project are always topologically ordered. For this reason, the SSGS can always be used to decode the canonical activity list $\lambda = (0, 1, 2, 3, 4, ..., J-2, J-1, J, J+1)$ into a feasible schedule without overtime. The makespan of this schedule is an upper bound of the duration of the shortest feasible schedule without overtime. Furthermore the makespan of the earliest start schedule delivers an efficiently computable lower bound for the shortest schedule with arbitrary legitimate amount of overtime. We call these the $\overline{T}$- and the $\underline{T}$-schedules, respectively. The makespans of those two schedules were then used as the $\underline{T}$ and $\overline{T}$ limits of the interesting makespan interval for the respective revenue function. In order to roughly match the structure of the profit curve in Figure 5, the revenue function has to be determined in a suitable way. We decided to use a partially parabolic function defined as follows:

$$u_t = \begin{cases} \overline{C}, & \text{if } t < \underline{T} \\ \overline{C} - \frac{\overline{C}}{(\overline{T}-\underline{T})^2}(t-\underline{T})^2, & \text{if } \underline{T} \leq t \leq \overline{T} \\ 0, & \text{if } \overline{T} < t \end{cases} \tag{15}$$

Here $\overline{C}$ denotes the *actual* cost of overtime associated with the earliest start schedule $\underline{T}$ as defined above, thus representing an upper bound for overtime cost. Note that the revenue function is concave and monotonically decreasing in the relevant makespan interval from $\underline{T}$ to $\overline{T}$. We hence know that at least one schedule exists with a profit of at least 0 monetary units for each project.

We extended all 480 PSPLIB instances from the set j30 with 30 actual activities and two dummy activities as described. We omitted instances in which the schedule computed using the SSGS and the canonical activity list and arbitrary amounts of overtime did *not* actually use overtime, which can be the case if the project structure is essentially serial as few activities can be executed in parallel. This is a rough heuristic to decide whether overtime potentially has relevance for this instance. We furthermore excluded all instances for which the Gurobi MIP solver could not find a proven optimal solutions within 1800 seconds of CPU time on a single processor with 32 GB of RAM. This resulted in 270

interesting problem instances with known optimal solutions. With this preparation, we were able to compute meaningful relative profit deviations when comparing our heuristic results to optimal solutions.

In a similar way, we examined j120 instances from the PSPLIB, where the total number of 600 instances got reduced to 585 projects relevant for our problem setting. We were unable to determine proven optimal solutions using Gurobi for this problem class of much larger project networks with 120 activities each. We hence evaluated the different heuristics against each other, using the best known solution per instance as a benchmark.

The activity list decoding schemes and the genetic algorithm were implemented in C++ to achieve computational performance and interoperability with the LocalSolver API. Based on numerical tests, we chose a mutation probability $P_{mutate} = 5\%$ and population size (size of one generation) $N^I = 80$. The results were obtained on a single processor with 16 GB of RAM workstation using one thread.

5.2. Results

For the 270 comparatively small projects with 30 non-dummy activities in conjunction with a time limit of 30 seconds we generated the results shown in Table 4.

This table shows the chronological progression of the relative gap of each solution method averaged over all instances. For each instance and method the relative gap is a result of computing the deviation between the known optimal reference solution computed by Gurobi and the solution that method discovered up to that point in time. More precisely, this deviation is defined as $\frac{p^*-p}{p^*}$, where p is the profit considered and p^* is the optimal profit. Please note that Gurobi is both used in a first run to compute optimal reference values and in a second run with time limit of 30 seconds to benchmark it as exact method against the heuristic methods.

The cells showing average gaps are colored using a palette between red for high gaps and green for low gaps. This helps following and discerning the comparative progression of gaps visually. All methods start from a initial seed solution with no profit and therefore a relative gap of 100% to the optimal solution.

The first three rows of Table 4 contain aggregated information for each method. This information is computed after reaching the time limit. It shows the highest relative gap of any individual problem instance, the percentage of instances which were solved to optimality and the average gap.

Overall the gaps show, that with a time limit of just 30 seconds all heuristic solution methods are able to generate good solutions and are able to move towards them rather quickly with gaps of only 0.02% to about 0.28%.

The table also includes the behavior of the MIP solver Gurobi. One can see that on average all heuristics outperform the exact reference method Gurobi during the considered timespan. Gurobi still has a gap of 1.46% after computation is terminated in its time limited run. Even though Gurobi is outperformed by the heuristics, this still shows that

Max Gap	100.00%	3.07%	20.00%	2.20%	1.00%	20.00%	2.98%
%Optimal	38.80%	39.49%	36.92%	41.03%	43.93%	42.56%	43.59%
ØGap	1.46%	0.10%	0.28%	0.08%	0.02%	0.12%	0.03%
Time	Gurobi	Genetic Algorithm			LocalSolver		
		$(\lambda\|\beta)$	$(\lambda\|\hat{z}_r)$	$(\lambda\|\hat{z}_{rt})$	$(\lambda\|\beta)$	$(\lambda\|\hat{z}_r)$	$(\lambda\|\hat{z}_{rt})$
0	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
0.01	100.00%	67.60%	64.66%	69.72%	82.50%	88.46%	82.51%
0.02	100.00%	4.41%	4.41%	3.59%	36.14%	50.68%	45.57%
0.03	97.41%	1.86%	1.84%	1.30%	5.30%	5.41%	20.82%
0.04	95.02%	1.26%	1.22%	0.95%	3.29%	2.20%	16.23%
0.05	90.44%	1.04%	1.06%	0.74%	2.04%	1.72%	13.75%
0.06	79.52%	0.82%	0.90%	0.58%	1.45%	1.46%	12.09%
0.07	70.58%	0.69%	0.77%	0.49%	1.25%	1.23%	10.59%
0.08	64.21%	0.63%	0.65%	0.42%	1.06%	1.15%	9.40%
0.09	57.38%	0.57%	0.53%	0.36%	1.00%	1.04%	8.53%
0.1	50.64%	0.53%	0.50%	0.32%	0.91%	0.93%	6.92%
0.11	46.77%	0.52%	0.47%	0.29%	0.86%	0.89%	6.01%
0.12	43.71%	0.47%	0.43%	0.28%	0.79%	0.83%	5.38%
0.32	21.20%	0.19%	0.35%	0.14%	0.46%	0.47%	1.53%
0.54	15.59%	0.16%	0.33%	0.12%	0.30%	0.38%	0.84%
1	9.62%	0.13%	0.32%	0.11%	0.19%	0.31%	0.53%
2	6.58%	0.13%	0.31%	0.09%	0.14%	0.25%	0.28%
3	5.26%	0.12%	0.31%	0.09%	0.10%	0.23%	0.21%
4	4.55%	0.12%	0.31%	0.09%	0.09%	0.20%	0.18%
5	3.96%	0.11%	0.31%	0.09%	0.09%	0.17%	0.12%
28	1.51%	0.10%	0.28%	0.08%	0.02%	0.12%	0.03%
29	1.51%	0.10%	0.28%	0.08%	0.02%	0.12%	0.03%
30	1.46%	0.10%	0.28%	0.08%	0.02%	0.12%	0.03%
Time	Gurobi	Genetic Algorithm			LocalSolver		
		$(\lambda\|\beta)$	$(\lambda\|\hat{z}_r)$	$(\lambda\|\hat{z}_{rt})$	$(\lambda\|\beta)$	$(\lambda\|\hat{z}_r)$	$(\lambda\|\hat{z}_{rt})$

Table 4: Numerical results for small projects with 30 non-dummy activities

the direct binary encoding x_{jt} yields acceptable results for such small instances when used as encoding for a MIP solver. This is in accord with the results from the experimental comparison of different RCPSP formulations in Artigues et al. (2015).

When comparing the heuristic solution methods, it becomes apparent that the genetic algorithm yields very good results in short time and with a small amount of schedules

respectively. This behavior is due to the fact that the problem-specific configuration of the genetic algorithm enables very good results by selecting promising schedules and combining them in a constructive way. However, after a few seconds this heuristic gets stuck in a local optimum.

In contrast to this, LocalSolver is not able to find such good solutions in the beginning due to the initially quite arbitrary creation of new schedules. However, the results by LocalSolver improve steadily so that they dominate the genetic algorithm for all representations after 9 seconds. The number of schedules visited at a certain point in time will be roughly the same in case of both LocalSolver and the genetic algorithms. The decoding procedures used in conjunction with LocalSolver might be even slightly slower due to being robust against non-topologically sorted lists, which are avoided as inputs of the fitness functions of the genetic algorithms. Hence, the superior performance of LocalSolver after a few seconds must be caused by more flexible neighborhood movement rules of this highly optimized generalized local search solver.

The results for the 585 big projects with 120 non-dummy activities with a time limit of 60 seconds are shown in Table 5. Since even for the simpler RCPSP there exist many j120 instances, which are not optimally solved yet, we did not attempt to compute optimal reference values for this set of big instances. Instead the relative deviation is computed referencing the best known solution of all methods. These best known solutions represent lower bounds for the optimal profit. In order to tighten these bounds we additionally computed profits using the most promising $(\lambda|\hat{z}_{rt})$ representation in a genetic algorithm on a computing cluster with a time limit of 30 minutes per instance.

For big projects with 120 activities the exact method is not able to produce any reasonably useful results within a one minute time limit. The heuristic methods still yield fairly good solutions with small gaps towards the best known solutions. The dominance of LocalSolver models using indirect solution encodings over the problem-specific genetic algorithm counterparts is now broken and flipped. The specific genetic algorithms clearly beat all other procedures considered when solving larger problem instances. So the low-effort solution method of using a standard solver is not advisable for big instances. Additionally it seems the $(\lambda|\hat{z}_{rt})$ representation is not very well suited for solving big problem instances in conjunction with LocalSolver. This may be due to LocalSolver not being able to efficiently traverse different $\hat{z}_{rt}$ assignments, of which there are many.

In summary, problem-specific procedures on average outperform black-box generic methods for big problem instances. This relationship is reversed when considering small instances, though. A generalized standard software beating a custom heuristic may not be intuitive at first sight. Although algorithmically the problem-specific approach is very likely to be superior, the generalized local search implementation is a commercial software product with years of development and effort by a huge number of programmers, whereas the genetic algorithm implementation was implemented in shorter time from scratch.

Max Gap	100.00%	3.80%	4.19%	3.74%	40.20%	8.87%	51.98%
%Best	27.18%	41.88%	35.73%	50.26%	27.18%	30.26%	25.30%
ØGap	60.57%	0.61%	0.59%	0.34%	2.87%	1.54%	8.68%
Time	Gurobi	Genetic Algorithm			LocalSolver		
		$(\lambda\|\beta)$	$(\lambda\|\hat{z}_r)$	$(\lambda\|\hat{z}_{rt})$	$(\lambda\|\beta)$	$(\lambda\|\hat{z}_r)$	$(\lambda\|\hat{z}_{rt})$
0	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
0.01	100.00%	68.46%	62.91%	62.89%	97.25%	94.55%	96.38%
0.02	100.00%	43.81%	43.39%	42.91%	72.40%	70.09%	69.69%
0.03	100.00%	35.77%	34.52%	34.58%	52.25%	43.98%	55.13%
0.04	100.00%	30.94%	30.26%	30.63%	47.59%	37.98%	51.42%
0.34	98.45%	5.25%	4.53%	4.57%	28.69%	7.76%	38.52%
0.52	93.66%	4.31%	3.75%	3.80%	24.58%	6.09%	36.88%
0.78	89.89%	3.69%	3.11%	3.16%	20.92%	4.95%	35.29%
1	87.49%	3.16%	2.62%	2.66%	18.39%	4.26%	34.48%
2	81.21%	2.40%	1.87%	1.86%	12.35%	3.19%	30.99%
3	79.09%	2.05%	1.52%	1.53%	9.94%	2.82%	28.20%
4	76.93%	1.83%	1.31%	1.31%	8.70%	2.58%	25.81%
5	75.42%	1.68%	1.16%	1.16%	7.52%	2.47%	23.83%
6	74.77%	1.56%	1.05%	1.04%	6.86%	2.38%	22.36%
7	73.85%	1.48%	0.97%	0.95%	6.40%	2.30%	20.68%
8	73.30%	1.39%	0.92%	0.88%	6.02%	2.25%	19.56%
9	72.81%	1.33%	0.87%	0.82%	5.79%	2.17%	18.35%
10	72.24%	1.28%	0.83%	0.78%	5.63%	2.14%	17.53%
60	60.57%	0.61%	0.59%	0.34%	2.87%	1.54%	8.68%
Time	Gurobi	Genetic Algorithm			LocalSolver		
		$(\lambda\|\beta)$	$(\lambda\|\hat{z}_r)$	$(\lambda\|\hat{z}_{rt})$	$(\lambda\|\beta)$	$(\lambda\|\hat{z}_r)$	$(\lambda\|\hat{z}_{rt})$

Table 5: Numerical results for big projects with 120 non-dummy activities

This might explain why a black-box heuristic solver is able to outperform a problem-specific genetic algorithm for small problem instances. For big instances, the algorithmic advantages from problem knowledge in the genetic algorithms seem to dominate any implementation issues in comparison to a commercially developed and optimized software.

6. Conclusion and outlook

In this paper we presented an extension of the RCPSP with overtime cost and a revenue function monotonically decreasing with project duration. We formalized the scheduling problem as a mixed-integer linear program and designed encodings as preparation step for the development of efficient solution procedures. We then developed a genetic algorithm for the problem, computed and interpreted results for a problem library based on a widely used RCPSP test set. We further investigated the use of a general local search heuristic, thus offering numerical results for both problem specific as well as generic black-box heuristic solution methods. The results are very promising. For huge practical real world projects with many activities, heuristic problem-specific solution methods beat generic black-box inexact solvers. For small size projects using a heuristic black-box method worked best.

For future research, it is promising to use modified operators of the genetic algorithm to achieve better results, for example the peak crossover operator proposed by Valls et al. (2008). This operator considers the fitness of the individuals in the crossover.

However, it is expected that the general solution behavior remains the same even with such improvements. Therefore, it would be even more interesting to use entirely different solution procedures or representations. A suitable and ideally more minimal solution encoding may speed up the solution process by removing more redundant points in the solution space. One idea is to evaluate a representation based on quasistable schedules known for resource-leveling problems with a heuristically defined makespan. Again, the goal is to explore the smallest possible set guaranteed to contain an optimal solution. However, to this end and also in order to develop exact algorithm, it would be extremely helpful to identify and formalize properties of optimal solutions.

Acknowledgments

The authors thank the German Research Foundation (DFG) for financial support of this research project in the CRC 871 “Regeneration of complex durable goods”.

References

Artigues, C., Koné, O., Lopez, P., & Mongeau, M. (2015). Mixed-integer linear programming formulations. In C. Schwindt, & J. Zimmermann (Eds.), *Handbook on Project Management and Scheduling Vol.1* International Handbooks on Information Systems (pp. 17–41). Cham Heidelberg New York Dordrecht London: Springer.

Ballestin, F., & Blanco, R. (2015). Theoretical and practical fundamentals. In C. Schwindt, & J. Zimmermann (Eds.), *Handbook on Project Management and Scheduling Vol.1* International Handbooks on Information Systems (pp. 411–427). Cham Heidelberg New York Dordrecht London: Springer.

Blazewicz, J., Lenstra, J. K., & Kan, A. R. (1983). Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, *5*, 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*, 3–41.

Brucker, P., & Knust, S. (2012). *Complex Scheduling*. GOR-Publications. Berlin Heidelberg: Springer.

Demeulemeester, E. L., & Herroelen, W. S. (2006). *Project scheduling: a research handbook* Volume 49 of *International Series in Operations Research and Management Science*. New York: Springer Science & Business Media.

Drexl, A., & Kimms, A. (2001). Optimization guided lower and upper bounds for the resource investment problem. *The Journal of the Operational Research Society*, *52*, 340–351.

Easa, S. M. (1989). Resource leveling in construction by optimization. *Journal of Construction Engineering and Management*, *115*, 302–316.

Hartmann, S. (1998). A competitive genetic algorithm for resource-constrained project scheduling. *Naval Research Logistics (NRL)*, *45*, 733–750.

Hartmann, S. (1999). *Project Scheduling under Limited Resources: Models, Methods, and Applications* Volume 478 of *Lecture Notes in Economics and Mathematical Systems*. Berlin Heidelberg: Springer.

Hartmann, S. (2012). Project scheduling with resource capacities and requests varying with time: a case study. *Flexible Services and Manufacturing Journal*, *25*, 74–93.

Hartmann, S. (2015). Time-varying resource requirements and capacities. In C. Schwindt, & J. Zimmermann (Eds.), *Handbook on Project Management and Scheduling Vol.1* International Handbooks on Information Systems (pp. 163–176). Cham Heidelberg New York Dordrecht London: Springer.

Hartmann, S., & Briskorn, D. (2010). A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, *207*, 1–14.

Herroelen, W., Demeulemeester, E., & De Reyck, B. (1997). *Resource-constrained project scheduling A survey of recent developments*. Berlin Heidelberg: Springer.

Holland, J. H. (1975). *Adaption in Natural and Artificial Systems: An Introductory Analysis with Application to Biology, Control, and Artificial Intelligence*. Detroit: The University of Michigan Press.

Kellenbrink, C., & Helber, S. (2015). Scheduling resource-constrained projects with a flexible project structure. *European Journal of Operational Research*, *246*, 379–391.

Kellenbrink, C., & Helber, S. (2016). Quality-and profit-oriented scheduling of resource-constrained projects with flexible project structure via a genetic algorithm. *European Journal of Industrial Engineering*, *10*, 574–595.

Klein, R. (1999). *Scheduling of resource-constrained projects* Volume 10 of *Operations Research / Computer Science Interfaces*. New York: Springer Science & Business Media.

Klein, R. (2000). Project scheduling with time-varying resource constraints. *International Journal of Production Research*, *38*, 3937–3952.

Kolisch, R., & Hartmann, S. (1999). *Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*. Boston, MA: Springer US.

Kolisch, R., & Hartmann, S. (2006). Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, *174*, 23–37.

Kolisch, R., & Padman, R. (2001). An integrated survey of deterministic project scheduling. *Omega*, *29*, 249–272.

Kolisch, R., & Sprecher, A. (1996). PSPLIB - A project scheduling problem library. *European Journal of Operational Research*, *96*, 205–216.

Li, K., & Willis, R. (1992). An iterative scheduling technique for resource-constrained project scheduling. *European Journal of Operational Research*, *56*, 370–379.

Neumann, K., Nübel, H., & Schwindt, C. (2000). Active and stable project scheduling. *Mathe-*

matical Methods of Operations Research, *52*, 441–465.

Neumann, K., Schwindt, C., & Zimmermann, J. (2003). *Project Scheduling with Time Windows and Scarce Resources: Temporal and Resource-Constrained Project Scheduling with Regular and Nonregular Objective Functions*. Berlin Heidelberg: Springer.

Özdamar, L., & Ulusoy, G. (1995). A survey on the resource-constrained project scheduling problem. *IIE transactions*, *27*, 574–586.

Pritsker, A. A. B., Waiters, L. J., & Wolfe, P. M. (1969). Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, *16*, 93–108.

Rieck, J., & Zimmermann, J. (2015). Exact methods for resource leveling problems. In C. Schwindt, & J. Zimmermann (Eds.), *Handbook on Project Management and Scheduling Vol.1* International Handbooks on Information Systems (pp. 361–387). Cham Heidelberg New York Dordrecht London: Springer.

Schwindt, C. (2005). *Resource allocation in project management*. Berlin Heidelberg: Springer.

Tormos, P., & Lova, A. (2001). A competitive heuristic solution technique for resource-constrained project scheduling. *Annals of Operations Research*, *102*, 65–81.

Valls, V., Ballestin, F., & Quintanilla, S. (2005). Justification and RCPSP: A technique that pays. *European Journal of Operational Research*, *165*, 375–386.

Valls, V., Ballestin, F., & Quintanilla, S. (2008). A hybrid genetic algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research*, *185*, 495–508.