J Sched (2008) 11: 137–148
DOI 10.1007/s10951-008-0059-7

Time-constrained project scheduling

T.A. Guldemond · J.L. Hurink · J.J. Paulus · J.M.J. Schutten

Published online: 14 February 2008

Abstract We propose a new approach for scheduling with strict deadlines and apply this approach to the Time-Constrained Project Scheduling Problem (TCPSP). To be able to meet these deadlines, it is possible to work in overtime or hire additional capacity in regular time or overtime. For this problem, we develop a two stage heuristic. The key of the approach lies in the first stage in which we construct *partial* schedules. In these partial schedules, jobs may be scheduled for a shorter duration than required. The second stage uses an ILP formulation of the problem to turn a partial schedule into a feasible schedule, and to perform a neighborhood search. The developed heuristic is quite flexible and, therefore, suitable for practice. We present experimental results on modified RCPSP benchmark instances. The two stage heuristic solves many instances to optimality, and if we substantially decrease the deadline, the rise in cost is only small.

Keywords Project scheduling · Strict deadlines

1 Introduction

In this paper we present a new scheduling methodology for scheduling problems with strict deadlines. The new approach is applied to a project scheduling problem with strict deadlines on the jobs, which we call the Time-Constrained Project Scheduling Problem (TCPSP). In many project scheduling problems from practice, jobs are subject to strict deadlines. In order to meet these deadlines, different ways to speed up the project are given, e.g., by working in overtime or hiring additional resource capacity. These options are costly but often not avoidable. The question arises how much, when, and what kind of extra capacity should be used to meet the deadlines against minimum cost.

The TCPSP is a variant on the well studied RCPSP (Resource-Constrained Project Scheduling Problem). However, there are fundamental differences between the time-constrained and the resource-constrained variant. In the TCPSP, the deadlines are strict and resource capacity profiles can be changed, whereas in the RCPSP, the given resource availability cannot be exceeded and the objective is to minimize the makespan. Moreover, in the TCPSP a non-regular objective function is considered. Therefore, the existing solution techniques of the RCPSP are not suitable for the TCPSP. For an overview of the literature on the RCPSP see, e.g., Herroelen et al. (1998), Kolisch and Hartmann (1999), and Kolisch and Padman (2001).

Although in practice deadlines often occur in projects, the Time-Constrained Project Scheduling has been considered only rarely in the literature. Deckro and Herbert (1989) give an ILP formulation for the TCPSP and discuss the concept of *project crashing*. In project crashing, the processing time of a job can be reduced to meet the project's deadline, at the cost of an increased resource usage, see also Li and Willis (1993) and Kis (2005). Kolisch (1995) discusses a heuristic procedure for the TCPSP with limited hiring. Another related problem is the *resource investment problem* (RIP), see Möhring (1984) and Demeulemeester (1995). For the RIP, it is also the goal to invest in resources as little as possible in order to meet a given project deadline. However,

T.A. Guldemond
ORTEC bv, PO Box 490, 2800 AL, Gouda, The Netherlands

J.L. Hurink (✉) · J.J. Paulus · J.M.J. Schutten
University of Twente, PO Box 217, 7500 AE, Enschede, The Netherlands
e-mail: j.l.hurink@utwente.nl

in the RIP only the investment in (new) resources is considered. If for a resource an investment in several units of this resource is done, these units are available for the complete planning horizon. The TCPSP is a scheduling problem where the resource availability in a time unit is fixed (e.g., amount of employees in a time unit during a working day) and it may be decided to extend in some of the time units these resource capacities by hiring extra resource units. Similarly, the *time driven rough-cut capacity planning problem*, see Gademann and Schutten (2005), is about matching demand for resources and availability of resources. For each time bucket, it can be decided how many resources are assigned to a job.

For the TCPSP, as presented here, the basic capacities of the resources are fixed. The resource investment decision took place in an earlier stage. What remains is the short term scheduling of the jobs so that costs of working in overtime and hiring are minimized. The TCPSP in this form is motivated by a cooperation with a Dutch company developing commercial planning software. It encounterd this problem with several of its clients and feels a need to extend its planning tool by a component which allows to tackle TCPSP like problems. To the best of our knowledge, in this paper the concept of overtime and multiple forms of irregular capacity are included in the modeling for the first time.

The outline of the paper is as follows. In Sect. 2, we introduce the presented solution approach that starts by planning jobs only fractionally. Section 3 gives a detailed problem description and presents an ILP to model the TCPSP. To solve the TCPSP, we develop a two stage heuristic in Sect. 4. The first stage of the heuristic constructs a partial schedule. The key of the approach lies in this first stage where jobs may be scheduled for a shorter duration than required. The second stage turns the partial schedule into a feasible schedule. Section 5 concerns the computational results. The test instances that we use are RCPSP benchmark instances from the PSPlib, see Kolisch and Sprecher (1997b), that are modified to TCPSP instances. In Sect. 6, we indicate how to extend the solution approach, e.g., for the multi-mode case. Sect. 7 gives a number of conclusions.

2 Scheduling with strict deadlines

For a good solution of the TCPSP, it is preferable to use regular working hours and avoid work in overtime and hiring, since these add to the costs. A typical greedy planning heuristic would start by scheduling jobs using only regular time and available resources as long as possible, and thereby avoiding costs. Only if a job would miss its deadline by using only regular working hours, working in overtime and hiring capacity comes into the picture. In our experience, using such a greedy strategy results in bad solutions. The reason for this is that by avoiding costs in the beginning, bottlenecks get shifted toward the end of the horizon and result in a pile up of cost toward the deadlines. The best solution might be to hire a small amount of additional capacity at the beginning to avoid costly situations when fitting in jobs that get scheduled close to their deadline.

Therefore, we propose a new scheduling methodology. Instead of scheduling jobs for their complete duration, we start by scheduling the jobs only for a fraction of their duration, and then we gradually increase the fraction for which the jobs are scheduled. This way, all jobs are partially admitted in the schedule before we address the bottlenecks. During this procedure it is guaranteed that we can turn the partial schedule into a feasible schedule. The goal is to create a schedule, in which the usage of irregular capacity is low, i.e., we try to prevent a pile up of costs toward the deadlines.

In the remainder of this paper we apply the idea of fractional planning and gradually increasing the fraction for which jobs are planned to the TCPSP. The computational test show that this idea works very well for the TCPSP. We believe that this new concept can be successfully applied to other scheduling problems with strict deadlines as well.

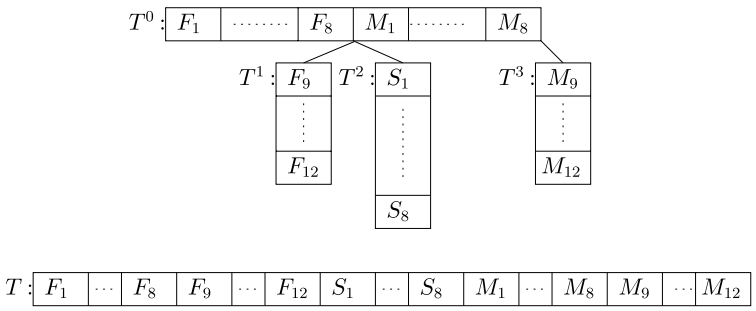
3 Problem description and ILP formulation

In this section, we describe the Time-Constrained Project Scheduling Problem (TCPSP) and formulate it as an integer linear program (ILP). Since distinguishing between regular time and overtime is crucial for the TCPSP, we first discuss (in Sect. 3.1) how time is divided in different units, what the cost structure of working in these different time units is, and what non-preemptiveness of jobs means. After that, Sects. 3.2 and 3.3 give a formal problem formulation of the TCPSP and a corresponding ILP model. Due to the complexity of the problem, we cannot expect to solve the TCPSP via this ILP-model within reasonable time. However, we present the ILP, since the heuristic presented in Sect. 4 makes use of the ILP formulation to construct feasible solutions and to perform a neighborhood search.

3.1 Modeling regular time and overtime

In project scheduling, a time horizon $[0, T)$ is divided into T time units, $t = 0, \ldots, T-1$, where time unit t represents the time interval $[t, t+1)$. The granularity of this discretization depends on the granularity of the input data. In the following, we will use T to refer to both the set of time units and the horizon $[0, T)$. However, for the Time-Constrained Project Scheduling Problem, this is not enough. The time units that represent the overtime have different properties than the time units that represent the regular time of a work day, e.g., different cost for hiring.

Fig. 1 Chains of time units



Therefore, we take a different view on the time units. We define one chain of regular time units $T^0 = \{t^0_1, \ldots, t^0_{N_0}\}$, and L chains of time units that are available for working in overtime $T^l = \{t^l_1, \ldots, t^l_{N_l}\}$, $l = 1, \ldots, L$. For each chain of overtime time units T^l, there is an index $\tau^l \in \{1, \ldots, N_0\}$ that indicates the last regular time unit ($t^0_{\tau^l} \in T^0$) before the start of the chain T^l. Furthermore, we assume that the chains do not overlap in time and that the overtime chain T^{l+1} is later in time than the overtime chain T^l. If the first time unit is not a regular time unit but an overtime time unit, we introduce an artificial regular time unit to start with. The corresponding set of time units T is the union of all chains, i.e., $T = \bigcup_{l=0}^{L} T^l$. As a consequence, each time unit $t \in T$ belongs to one unique chain T^l, $l \in \{0, \ldots, L\}$. Due to the above mentioned constraints, the set of time units itself is also a chain, so we can compare each pair of time units, no matter whether these time units are in regular time or in overtime. Note that it is possible that certain regular time units are followed by more than one chain of overtime time units.

Consider the following example. For processing, we have 8 regular hours available on Friday and 8 on Monday, and 4 overtime hours on Friday evening, 8 on Saturday, and again 4 on Monday evening. This means that $T^0 = \{F_1, \ldots, F_8, M_1, \ldots, M_8\}$, $T^1 = \{F_9, \ldots, F_{12}\}$, $T^2 = \{S_1, \ldots, S_8\}$ and $T^3 = \{M_9, \ldots, M_{12}\}$. Furthermore, $t^0_{\tau^1} = t^0_{\tau^2} = F_8$ and $t^0_{\tau^3} = M_8$. Figure 1 illustrates this example.

As in most project scheduling problems, preemption is not allowed during regular time. However, for the problem at hand, it is allowed to work (without gaps) in some of the time units in overtime, then stop and continue the work in the next regular time unit or in a subsequent chain of overtime, if no regular time is in between these two chains of overtime. This modified non-preemption requirement can be formally stated as follows. If a job is processed in two time units of a chain T^l, it is also processed in all time units in between these two time units, and if a job is processed in two different overtime chains T^k and T^l, $1 \le k < l \le L$, then this job has to be processed in all regular time units $\{t^0_{\tau^k+1}, t^0_{\tau^k+2}, \ldots, t^0_{\tau^l}\} \subset T^0$.

3.2 TCPSP with working in overtime, and hiring in regular time and in overtime

For the TCPSP, a set of n jobs, $\{J_1, \ldots, J_n\}$, each job J_j with a release date r_j and a deadline d_j, has to be scheduled without preemption (according to the modified non-preemption requirement of Sect. 3.1) for p_j time units on a time horizon $[0, T)$. This time horizon is divided into one chain of regular time units and multiple chains of overtime time units (as in Sect. 3.1). The release date r_j gives the first time unit in which job J_j is allowed to be processed and its processing has to be finished before d_j, i.e., $d_j - 1$ is the last time unit where it is allowed to work on job J_j. For processing the jobs a set of K resources, $\{R_1, \ldots, R_K\}$, is available, where resource R_k has a capacity Q_{kt} in regular time unit t. To hire one extra unit of resource R_k in time unit $t \in T^0$, an amount c^H_{kt} has to be paid. In an overtime time unit $t \in T \setminus T^0$ the use of one unit of the available resource R_k costs c^O_{kt} and hiring one extra unit costs c^{OH}_{kt}. It is assumed that the amount of regular resource available in overtime T^l, is equal to the regular resource capacity in the last regular time unit, i.e., $Q_{k,t^0_{\tau_l}}$. There is no limitation to the amount of resource hired. The processing of the jobs is restricted by precedence relations, which are given by sets of jobs P_j, denoting all direct predecessors of job J_j, that have to complete before J_j starts. Each job J_j has a specified processing time p_j and during the processing of job J_j it requires q_{jk} units of resource R_k.

Motivated by practice we incorporate one additional requirement on working in overtime. If personnel work during one time unit in overtime, they have to be present from the beginning of that overtime chain until that time unit. Although, they might not have to work immediately. This requirement reflects the fact that normally personnel start working in overtime immediately after the regular time, and then work for a continuous period in that one overtime chain. Thus, the amount of available resource used in an overtime chain is non-increasing.

3.3 ILP-formulation of the TCPSP

As a generalization of the classical TCPSP (see Neumann et al. 2002), the considered problem is also NP-hard. To model

the TCPSP, we employ one type of binary decision variables x_{jt} that are equal to 1 if job J_j is being processed in time unit t. To formulate the problem as an ILP, we use four types of variables that can be deduced from the variables x_{jt}. We use binary variables s_{jt} that are equal to 1 if time unit t is the first time unit where job J_j is being processed. The nonnegative variables H_{kt} represent the amount of capacity hired of resource R_k in time unit $t \in T^0$, and nonnegative variables O_{kt} and HO_{kt} represent the amount of capacity made available through working in overtime and hiring in overtime for resource R_k in time unit $t \in T \setminus T^0$, respectively.

Using these variables, the TCPSP can be modeled by the following ILP:

$$\text{minimize: } \sum_{k=1}^{K}\left[\sum_{t\in T^0} c_{kt}^{H} H_{kt} + \sum_{t\in T\setminus T^0}\left(c_{kt}^{O} O_{kt} + c_{kt}^{HO} HO_{kt}\right)\right] \tag{1}$$

subject to:

$$\sum_{t=r_j}^{d_j-1} x_{jt} = p_j, \quad \forall j; \tag{2}$$

$$\sum_{j=1}^{n} q_{jk} x_{jt} \le Q_{kt} + H_{kt}, \quad \forall k, t \in T^0; \tag{3}$$

$$\sum_{j=1}^{n} q_{jk} x_{jt} \le O_{kt} + HO_{kt}, \quad \forall k, t \in T \setminus T^0; \tag{4}$$

$$O_{k,t_1^l} \le Q_{k,t_\tau^0}, \quad \forall l \ge 1, k; \tag{5}$$

$$O_{k,t_h^l} \le O_{k,t_{h-1}^l}, \quad \forall l \ge 1, h > 1, k; \tag{6}$$

$$\sum_{t=r_j}^{d_j-p_j} s_{jt} = 1, \quad \forall j; \tag{7}$$

$$x_{j,t_1^0} = s_{j,t_1^0}, \quad \forall j; \tag{8}$$

$$x_{j,t_h^0} \le x_{j,t_{h-1}^0} + s_{j,t_h^0} + \sum_{t\in\bigcup_{\{l|\tau^l=h-1\}} T^l} s_{j,t}, \quad \forall j, h \ge 1; \tag{9}$$

$$x_{j,t_1^l} \le x_{j,t_\tau^0} + s_{j,t_1^l} + \sum_{t|t<t_1^l, t\in\bigcup_{\{k|\tau^k=\tau^l\}} T^k} s_{j,t}, \quad \forall l \ge 1, j; \tag{10}$$

$$x_{j,t_h^l} \le x_{j,t_{h-1}^l} + s_{j,t_h^l}, \quad \forall h > 1, l \ge 1, j; \tag{11}$$

$$p_i(1 - s_{j,t}) \ge \sum_{\bar{t}\ge t} x_{i\bar{t}}, \quad \forall t \in T, J_j, J_i \in P_j; \tag{12}$$

$$s_{jt} = 0, \quad \forall t \notin \{r_j, \ldots, d_j - p_j\}; \tag{13}$$

$$x_{jt} = 0, \quad \forall t \notin \{r_j, \ldots, d_j - 1\}; \tag{14}$$

$$x_{jt}, s_{jt} \in \{0, 1\}, \quad \forall j, t \in T; \tag{15}$$

$$H_{kt} \ge 0, \quad \forall k, t \in T^0; \tag{16}$$

$$O_{kt}, HO_{kt} \ge 0, \quad \forall k, t \in T \setminus T^0. \tag{17}$$

The objective function (1) minimizes the total costs. Constraint (2) ensures that each job is processed for the required duration between the release date and the deadline. Constraint (3) ensures that the amount of required resource does not exceed the amount of available resource in regular time units, using regular capacity and hiring. Constraint (4) ensures that the resource usage in an overtime time unit cannot exceed the amount that is available through working in overtime and hiring in overtime. Constraints (5) and (6) force the amount of work in overtime to be non-increasing in each chain of overtime, and not to exceed the capacity of the regular resource in the last regular time unit. Constraint (7) ensures that each job starts exactly once. The modified non-preemption requirement is specified in constraints (8) to (11). In constraint (8), we guarantee that job J_j is processed in the first time unit of T^0 if it also starts in that time unit. For the other regular time units, constraint (9) ensures that job J_j can only be processed if it is processed in the previous regular time unit, or starts in this regular time unit, or starts in an overtime time unit directly succeeding the previous regular time unit. For processing in overtime time units, constraint (10) states that we are allowed to work on a job J_j in the first time unit of an overtime chain if we work on it in the last regular time unit, or we start the job at this time unit, or the job starts in an overtime time unit which succeeds the last regular time unit, but preceding this overtime time unit. Constraint (11) states that it is only allowed to work on a job J_j in a time unit that is not the start of an overtime chain, if that is done also in the previous overtime time unit or it is starts in this overtime time unit. The precedence relations are managed in constraint (12). If job J_j starts in time unit t, then the left hand side of the constraint becomes zero, implying that in none of the time units after t there can be worked on job J_i, i.e., job J_i is finished before t. On the other hand, if job J_j does not start in time unit t, constraint (12) gives no restriction. Constraints (13) and (14) put all non-relevant s_{jt} and x_{jt} variables to zero. Finally, constraints (15) to (17) define the domain of the variables.

4 Solution approach

In the previous section, we presented an ILP-formulation of the TCPSP. However, we cannot expect to solve large instances with the ILP-formulation. In this section, we present a heuristic approach based on the concept of planning fractionally.

4.1 Two stage heuristic

4.1.1 Outline

After initialization, in which feasibility of the instance is checked, the first stage constructs a number of different partial schedules by randomized sampling. Partial schedules are constructed by scheduling one job at a time for only a fraction of its duration. The job to be scheduled next is selected with a probability derived from its deadline and the state of the schedule constructed so far. Once all jobs are partially admitted in the schedule, the fraction for which the jobs are scheduled is gradually increased. In this first stage, no use of overtime is made and, therefore, the resulting schedule in general does not contain the jobs for their complete duration. However, it is guaranteed that in the second stage the partial schedules can be made feasible by allowing the use of overtime. On a small subset of good feasible schedules, a neighborhood search will be performed.

4.1.2 Initialization

In the initialization stage, we calculate modified release dates and modified deadlines, which are sharp bounds on the start and completion times of the jobs. From these modified release dates and modified deadlines, it is possible to determine in advance whether there exists a feasible schedule.

The modified release date $\tilde{r}_j$ of a job J_j is the first time unit, such that if the job starts in this time unit, all its predecessors can be scheduled if there is abundant resource capacity, both in regular time and overtime. The modified deadline $\tilde{d}_j$ of a job J_j is the last time unit, such that if the job finishes the unit before, all its successors can still be scheduled if there is abundant resource capacity, both in regular time and overtime. A feasible schedule exists if and only if for each job J_j the interval $[\tilde{r}_j, \tilde{d}_j]$ is large enough to process the job.

The modified release dates $\tilde{r}_j$ can be calculated by a forward recursion through the precedence network:

$$\tilde{r}_j := \max\left\{r_j, \max_{i \in P_j}\{\tilde{r}_i + p_i\}\right\}, \quad \forall j. \tag{18}$$

With a backward recursion, we calculate the modified deadlines $\tilde{d}_j$:

$$\tilde{d}_j := \min\left\{d_j, \min_{\{i \mid j \in P_i\}}\{\tilde{d}_i - p_i\}\right\}, \quad \forall j. \tag{19}$$

All time windows, the time between the modified release date and modified deadline, should be large enough to accommodate the job:

$$\tilde{d}_j - \tilde{r}_j \geq p_j, \quad \forall j. \tag{20}$$

Note that the modified release dates and modified deadlines are calculated with respect to the time horizon T.

4.1.3 Stage 1

In the first stage, we generate a schedule containing all jobs in which jobs get intentionally scheduled for a shorter duration than necessary, using only regular time units. For each job J_j, we are going to determine a start time $S_j \geq \tilde{r}_j$, a completion time $C_j < \tilde{d}_j$ such that in $[S_j, C_j]$ there is enough time available to process the job (with respect to T). However, at this first stage, we assign only the regular time in $[S_j, C_j]$ to job J_j. To start, we *aim* to create a partial schedule such that each job is scheduled for at least a fraction $a_0 \in (0, 1]$. However, there is no guarantee that each job reaches this fraction, but we do ensure that the assigned starting and completion times allow for a feasible solution when using overtime and hiring extra capacity are allowed. Next, we describe this selecting and scheduling in detail.

We generate the partial schedules with a randomized sampling procedure and then try to improve them by increasing the fraction for which jobs are scheduled. In a serial manner, we select jobs and include them in the schedule. Let D_{jobs} denote the decision set from which we select the job to be scheduled next. The set D_{jobs} contains all jobs for which all predecessors are already in the current schedule. Initially,

$$D_{\text{jobs}} := \{J_j \mid P_j = \emptyset\}.$$

In each iteration, we select a job from the decision set D_{jobs}. For each job $J_j \in D_{\text{jobs}}$, we determine a priority v_j. This priority depends on how early the job can start. Let e_j denote the earliest start time of job J_j in the following sense: e_j is the earliest time unit in T, greater than or equal to the modified release date $\tilde{r}_j$ and greater than the completion times of job J_j's predecessors, such that for each of the $\lceil a_0 \cdot p_j \rceil$ regular time units following e_j still enough resource capacity is available to schedule job J_j. This is done, because at this stage we only try to schedule job J_j for a fraction a_0 in only the regular time T^0.

For each job J_j, we define a slack, sl_j, with respect to T^0 (see Fig. 2):

$$sl_j := \tilde{d}_j - \lceil a_0 \cdot p_j \rceil - e_j, \quad \forall J_j \in D_{\text{jobs}}.$$

It is possible that the slack of a job becomes negative, implying that we cannot schedule the job for a fraction a_0 without hiring or working in overtime. As a consequence, we either have to schedule this job for a smaller fraction

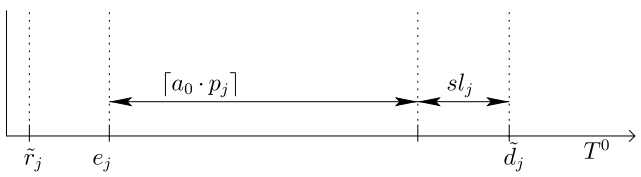


Fig. 2 Derivation of the slack of job J_j

at this stage or hire a small amount of resources in regular time to enlarge the fraction for which this job is scheduled in regular time. We get back to this problem later.

If a job has a small slack value, there is not much room to maneuver this job, and therefore we prefer to schedule this job next. We give such a job a high priority. More precisely, the priority value of job J_j becomes:

$$v_j := \max_{J_i \in D_{\text{jobs}}} \{sl_i\} - sl_j, \quad \forall J_j \in D_{\text{jobs}}.$$

Note that $v_j \geq 0$. To get strictly positive selection probabilities for each job in D_{jobs}, we add 1 to the priority and normalize these priority values. The resulting selection probability η_j of job J_j is:

$$\eta_j := \frac{(v_j + 1)^\alpha}{\sum_{J_i \in D_{\text{jobs}}} (v_i + 1)^\alpha}, \quad \forall J_j \in D_{\text{jobs}},$$

where the value of α lies in $[0, \infty]$, and indicates the importance of the priority value when selecting a job. If α equals 0, jobs are selected uniformly at random from D_{jobs}. If α tends to infinity, the job with the highest priority gets selected deterministically.

Now that we have selected a job, we determine its start and completion time, $S_j \in T$ and $C_j \in T$, respectively. Since the job J_j gets scheduled in the regular time units in $[S_j, C_j]$, for S_j and C_j the following must hold to ensure that we get a large enough interval for job J_j and leave enough room for the remaining jobs to be scheduled:

$$S_j \geq \tilde{r}_j, \tag{21}$$

$$C_j < \tilde{d}_j, \tag{22}$$

$$S_j > C_i, \quad \forall J_i \in P_j, \tag{23}$$

$$C_j - S_j + 1 \geq p_j \quad \text{with respect to } T. \tag{24}$$

There can be many pairs (S_j, C_j) satisfying these four constraints. Therefore, we select a pair by applying the following criteria, in the presented order:

1. Select a pair that hires as little as possible.
2. Select a pair that minimizes $\max\{0, \lceil a_0 \cdot p_j \rceil - (C_j - S_j)\}$ with respect to T^0.
3. Select a pair that minimizes C_j.
4. Select a pair that maximizes S_j.

After the first three criteria there may still be several pairs left, whereas after Criterion 4 the values for S_j and C_j are uniquely determined. Due to constraints (21) to (24), it might be necessary to hire resources in regular time, but via criterion 1 we try to avoid this. The second criterion states that we select from the remaining start and completion times, those that schedule the job with minimal shortage to the fraction a_0. This way we try to schedule each job close to the desired fraction a_0. The third criterion gives, from the remaining start and completion times, those that minimize the completion time, not to hinder the jobs that still have to be scheduled. Finally, we choose from the remaining pairs of start and completion time, the pair that maximizes S_j.

If all jobs are partially scheduled, it might be possible to extend the processing time of jobs within regular time, such that they are scheduled for a larger fraction. To prevent the shifting of problematic situations (as described in Sect. 2), we would like to spread this increase evenly over all jobs. Therefore, we use a procedure that repeatedly tries to extend the jobs to a higher fraction, going through the schedule alternating from back to the front and from the front to the back. Let $(a_0, a_1, \ldots, a_k)$ denote a non-decreasing sequence of fractions that we apply, where a_0 equals the fraction used in the randomized sampling.

One extension step consists of a backward and a forward extension. In the ith backward extension, we go through the current schedule from the back to the front and search for each job J_j a new pair (S_j, C_j) satisfying (21) to (24), by the following four criteria in the presented order:

1. Select a pair that requires no more hiring than before.
2. Select a pair that minimizes $\max\{0, \lceil a_i \cdot p_j \rceil - (C_j - S_j)\}$ with respect to T^0.
3. Select a pair that maximizes S_j.
4. Select a pair that minimizes C_j.

The ith backward extension is followed by the ith forward extension, which is a mirrored version of the ith backward extension.

4.1.4 Stage 2

The result of Stage 1 is a schedule containing all jobs, scheduled in regular time units and not necessarily for the required length. In Stage 2, we use working in overtime and hiring in regular time and overtime to get a feasible solution of the TCPSP. The main idea to get a feasible solution is the following. Iteratively, for each job J_j that is not scheduled for its required duration, we solve an ILP. This ILP is given by a restricted version of the ILP in Sect. 3.3, where all jobs except job J_j get 'frozen' as they are in the current schedule and only the timing of job J_j is left to the solver. More precisely, we deduce a release date and a deadline for job J_j, imposed by the current schedule and the original release date and deadline. Furthermore, we deduce the regular capacity that is still available for scheduling job J_j. For the regular time units, this is the original capacity minus the capacity used by the other 'frozen' jobs in the current schedule. For the overtime time units, this is the work in overtime that is imposed by other jobs, but not used as a consequence of constraint (6). The solver returns the timing of the job J_j and the corresponding use of irregular capacity. This, together with

the 'frozen' jobs, gives a schedule that is the same as before except for the scheduling of job J_j. Job J_j is now scheduled for its required duration. Note that the requirements (21) to (24) for the start and completion times S_j and C_j of job J_j ensure that there is always a feasible scheduling of job J_j. At the end of this iterative process, we have a feasible solution for the TCPSP.

The order in which the jobs are extended to their required duration can be chosen through numerous criteria. Possible orderings are: smallest scheduled fraction ($\bar{a}_j$) first, largest unscheduled processing time ($(1-\bar{a}_j)p_j$) first, and an ordering deduced from the precedence network.

Now that we have a feasible schedule for the TCPSP, we apply a neighborhood search to improve upon this schedule. We use a neighborhood search based on a method proposed by Palpant et al. (2004). This method selects a number of jobs, 'freezes' the remainder of the schedule, and calculates for the resulting ILP an optimal schedule. It is similar to the first part of Stage 2, but now the timing of a small number of jobs is left to the ILP-solver. One iteration of the neighborhood search consists of the following steps:

1. Select a subset of the jobs, $J^{\text{Neighbor}} \subset \{J_1, \ldots, J_n\}$.
2. 'Freeze' all jobs $J_j \notin J^{\text{Neighbor}}$.
3. Determine release dates, deadlines, available capacities for the jobs in J^{Neighbor}.
4. Solve the resulting ILP.

There are numerous ways to select a subset J^{Neighbor} of jobs. For example, Palpant et al. propose (1) to select a job together with all its predecessors, or (2) select a job and all jobs scheduled parallel with, and contiguous to it. One can think of many more selection criteria, but the main idea is not to select jobs arbitrarily, but to select jobs that occur close to each other in the schedule. Otherwise, there is not much to improve, i.e., the neighborhood is too small.

5 Computational results

This section describes the setup of the computational tests and the results for testing the solution approach presented in this paper. Since this is the first attempt to tackle the TCPSP with working in overtime and hiring in regular time and overtime, the presented heuristic cannot be compared with any existing method. However, there is much known for the RCPSP. Therefore, we take benchmark instances of the RCPSP and transform them into instances of the TCPSP. This is done in such a way that we can draw conclusions with respect to the quality of the TCPSP solutions. This section starts with describing the transformation of the instances and the parameter setting in the heuristic, before presenting the computational results.

5.1 Construction of TCPSP instances

For the RCPSP, a set of benchmark instances called PSPlib generated by Kolisch and Sprecher (1997b) and Kolisch et al. (1995) can be found on the web (Kolisch and Sprecher 1997a). These instances have been employed in many studies, for example by Demeulemeester and Herroelen (1997), Kolisch and Drexl (1996), and Kolisch et al. (1995). These RCPSP instances form the base of the TCPSP instances. To transform the instances from the PSPlib into TCPSP instances, three additional aspects have to be introduced: overtime and hiring possibilities with their associated costs, and deadlines.

The TCPSP distinguishes between regular and overtime time units, where the RCPSP has only one type of time units. We let each time unit from the RCPSP correspond to a regular time unit in the TCPSP. In addition, we introduce overtime in a weekly pattern. Day 1 of the week, the Sunday, contains only 8 overtime time units. Days 2 to 6, the weekdays, start with 8 regular time units, followed by 4 overtime time units. Day 7, the Saturday, contains again only 8 overtime time units. This weekly pattern is repeated until the largest deadline of the jobs is reached. All other aspects, like job durations, precedence relations, resource availability, and resource requirements remain unchanged.

To get insight in the quality of the solution generated by the two stage heuristic, we compare the TCPSP solution with the RCPSP solution. A comparison can be made if we set all costs equal to 1 and let the deadline of all jobs be the (best known upper bound on the) minimum makespan of the RCPSP instance. This means that the number of regular time units before the deadline in the TCPSP, is exactly the (best known upper bound on the) minimum makespan of the RCPSP instance. The quality of the schedule is given by its costs. If a schedule has zero costs, the two stage heuristic gives an optimal (best known) schedule for the RCPSP instance. If not, the costs of a schedule give an indication on how far we are from the best known schedule, i.e., they give the amount of irregular capacity used to reach the best known makespan.

Setting all costs equal to 1 implies that working in overtime is equally costly as hiring in regular time or overtime. This does not fit the real world situation, but it allows us to measure the total amount of irregular capacity needed. Note that due to constraint (6) and all costs equal to 1, hiring in overtime is at least as good as working in overtime. Therefore, the possibility of working in overtime could be removed from the model. However, we choose not to do this, since it would reduce the computational time and thereby give a false indication on the computational time of real life instances.

Besides choosing the deadline equal to the (best known upper bound the) minimum makespan, which we denote by

Table 1 Different values of α on the 100 schedules generated

# iterations	2	2	4	4	4	4	5	5	5	5	10	10	10	30
α	100	50	20	10	9	8	7	6	5	4	3	2	1	0

Table 2 Different values of a_0 on instances with 30 jobs and $b = 0.9$

RS a_0	Extension seq. $a_1, a_2, \ldots$	Average percentage planned after RS	Average percentage planned after extension	Average cost after feasibility
0.5	0.6, 0.7, 0.8, 0.9, 1	78.0	88.9	97.4
0.6	0.7, 0.8, 0.9, 1	80.2	89.1	90.5
0.7	0.8, 0.9, 1	83.7	90.3	91.1
0.8	0.9, 1	87.3	91.0	76.2
0.9	1	88.0	90.7	87.0
1	–	88.0	88.0	154.4

$C_{\max}$, it can be chosen as a fraction of $C_{\max}$. By letting the deadline be equal to $\lceil b \cdot C_{\max} \rceil$, where $b \in (0, 1]$, the problem becomes tighter, and more irregular capacity will be needed. The resulting objective value will indicate the costs to complete the project earlier. Since the problem becomes tighter with $b < 1$, we get a better insight in the influence of the different parameter settings of the two stage heuristic.

5.2 Parameter setting

In each stage of the heuristic, there are a number of parameters that need to be set. In the first stage, these are the fraction a_0, the α value for the randomized sampling, and the extension sequence in the improvement. In the second stage, there is the order in which the jobs are extended to their required duration, and the choice of the neighborhood for the improvement. This subsection concerns the setting of these parameters.

Since there are far too many different parameter settings to test all possible combinations, we determine the parameter setting one parameter after the other and evaluate the achieved result after the feasibility step (and not after the neighborhood search). For each instance, we generate a number of partial schedules (by random sampling), extend them, and turn them into a feasible schedule. Out of these schedules, we select the one with minimum costs to be the solution.

For testing the parameter settings, we use a selection of 10 instances with 30 jobs and 10 instances with 120 jobs from the PSPlib. Initially, we use a deadline of 90% of the best known upper bound on the makespan, i.e., $b = 0.9$. We are then quite sure that the objective value will not equal zero, allowing us a better measurement of the effect of the parameters. For these initial tests, we use *largest unscheduled processing time first* as the priority rule in the feasibility step.

To determine good values for α, we fix all other parameters. When testing different values for α, it turns out that if a small number of schedules for each instance are generated, large values for α outperform small values. However, as the number of schedules generated per instance increases, the random search ($\alpha = 0$) outperforms the higher values for α. Therefore, we conclude that it is best to start with a high value for α and decrease it as more schedules for the same instance are being generated. From these tests, we conclude that taking 100 randomized samples gives enough diversity. Table 1 presents the decreasing values of α we use for the 100 random samples. It states that the first two random samples are taken with an α value of 100, the next two with value of 50, and so on. These values are used in the remainder of the computational experiments.

For the randomized sampling (RS) to generate partial schedules, we need an initial fraction a_0. If we choose a_0 too close to 1, we do not benefit from the idea that scheduling only a fraction prevents the shifting of problems toward the deadlines. If we choose a_0 too small, we observed that the jobs are pulled to the front of the scheduling period, causing problematic situations at the beginning of the horizon. Tables 2 and 3 report on tests using different values of a_0. Within the tests, an extension with step size 0.1 is chosen. The tables give the results after the randomized sampling, after the extension, and after the feasibility procedure. Before the feasibility procedure, the costs of a schedule have no meaning, and we therefore display the average percentage planned. If we only consider the average fraction planned after the randomized samples, $a_0 = 0.9$ is best. However, with the extension and feasibility, $a_0 = 0.8$ is best. Note that the average cost for instances with 120 jobs is about 4 times as large as the average cost for instances with 30 jobs. This is due to the fact that the instances with 120 jobs approximately have a 4 times higher total resource requirement (*work content*), while the deadlines are about the same.

Table 3 Different values of a_0 on instances with 120 jobs and $b = 0.9$

RS a_0	Extension seq. $a_1, a_2, \ldots$	Average percentage planned after RS	Average percentage planned after extension	Average cost after feasibility
0.5	0.6, 0.7, 0.8, 0.9, 1	82.9	92.6	467.3
0.6	0.7, 0.8, 0.9, 1	84.7	92.7	439.4
0.7	0.8, 0.9, 1	86.9	93.7	417.2
0.8	0.9, 1	89.4	93.8	357.9
0.9	1	92.8	92.8	445.7
1	–	90.1	90.1	580.4

Table 4 Different extension sequences, 120 jobs, $b = 0.9$, and $a_0 = 0.8$

Extension seq. $a_1, a_2, \ldots$	Average percentage planned after RS	Average percentage planned after extension	Average cost after feasibility
0.8, 0.9, 1	89.4	93.3	386.9
0.9, 1	89.4	93.8	357.9
1	89.4	93.6	370.3
–	89.4	89.4	539.8

As already can be seen from Tables 2 and 3, the extension step is important. In the next series of tests, we have applied four sequences: no extension, directly from $a_0 = 0.8$ to $a_1 = 1$, using sequence (0.9, 1), and using sequence (0.8, 0.9, 1). Comparing the resulting schedules when using no extension at all with an extension sequence equal to [0.9, 1], we observe a major improvement, see Table 4. With extension, the result is 1.5 times better. The result of using different extension sequences is less diverse. We choose to use the extension sequence [0.9, 1].

For Stage 2, we have to decide in which order we address the jobs to schedule them for the required duration. We compare: *largest unscheduled processing time first*, *smallest scheduled fraction first*, *smallest start time first*, and *largest start time first*. It turns out that between these different orderings there is no significant difference in the costs after the feasibility step. However, *largest unscheduled processing time first* and *smallest scheduled fraction first* require far less computational time, and the latter performs slightly better. Therefore, we choose to use *smallest scheduled fraction first*.

The neighborhood search is the most time consuming part of the heuristic, since it has to solve many ILP's. Therefore, we select out of the 100 constructed schedules of Stage 1 the one with lowest costs, and do only the neighborhood search on that schedule. Moreover, for the neighborhood search, it is important to keep the running time low, but still search a large part of the neighborhood. In each step, a number of jobs are removed from the schedule, the other jobs are fixed before we reinsert the removed jobs optimally into the schedule. To do this, we choose a point in time and remove each job that is contiguous to it. Due to the concept of overtime we define a job contiguous to time t if its start time is at most the first regular time unit after t and its completion time is at least the last regular time unit before t. By choosing a single point in time, we keep the number of removed jobs small, and all these jobs are close together. The set of considered points in time is chosen as the set of completion times of the jobs in the schedule. We process these points in an increasing order. Always when an improvement occurs, we replace the current schedule by the new schedule. We do not recalculate the set of time points to be considered. One option is to go through the schedule only once (a single pass); another option is to go through the schedule several times (a multi pass). If we do a multi pass, we use in each pass the new completion times, and stop if the last pass does not improve the schedule. From our tests, we have seen that it can take a long time to determine the optimal placements of the removed jobs, i.e., solve the ILP. We can restrict the computational time spent on one reinsertion by using time limits. If we reach such a time limit, we use the best found reinsertion of the removed jobs (this can be the placement we had before, so we are guaranteed to have a solution that is at least as good as before). Table 5 displays the quality against time trade-off. ($CT(s)$ stands for computational time in seconds.) Multi pass gives solutions of higher quality at a price of larger computational times; single pass with time limit (we used a 10 second time limit) gives a solution very fast, but one with higher costs. From the second and third column in Table 5, we see that there are a few instances with very large computational time. We have observed that there are only a few instances in which the time limit of 10 seconds is reached. Therefore, it pays to limit the computational time spent on one reinsertion. Comparing the values in Tables 4 and 5, we see that the neighborhood search reduces the objective value by 20% to 25%.

Table 5 Quality time trade-off, 120 jobs, $b = 0.9$

Strategy	Average $CT(s)$	Max $CT(s)$	Objective
Single pass/No time limit	3,700.8	32,919.4	291.0
Single pass/10 s time limit	84.7	192.6	289.1
Multi pass/No time limit	3,900.9	33,385.3	264.1
Multi pass/10 s time limit	315.8	750.7	265.6

Table 6 Summary of the results

Jobs	Instances	Work content	$b = 1.0$				$b = 0.9$		
			Maximum $CT(s)$	Average $CT(s)$	Objective value	# with cost = 0	Maximum $CT(s)$	Average $CT(s)$	Objective value
30	480	2,310.7	26.1	2.9	3.2	294	55.1	4.5	58.6
60	480	4,562.9	241.7	17.2	13.4	276	303.4	24.7	124.4
90	480	6,812.2	932.7	53.0	22.6	282	845.3	73.9	180.8
120	600	9,073.5	2,163.4	308.3	72.4	75	2,798.2	362.2	326.6

From this subsection, we may conclude that each step in the presented method has its contribution to finding a good schedule, and that the presented method is very flexible. Depending on the purpose of use, not only the parameters can be chosen appropriately, but there is also a choice in the quality against time trade-off.

5.3 Computational results

In the previous subsection, we have used only a small number of instances to determine the choices for the parameters of the two stage approach. In this subsection, we present a summary of the computational results for a large set of instances. We have used all single mode instances from the PSPlib and have set the algorithm as discussed in the previous subsection with a multi pass, 10 second time limited, neighborhood search. For each instance, we construct a schedule with a deadline on 100% and 90% of $C_{\max}$. Table 6 summarizes the test results. The computational experiments were performed on a computer with a Intel Centrino processor running at 2.0 GHz. We used Delphi 7 to code the algorithm and CPLEX 9.1 to solve the ILP's. The details of the computational tests can be found on a website (Guldemond et al. 2006).

If we set the upper bound equal to $C_{\max}$ and solve an instance with zero cost, this means that we have an optimal (or best known) schedule for the RCPSP. We see that our method solves about 60% of the instances with zero cost for the 30, 60, and 90 job instances, but only 13% of the 120 job instances. The 30, 60, and 90 job instances are generated with similar characteristics, whereas the 120 job instances have a lower relative resource availability and are therefore tighter and more difficult. The average objective value, the amount of used irregular capacity, is only a small fraction of the total work content. In the solutions of the instances with 30 jobs, on average, 0.14% of the required capacity is satisfied with irregular capacity. This percentage goes up to 0.33% for the instances with 90 jobs. For the tighter 120 job instances, the percentage of work done with irregular capacity is 0.80%. These percentages are very low, and therefore we can conclude that the achieved schedules are of high quality.

Setting the deadline to 90% of $C_{\max}$, we can no longer verify whether or not optimal schedules are found. As expected, the objective values increase. However, the 10% reduction of the time horizon results in schedules that have only 2.5% of the total work content in irregular capacity for the 30 job instances, and up to 3.5% for the 120 job instances. So completing a project 10% earlier does not have to be too costly.

The computational time grows as the number of jobs grows, but the relative resource availability seems more important. The instances that have a low resource availability require more computational time than instances with high availability. The instances that require a lot of computational time are exactly those for which the corresponding RCPSP instances are also difficult and where the best found makespan is often not proven to be optimal.

6 Extensions of the TCPSP

In practice a more elaborate modeling of the project might be required, such as including multi-mode scheduling of jobs and time-lags on precedence relations. In this section we indicate how to extend the presented heuristic in these cases.

6.1 Multi-mode TCPSP

One possible extension of the TCPSP is by allowing multiple modes for scheduling the jobs. Then, for each job J_j a set M_j of different execution modes is given. Each mode $m \in M_j$ has a specified processing time p_{jm} and during the processing of job J_j in mode m it requires q_{jmk} units of resource R_k.

The presented solution approach can easily be extended to the multi-mode TCPSP. In the randomized sampling one not only has to select a job, but also a corresponding mode. This mode can be fixed until a feasible schedule is reached.

Once the multi-mode is incorporated, the presented model cannot only deal with working in overtime and hiring in regular time and overtime, but also with the possibility to outsource. Outsourcing can be included by introducing a mode for each job which represents the outsourcing. We introduce for such an outsource mode a processing time and a resource requirement for an artificial resource that has to be hired. The processing time of this mode and the cost for hiring correspond to the outsourcing.

6.2 Including time-lags in the model

Within the new concept of time chains of Sect. 3.1, using time-lags on precedence relations can lead to problems. They are not properly defined, since it is not clear whether the time-lags only refer to regular time units or also to overtime time units. Consider the following example. If the time-lag is a consequence of a lab test that has to be done between two processes, the opening hours of the lab determine to which time units the time-lag applies. To overcome this problem, it is possible to introduce a dummy job and a dummy resource for each time-lag. With the proper resource requirements and resource availability for each dummy job and dummy resource, time-lags can have any desired property with respect to regular and overtime time units. Therefore, our approach can also deal with time-lags.

7 Conclusions

In this paper, a new scheduling methodology is presented for scheduling jobs with strict deadlines. First jobs are scheduled for only a fraction for their required duration and then the fraction for which the jobs are scheduled is gradually increased. This idea is applied to find solutions for the TCPSP with hiring and working in overtime.

Since there are no benchmark instances for the TCPSP, we used benchmark instances from the RCPSP to get insight in the quality of the achieved schedules. It turned out that a large amount of the instances are solved to optimality. Decreasing the deadline of a project by 10% results in schedules that have far less than 10% of the work done with irregular capacity. Thus, we can state that the schedules generated by the two stage heuristic are of high quality. The computational tests also show that there is a lot of flexibility in the developed method. The flexibility is not only due to the parameter setting, but also due to the possibility to choose where to spend the computational effort. Therefore, we believe this method is very suited for practical use.

The computational tests demonstrate the effectiveness of the fractional scheduling method. We believe this methodology may also work well for other scheduling problems with strict deadlines. Moreover, the method may work well for any scheduling problems with a non-regular objective function, like earliness/tardiness problems.

Acknowledgements The authors are grateful to the anonymous referees for the helpful comments on an earlier draft of the paper. Part of this research has been funded by the Dutch BSIK/BRICKS project.

References

Deckro, R. F., & Herbert, J. E. (1989). Resource constrained project crashing. *OMEGA International Journal of Management Science*, *17*, 69–79.

Demeulemeester, E. (1995). Minimizing resource availability costs in time-limited project networks. *Management Science*, *41*(10), 1590–1598.

Demeulemeester, E., & Herroelen, W. (1997). New benchmark results for the resource-constrained project scheduling problem. *Management Science*, *43*, 1485–1492.

Gademann, N., & Schutten, M. (2005). Linear-programming-based heuristics for project capacity planning. *IIE Transactions*, *37*, 153–165.

Guldemond, T. A., Hurink, J. L., Paulus, J. J., & Schutten, J. M. J. (2006). *Time-constrained project scheduling; details of the computational tests*. http://tcpsp.ewi.utwente.nl/.

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

Kis, T. (2005). A branch-and-cut algorithm for scheduling of projects with variable intensity activities. *Mathematical Programming*, *103*, 515–539.

Kolisch, R. (1995). *Project scheduling under resource constraints*. Berlin: Physica.

Kolisch, R., & Drexl, A. (1996). Adaptive search for solving hard project scheduling problems. *Naval Research Logistics*, *43*, 23–40.

Kolisch, R., & Hartmann, S. (1999). Heuristic algorithms for solving the resource-constrained project scheduling problem: classification and computational analysis. In J. Weglarz (Ed.), *Handbook on recent advances in project scheduling* (pp. 197–212). Dordrecht: Kluwer.

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

Kolisch, R., & Sprecher, A. (1997a). *Project scheduling library—PSPlib*. http://129.187.106.231/psplib/.

Kolisch, R., & Sprecher, A. (1997b). PSPLIB a project scheduling problem library. *European Journal of Operational Research*, *96*, 205–216.

Kolisch, R., Sprecher, A., & Drexl, A. (1995). Characterization and generation of a general class of resource-constrained project scheduling problems. *Management Science*, *41*(10), 1693–1703.

Li, R. K.-Y., & Willis, R. J. (1993). Resource constrained scheduling within fixed project durations. *The Journal of the Operational Research Society*, *44*, 71–80.

Möhring, R. H. (1984). Minimizing costs of resource requirements in project networks subject to a fixed completion time. *Operations Research*, *32*(1), 89–120.

Neumann, K., Schwindt, C., & Zimmermann, J. (2002). *Project scheduling with time windows and scarce resources*. *Lecture notes in economics and mathematical systems* (Vol. 508). Berlin: Springer.

Palpant, M., Artigues, C., & Michelon, P. (2004). LSSPER: solving the resource-constrained project scheduling problem with large neighbourhood search. *Annals of Operations Research*, *131*, 237–257.