J Sched (2010) 13: 393–409
DOI 10.1007/s10951-010-0188-7

Scheduling technicians and tasks in a telecommunications company

Jean-François Cordeau · Gilbert Laporte · Federico Pasin · Stefan Ropke

Published online: 17 July 2010

Abstract This paper proposes a construction heuristic and an adaptive large neighborhood search heuristic for the technician and task scheduling problem arising in a large telecommunications company. This problem was solved within the framework of the 2007 challenge set up by the French Operational Research Society (ROADEF). The paper describes the authors' entry in the competition which tied for second place.

Keywords Heuristics · Simulated annealing · Large neighborhood search · Scheduling · Manpower planning

1 Introduction

This paper proposes a construction heuristic and an adaptive large neighborhood search heuristic for the technician and task scheduling problem (TTSP) in a large telecommunications company. These tasks can be maintenance, installation or construction jobs. The TTSP is a difficult real-life problem, introduced as the subject of the 2007 challenge set up by the French Operational Research Society (ROADEF) in collaboration with France Télécom. This challenge is organized on a different theme every second year. Challengers are given the definition of a hard optimization problem arising in practice and are asked to develop an algorithm for its solution. Several data sets are provided by the organizers for testing purposes but new data are used at the final stage of the competition. In 2007, 31 teams entered the challenge and 11 made it to the final stage. This paper describes the authors' entry in the competition which tied for second place.

The TTSP is defined in Dutot et al. (2006). We are given a set of tasks $N = \{1, \dots, n\}$ and a set of technicians $\mathcal{T} = \{1, \dots, m\}$. Each technician is proficient in a number of *skill domains*. There are q skill domains and a technician's level of proficiency in a given domain is described by an integer from 0 to p, where 0 means that the technician has no skill in the associated domain. We can express each technician's skills by a q-dimensional *skill vector* in which the l^{th} entry indicates the technician's skill level in the l^{th} skill domain.

The tasks vary in difficulty and some require more than one technician. The number of technicians required and the difficulty of a task i are described by a $p \times q$ *skill requirement matrix* $(s^i_{\alpha\beta})$. The columns in the matrix correspond to skill domains and the rows correspond to proficiency levels. The entry $s^i_{\alpha\beta}$ indicates the number of technicians with a level of at least α in domain β that are necessary to perform task i. Note that a requirement for a high skill level carries over to the lower skill levels. An example of a skill requirement matrix with four domains and three skill levels is given

J.-F. Cordeau (✉)
Canada Research Chair in Logistics and Transportation, HEC Montréal, 3000, chemin de la Côte-Sainte-Catherine, Montréal, Canada H3T 2A7
e-mail: jean-francois.cordeau@hec.ca

G. Laporte
Canada Research Chair in Distribution Management, HEC Montréal, 3000, chemin de la Côte-Sainte-Catherine, Montréal, Canada H3T 2A7
e-mail: gilbert.laporte@hec.ca

F. Pasin
Department of Logistics and Operations Management, HEC Montréal, 3000, chemin de la Côte-Sainte-Catherine, Montréal, Canada H3T 2A7
e-mail: federico.pasin@hec.ca

S. Ropke
Department of Transport, Technical University of Denmark, Bygningstorvet 115, 2800 Kgs. Lyngby, Denmark
e-mail: sr@transport.dtu.dk

below:

$$\begin{pmatrix} 1 & 2 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 2 \end{pmatrix}.$$

In this example the task requires one technician with a proficiency of at least 1 in domain 1. It requires two technicians to be proficient in domain 2: one must be at least a level 1 technician and the other must be at least level 3. No skilled technician in domain 3 is needed and two technicians with skill level 3 in the fourth domain are necessary.

Similarly, we represent the skills of technician j as a matrix $(v^j_{\alpha\beta})$. The skill matrix of a technician with skill vector $(2, 1, 1, 3)$, $p = 3$ and $q = 4$ is

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Technicians are grouped into teams in order to perform the tasks. If two technicians have skill vectors $(2, 1, 1, 3)$ and $(1, 3, 2, 3)$, then a team consisting of these two technicians would be able to serve a task with the above skill requirement matrix. The team would be overqualified for the task, but this is allowed. Usually it is impossible to perform all tasks of a TTSP instance in a single day and the tasks are performed over several days. A team must stay together on a given day, but can be broken up on the following day. A task can always be completed in a single day and the execution of a task cannot be interrupted nor split over several days. Technicians can be unavailable on specific days due to holidays or training. We will denote the set of technicians available on day k by $\mathcal{T}_k$.

Apart from the skill requirement matrix, each task $i \in N$ is characterized by a duration $d_i \in \mathbb{Z}^+$, an outsourcing cost $c_i \in \mathbb{Z}^+$, a set π_i of predecessor tasks, a set σ_i of successor tasks, and a priority level $p_i \in \{1, \ldots, 4\}$. The set π_i contains the tasks that must be completed before i starts and the set σ_i contains the tasks that cannot be started before task i is finished. It is assumed that $j \in \pi_i \Leftrightarrow i \in \sigma_j$ for all $i, j \in N$. The duration d_i states how long it takes to perform task i. This duration is constant and remains the same no matter how many technicians are assigned to the task.

The priority p_i of a task is used in the objective function of the problem, which is a weighted makespan of each priority type. More precisely we minimize the function $w_1 t_1 + w_2 t_2 + w_3 t_3 + w_4 t_4$, where t_i is the ending time of the last task of priority type $i \in \{1, 2, 3\}$ and t_4 is the ending time of the last task under consideration. The weights w_i are defined in Dutot et al. (2006) as $w_1 = 28$, $w_2 = 14$, $w_3 = 4$, $w_4 = 1$. One sees that it is important to serve priority 1 tasks as early as possible.

A budget C is available to outsource tasks. A set of tasks A can be outsourced if $\sum_{i \in A} c_i \le C$ and $\sigma_i \subseteq A, \forall i \in A$. Outsourcing tasks is not penalized in the objective.

Figure 1 depicts an example of a solution to a TTSP instance with 20 tasks, no precedence relations, seven technicians, and $C = 0$. The solution is displayed as a Gantt chart and spans three days. Each task is shown as a rectangle whose width corresponds to the duration of the task. The shade of the rectangle encodes the priority of the task (lighter rectangles correspond to more important tasks). Each row in the figure corresponds to a team; the technicians in the team are displayed on the left. The x-axis shows the time (each day is divided into 120 time units). The horizontal placement of each task shows its starting and ending times. The figure illustrates that although priority 1 tasks are the most important, it is not always best to serve the tasks in order of decreasing importance. In the example, priority 2 tasks are finished quickly as there only are a few of these tasks. The cost of the solution is $28 \cdot 300 + 14 \cdot 120 + 4 \cdot 360 + 1 \cdot 360 = 11{,}880$.

The problem under study belongs to the class of multi-skill project scheduling problems with hierarchical skill levels, investigated by Bellenguez and Néron (2005) and Bellenguez-Morineau and Néron (2007). With the exception of the presentations made at the 2007 ROADEF conference, the exact problem considered in this paper has not been previously addressed. In particular, we are the only ones to minimize a weighted prioritized objective. Our paper, like those of Caramia and Giordani (2009), Bellenguez and Néron (2005) and Bellenguez-Morineau and Néron (2007), does not consider routing costs between tasks. In these three references this is only natural because tasks are assigned to fixed resources like machines. All other references we are aware of Caseau and Koppstein (1992), Begur et al. (1997), Tsang and Voudouris (1997), Weigel and Cao (1999), Xu and Chiu (2001), Bertels and Fahle (2006), Eveborn et al. (2006) handle skill levels, as we do, as well as routing costs. No other study considers outsourcing. Like us, Eveborn et al. (2006), Bellenguez and Néron (2005) and Bellenguez-Morineau and Néron (2007) assign several technicians of different skills to the same task, while most other studies assign a single technician to a task. With the exception of Bellenguez and Néron (2005) and (Bellenguez-Morineau and Néron 2007) who have developed lower bounds and an exact branch-and-bound algorithm, all authors have proposed heuristics: classical heuristics (Begur et al. 1997; Eveborn et al. 2006), metaheuristics (Bertels and Fahle 2006; Caramia and Giordani 2009; Tsang and Voudouris 1997; Weigel and Cao 1999; Xu and Chiu 2001) or constraint programming combined with heuristics (Caseau and Koppstein 1992). We close this section with two references to Hurkens (2007) who won the Challenge, and to Estellon et al. (2008) who tied for second place. Hurkens applies a local search heuristic in which

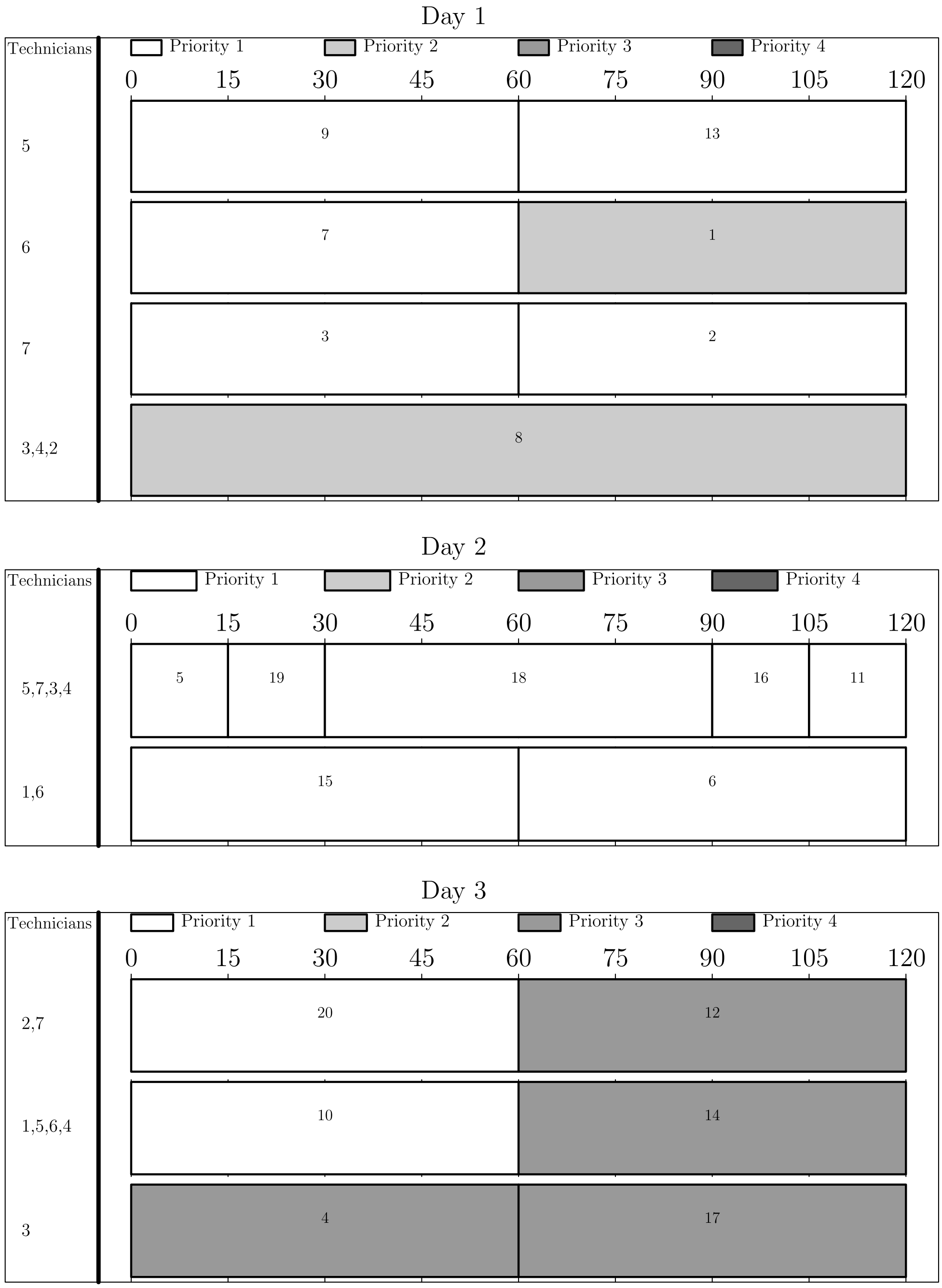

Fig. 1 Solution of an instance with 20 tasks and seven technicians

large neighborhoods are explored through the exact solution of integer linear programs. The Estellon, Gardi and Nouioua team has developed a local search heuristic that makes use of first improvement descents in which several neighborhoods are defined and randomly selected at each iteration.

The remainder of this paper is organized as follows. In Sect. 2 we provide a mathematical model for the problem. Sections 3 and 4 describe our construction heuristic and our adaptive large scale neighborhood search heuristic, respectively. Computational results follow in Sect. 5.

2 Mathematical model

This section presents a mathematical model to precisely define the problem. We introduce further notation. Let N_p denote the set of tasks with priority p, N^σ the set of tasks with successors, and K the index set of all days. There is no imposed upper bound on the number of days used to schedule all tasks, but by using an upper bound u on the objective function value, one can derive a limit on the number of days: since an optimal solution can use at most $\lceil u/120 \rceil$

days, $K = \{1, \ldots, \lceil u/120 \rceil\}$. In the following, M denotes a large number and setting $M = 120|K|$ is sufficient.

Our model uses binary variables $x_{jkr} \in \{0, 1\}$ equal to 1 if and only if technician j is assigned to team r on day k, and $y_{ikr} \in \{0, 1\}$ equal to 1 if and only if task i is assigned to team r on day k. We also use continuous variables $b_i \geq 0$ for the start time of task i and $e_p \geq 0$ for the ending time of the latest task of priority p for $p \in \{1, 2, 3\}$ and the latest task overall for $p = 4$. Binary variables $z_i \in \{0, 1\}$ take value 1 if and only if task i is outsourced. Finally, binary variables $u_{ii'} \in \{0, 1\}, i, i' \in N, i \neq i'$ take value 1 if and only if task i finishes before task i' starts. These are used to ensure that two tasks served by the same team on the same day do not overlap. Using this notation, the TTSP can be modeled as follows:

$$\text{minimize} \sum_{p=1}^{4} w_p e_p \tag{1}$$

subject to

$$Mz_i + e_p \geq b_i + d_i \quad \forall p \in \{1, 2, 3\}, \forall i \in N_p \tag{2}$$

$$Mz_i + e_4 \geq b_i + d_i \quad \forall i \in N \tag{3}$$

$$\sum c_i z_i \leq C \tag{4}$$

$$|\sigma_i| z_i \leq \sum_{i' \in \sigma_i} z_{i'} \quad \forall i \in N^{\sigma} \tag{5}$$

$$\sum_{r=1}^{m} x_{jkr} \leq 1 \quad \forall k \in K, \forall j \in \mathcal{T}_k \tag{6}$$

$$\sum_{r=1}^{m} x_{jkr} = 0 \quad \forall k \in K, \forall j \in \mathcal{T} \setminus \mathcal{T}_k \tag{7}$$

$$z_i + \sum_{k \in K} \sum_{r=1}^{m} y_{ikr} = 1 \quad \forall i \in N \tag{8}$$

$$s^i_{\alpha\beta} y_{ikr} \leq \sum_{j \in \mathcal{T}_k} v^j_{\alpha\beta} x_{jkr}$$
$$\forall i \in N, \forall k \in K, \forall r \in \{1, \ldots, m\},$$
$$\forall \alpha \in \{1, \ldots, p\}, \forall \beta \in \{1, \ldots, q\} \tag{9}$$

$$b_i + d_i \leq b_{i'} + Mz_{i'} \quad \forall i \in N^{\sigma}, \forall i' \in \sigma_i \tag{10}$$

$$\sum_{k \in K} \left(120(k-1) \sum_{r=1}^{m} y_{ikr} \right) \leq b_i \quad \forall i \in N \tag{11}$$

$$Mz_i + \sum_{k \in K} \left(120k \sum_{r=1}^{m} y_{ikr} \right) \geq b_i + d_i \quad \forall i \in N \tag{12}$$

$$b_i + d_i - (1 - u_{ii'})M \leq b_{i'} \quad \forall i, i' \in N, i \neq i' \tag{13}$$

$$y_{ikr} + y_{i'kr} - u_{ii'} - u_{i'i} \leq 1$$
$$\forall i, i' \in N, i \neq i', \forall k \in K, \forall r \in \{1, \ldots, m\} \tag{14}$$

$$x_{jkr} \in \{0, 1\} \quad j \in \mathcal{T}, \forall k \in K, \forall r \in \{1, \ldots, m\} \tag{15}$$

$$y_{ikr} \in \{0, 1\} \quad i \in N, \forall k \in K, \forall r \in \{1, \ldots, m\} \tag{16}$$

$$z_i \in \{0, 1\} \quad i \in N \tag{17}$$

$$u_{ii'} \in \{0, 1\} \quad i, i' \in N, i \neq i' \tag{18}$$

$$e_p \geq 0 \quad \forall p \in \{1, 2, 3, 4\} \tag{19}$$

$$b_i \geq 0 \quad i \in N. \tag{20}$$

Inequalities (2) and (3) define the ending times used in the objective, and inequality (4) enforces the outsourcing budget. Inequalities (5) ensure that if a task is outsourced then so are all its successors. Inequalities (6) ensure that a technician is used at most in one team per day, and inequalities (7) mean that unavailable technicians are not used. Inequalities (8) ensure that every task is either performed or outsourced, while inequalities (9) state that each task is performed by a team with the appropriate skills (if it is not outsourced), and inequalities (10) enforce the precedence constraints. Inequalities (11) and (12) set lower and upper bounds on the starting time of each task based on which day the task is performed. Inequalities (13) set the $u_{ii'}$ variables correctly relative to the starting times of task i and i', while (14) guarantee that task i and i' do not overlap if they are performed by the same team on the same day: if two distinct tasks i and i' are served by the same team r on the same day k, then $y_{ikr} + y_{i'kr} = 2$ and either $u_{ii'}$ or $u_{i'i}$ must be equal to 1 for the inequality to be satisfied. Constraints (15)–(20) define the domains of the decision variables.

3 Construction heuristic

This section describes a construction heuristic for the TTSP. This heuristic can reasonably quickly provide a feasible solution to the problem and is used as an important component in the metaheuristic described in Sect. 4. It plans one day at a time, using a two-phase approach. In the first phase teams are constructed and a single task is assigned to each of them, while in the second phase more tasks are assigned to the already constructed teams.

The precedence constraints are cumbersome to handle and we have therefore chosen a simple strategy to ensure that they are satisfied by the solution produced by the construction heuristic. The strategy prevents tasks having unplanned predecessors from being inserted by the construction heuristic. As soon as all predecessors of a task have been inserted, the task becomes available for the construction heuristic and it can then be inserted in a position that satisfies the precedence constraint.

3.1 Phase 1: constructing teams

The heuristic constructs teams by selecting an unserved task and constructing a team for it. We call such a task a *seed* *task.* Seed tasks are selected based on three criteria: *criticality*, *difficulty* and *similarity*. The first criterion measures how important it is to serve the task early, depending on the priority and duration of the task and of its successors. More precisely, we define the criticality α_i of a task i as

$$\alpha_i = w_{p_i} d_i + \sum_{j \in \sigma_i} w_{p_j} d_j.$$

Recall that p_i is the priority of task i and w_j is the weight used in the objective for priority j. One sees that high priority tasks and tasks with many successors are considered critical. The second criterion, difficulty, is a measure of how difficult it is to create a team for the task. It is based on the skill matrix of the task. We define the difficulty β_i of a task i as

$$\beta_i = \sum_{k=1}^{p} \sum_{l=1}^{q} s_{kl}^i.$$

This difficulty measure reacts to tasks that require highly skilled technicians. For example, if a task requires precisely one level 1 technician in a single domain its difficulty score is 1. If a task requires precisely one level 5 technician in a single domain its difficulty score is 5 because the task's skill matrix would have $s_{kl} = 1$ for $l = 1, \ldots, 5$ for the particular domain k. Such a task would seem much more difficult.

The last criterion, similarity, measures how similar a task is to the tasks already chosen as seed tasks. We want the seed tasks to be different from each other in order to be able to serve a variety of tasks on a given day and not just a small subset of the tasks. We define the similarity γ_{ij} between two tasks i and j as

$$\gamma_{ij} = \sum_{k=1}^{p} \sum_{l=1}^{q} \left| s_{kl}^i - s_{kl}^j \right|.$$

This similarity measure is defined in terms of the difference in the skill requirements of the two tasks. We also define the similarity γ_{iS} of a task i, relative to a set of already chosen seed tasks S, as

$$\gamma_{iS} = \sum_{j \in S} \gamma_{ij}.$$

The three measures are used to construct a score for each unassigned task i as follows:

$$f(i, S) = w_\alpha \alpha_i + w_\beta \beta_i + w_\gamma \gamma_{iS},$$

where S is the set of existing seed tasks and w_α, w_β and w_γ are parameters that determine the importance of each measure. The task with the highest score is chosen as a possible seed task. If it is possible to perform it with the technicians available on the current day, then this task is declared a seed task and the corresponding team is created (the process will be explained later). If this is impossible, then the algorithm proceeds with the next best task based on the $f(i, S)$ score until a task that can be served has been found or all tasks have been considered. Each time a team has been constructed, the number of unassigned technicians is checked. If fewer than $\theta|\mathcal{T}_k|$ technicians are unassigned then the construction of teams is stopped. Here $\theta \in [0, 1]$ is a *safety stock* parameter on day k, introduced to ensure that some technicians are left unassigned for the second phase of the heuristic where tasks are assigned to the existing teams. These remaining technicians can be used to adjust existing teams when a team lacks a few skills in order to serve a certain task.

When constructing teams the algorithm follows a greedy approach. Initially an empty team is created. Technicians are then added one at a time to the team until it is able to serve the task, or when it is determined that the task cannot be served by the available technicians. The algorithm always adds the technician who is able to *cover* most of the demanded skills not already covered by the other technicians in the team. To illustrate the concept of covering skill requirements we use the following example. Consider the skill requirement matrix used in Sect. 1:

$$A = \begin{pmatrix} 1 & 2 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 2 \end{pmatrix},$$

and three technicians with skill vectors $(2, 1, 1, 3)$, $(1, 3, 2, 3)$ and $(1, 1, 0, 0)$. We can express these skill vectors as the three matrices B, C and D:

$$B = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$C = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix},$$

$$D = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

The formula for calculating by how much a technician with skill matrix $Y = (y_{kl})$ covers a task with skill requirement matrix $X = (x_{kl})$ is

$$\sum_{k=1}^{p} \sum_{l=1}^{q} \min\{x_{kl}, y_{kl}\}.$$

In this example the covering scores for the three technicians are 5, 7 and 2, respectively. The greedy heuristic would consequently chose the second technician, after which the skill requirement matrix is updated since some skills have been covered. We construct a new skill requirement matrix $X' = (x'_{kl})$ from the prior skill requirement matrix X and the technician skill matrix Y by setting $x'_{kl} = \max\{0, x_{kl} - y_{kl}\}$, $k = 1, \ldots, p$, $l = 1, \ldots, q$. The updated skill requirement matrix A' in the example is

$$A' = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

and the covering scores by using matrix B and D are 4 and 1, respectively. The greedy algorithm would choose to add the first technician to the team and the resulting team could then perform the task.

If several technicians have the same covering score, a second criterion is then used to break ties. It measures the amount of skills *wasted* by assigning the technician to the task and team. If the technician's skill matrix is Y and the skill requirement matrix is X then the score for wasted skills is

$$\sum_{k=1}^{p} \sum_{l=1}^{q} \max\{0, y_{kl} - x_{kl}\}.$$

The skill requirement matrix A and the skill matrices B, C and D result in the waste scores 2, 2 and 0 respectively. Further tie breaking is done by selecting the technician with lowest index.

The first phase of the construction heuristic is outlined in Algorithm 1. The outer repeat loop adds teams to the day as long as there are enough technicians and unassigned tasks left. The inner repeat loop finds a suitable seed task (line 5). Recall that the arg max function returns the element that maximizes its expression. In line 7 a suitable team is created for the seed task. The function makeTeam(i, M') creates a team for task i using technicians from the set M' as described above. The function returns the set of technicians on the team. If it is impossible to create a team for the task then the empty set is returned. If a team could not be created for the seed task and there still are tasks left, then the inner loop is performed again (condition in line 9). Line 10 checks whether a team was created. If so, the team is added by calling the function addTeam(k, U) that adds the team consisting of the technicians in the set U to the current plan on day k. In line 11 we also add the chosen task to the set of seed tasks S.

Algorithm 1 Phase 1: Team construction

1: input: k (day), T (unassigned tasks);
2: $S = \emptyset$; M' = available technicians on day k;
3: **repeat**
4:   **repeat**
5:     $i' = \arg\max_{i \in T}\{f(i, S)\}$;
6:     $T = T \setminus \{i'\}$;
7:     $U = \text{makeTeam}(i', M')$;
8:     $M' = M' \setminus U$;
9:   **until** $U \neq \emptyset \vee T = \emptyset$;
10:   **if** $U \neq \emptyset$ **then**
11:     addTeam(k, U); assign i' to new team;
      $S = S \cup \{i'\}$;
12:   **end if**
13: **until** $|M'| \leq \theta m_k \vee T = \emptyset$;

3.2 Phase 2: assigning tasks

In the second phase of the construction of a day in the schedule, the algorithm assigns tasks to the teams constructed in phase 1. The core of the second phase is a score $g(i, r)$ that measures how promising it is to assign task i to team r. This score is composed of three weighted terms: (1) the criticality α_i of task i as defined in Sect. 1, (2) the number of technicians $h_1(i, r)$ that must be added to team r so that it can perform task i, and (3) the skills $h_2(i, r)$ wasted by assigning task i to team r after team r has been extended to be capable of serving i (if necessary). The term $h_1(i, r)$ is calculated by using the algorithm for constructing teams outlined in Sect. 1 and by using the skill requirement matrix that remains after covering the requirement matrix of task i with the technicians in team r. Of course, only the remaining free technicians can be used to extend the team. The term $h_2(i, r)$ is calculated for the extended team found when calculating $h_1(i, r)$ and only takes into account the skills wasted when serving task i, not the skills wasted while serving the other task assigned to team r. The formula for calculating $g(i, r)$ is

$$g(i, r) = w_\kappa h_1(i, r) + w_\lambda h_2(i, r) - w_\mu \alpha_i,$$

where w_κ, w_λ and w_μ are parameters. If task i cannot be served by team r, even after adding extra technicians, then we set $g(i, r) = \infty$. Obviously low scores are preferred.

Having defined $g(i, r)$ the rest of phase 2 is simple. We scan all teams and all unassigned tasks and choose the task and team for which $g(i, r)$ is lowest. The selected task i is inserted into the selected team r and technicians are added to team r if necessary. This process stops when no more tasks can be inserted into any team on the current day. When that happens the algorithm checks whether some technicians are still unassigned. If this is the case, then one of these technicians is chosen and a team consisting of only that technician is formed. The algorithm then attempts to insert tasks into this team using the $g(i, r)$ score. The team can be extended with more technicians if possible and necessary. We continue adding teams in this fashion until all technicians have been assigned to a team and no more tasks can be inserted. If there still are unassigned tasks left after finishing phase 2 we start on a new day and repeat phases 1 and 2.

3.3 Further details and improvements

This section describes details and features of the construction algorithm. Section 3.3.1 explains how the algorithm selects the tasks that should be outsourced and Sect. 3.3.2 discusses the parameters of the construction algorithm.

3.3.1 Outsourcing tasks

In the description of the construction algorithm we have not explained how the algorithm deals with the opportunity to outsource tasks. Since there is no cost in the objective function for outsourcing tasks, only a budget constraint, the algorithm should take advantage of this opportunity.

We select the tasks to outsource before the construction starts. We assign a score $\chi(p_i)d_i\nu_i/c_i$ to each task i, where $\chi(p_i)=(w_{p_i})^2$ (recall that w_{p_i} is the objective weight of the priority class of task i) and ν_i is a lower bound on the number of technicians needed to perform task i. This value is determined as $\nu_i = \max_{k\in\{1,\dots,p\},l\in\{1,\dots,q\}}\{s^i_{kl}\}$. The score expresses a preference to outsource tasks that take a long time, have an important priority and need many technicians but are cheap to outsource. Tasks are sorted according to a decreasing score and are outsourced by processing the sorted list as long as the budget allows it. When we encounter a task that cannot be outsourced because of insufficient budget we skip it and try the next one. We take care not to remove tasks that have non-outsourced successor tasks in order to avoid infeasible solutions.

3.3.2 Setting parameters

The construction heuristic is controlled by many parameters, which is usually undesirable. We can, however, use the parameters to our advantage to create a diversification effect. This is done by setting the parameters randomly within certain intervals. The parameters are w_α, w_β, w_γ and θ (parameters used in first phase of construction algorithm), w_κ, w_λ and w_μ (parameters used in the second phase of the construction algorithm). The default values of the parameters are $w_\alpha=0.1$, $w_\beta=1$, $w_\gamma=1$, $\theta=0.2$ $w_\kappa=5$, $w_\lambda=1$ and $w_\mu=0.025$. In the randomizing process, the values of the parameters are chosen randomly within the following intervals $w_\alpha\in[0;1]$, $w_\beta\in[0;10]$, $w_\gamma\in[0;10]$, $\theta\in[0;0.5]$ $w_\kappa\in[1;20]$, $w_\lambda\in[0.5;2]$ and $w_\mu\in[0;0.2]$. The parameter values and intervals were calibrated by performing a limited number of tests.

In Sect. 4 it will become clear how the different versions of the construction heuristic are used within the adaptive large neighborhood search algorithm.

4 Adaptive large neighborhood search

The metaheuristic used in this paper is based on the *large neighborhood search* (LNS) metaheuristic proposed by Shaw (1998). A closely related concept is that of *very large scale neighborhood* (VLSN) search introduced by Ahuja et al. (2002). Whereas the LNS is a heuristic framework, VLSN is the family of heuristics that search neighborhoods whose sizes grow exponentially as a function of the problem size, or neighborhoods that simply are too large to be searched explicitly in practice, according to Ahuja et al. (2002). Thus, LNS is an example of a VLSN heuristic. In LNS heuristics one does not explicitly define the neighborhood as in classical neighborhood search heuristics. Instead the neighborhood is defined implicitly by a *destroy* and a *repair* method. A destroy method destructs part of the current solution while a repair method rebuilds the destroyed solution. The destroy method typically contains an element of stochasticity so that different parts of the solution are destroyed at every invocation of the method. The neighborhood of a solution is then defined as the set of solutions that can be reached by first applying the destroy method and then the repair method. In the LNS described in this paper, a solution is destroyed by removing a subset of the tasks from the solution, and the solution is repaired by reinserting these tasks using the construction heuristic described in Sect. 3. Because the destroy method can destruct a rather large part of the solution the neighborhood contains a large amount of solutions and hence the name. A very similar heuristic framework has also been proposed under the name *ruin and recreate* heuristic by Schrimpf et al. (2000). LNS is also related to variable neighborhood search (VNS) which systematically changes neighborhoods during the search and typically moves from one solution to another without fully exploring the neighborhood (Mladenović and Hansen 1997).

The LNS heuristic has previously been applied with success to some scheduling problems. See for example Danna and Perron (2003), Palpant et al. (2004) and Laborie and Godard (2007). Examples of VLSN heuristics for scheduling problems, that do not fit into the LNS framework, are those of Meyers and Orlin (2007) and of Rios-Solis and Sourd (2008).

The *adaptive large neighborhood search* (ALNS) heuristic extends LNS by applying several destroy and repair methods. It was proposed by Ropke and Pisinger (2006a) and has been successful in solving many variants of the vehicle routing problem (see Pisinger and Ropke 2007; Ropke and Pisinger 2006b). A similar framework has been successful for a range of scheduling problems (see Laborie and Godard 2007). In the ALNS the performance of each destroy and repair method is recorded and the heuristic adapts to the current instance by favoring methods that have so far performed well.

Algorithm 2 Adaptive Large Neighborhood Search

1: input: a feasible solution x;
2: $x^* = x$; $\rho^d = (1, \dots, 1)$; $\rho^r = (1, \dots, 1)$;
3: **repeat**
4: choose a destroy method d and a repair method r by roulette-wheel selection using ρ^d and ρ^r;
5: $x' = r(d(x))$;
6: **if** accept(x', x) **then**
7: $x = x'$;
8: **end if**
9: **if** $f(x') < f(x^*)$ **then**
10: $x^* = x'$;
11: **end if**
12: update ρ^d and ρ^r;
13: **until** stop criterion is met;
14: **return** x^*;

Algorithm 2 shows the pseudo-code for the ALNS algorithm. Five variables are maintained by the algorithm. The variable x^* is the best solution observed during the search, x is the current solution and x' is a temporary solution that can be discarded or promoted to the status of current solution. The variables ρ^d and ρ^r are vectors with a component for each destroy and repair heuristic, respectively. A component is a weight that measures how well the corresponding removal or repair heuristic has performed during the search. In line 2 the global best solution is initialized and the vectors containing the scores of the destroy and repair methods are initialized (initially all methods are given equal weight). In line 4 the heuristic selects a destroy and repair method based on the current score vector. This selection process is described in detail in Sect. 4.3. In line 5 the heuristic first applies the chosen destroy heuristic and then the chosen repair heuristic to obtain a new solution x'. More specifically, $d(x)$ returns a copy of x that is partly destroyed, that is, some tasks have been removed from the plan. Applying $r(\cdot)$ to the partly destroyed solution repairs it, that is, it returns a feasible solution built from the destroyed one. The destroy and repair methods are described in Sects. 4.1 and 4.2, respectively. In line 6 the new solution is evaluated, and the heuristic determines whether this solution should become the new current solution (line 7) or whether it should be rejected. In our implementation we use a simulated annealing criterion as in Ropke and Pisinger (2006a). The new solution x' is always accepted if $f(x') \le f(x)$, and accepted with probability $e^{-(f(x')-f(x))/T}$ if $f(x') \ge f(x)$. Here $f(x)$ denotes the objective of solution x and $T > 0$ is the current *temperature*. The temperature is initialized at $T_0 > 0$ and is decreased at each iteration by performing the update $T_{new} = \zeta T_{old}$, where $0 < \zeta < 1$ is a parameter. Because of the acceptance criterion, the metaheuristic proposed in this paper can also be viewed as a standard simulated annealing algorithm with a complex neighborhood definition. Details about simulated annealing algorithms can be found in Kirkpatrick et al. (1983). Line 9 checks whether the new solution is better than the best known solution, which is then updated in line 10. In line 12 the score-vectors are updated based on the performance of the destroy and repair heuristics chosen at the current iteration; this is explained in detail in Sect. 4.3. In line 13 the termination condition is checked. The heuristic stops if a certain number of iterations have been performed or if the allocated running time is exhausted (see Sect. 4.4 for details). In line 14 the best known solution is returned.

4.1 Destroy heuristics

This section describes the five destroy heuristics. A solution is destroyed by removing a set S of tasks, while ensuring that $i \in S \Rightarrow \sigma_i \subseteq S$, which makes it easier to repair the solution. To illustrate the difficulties we would encounter if we did not remove successors, consider an example. If a task i is removed but a successor $i' \in \sigma_i$ is not, then it becomes more difficult to reinsert i. Either we would have to impose a deadline on when i can be served, or we would have to relocate i' if i were inserted too late. Removing i' together with i removes these difficulties. In practice we check the successors of i every time a task i is added to the set S of tasks to be removed. As stated above this is done for all removal heuristics and we do not describe this common step in the detailed descriptions of the heuristics below.

When a number of tasks have been removed it is often possible to *trim* the teams, that is to remove redundant technicians. An extreme example is when all tasks assigned to a team are removed, in which case all technicians can be removed from the team and the team itself can be discarded. The heuristic always performs such a trimming procedure when a task removal has been performed. This ensures that we are able to move to solutions that use different team assignments compared with the initial solution. The trim heuristic is extremely simple: for each technician in the team it tests whether the team would still be able to serve its tasks after removing that technician. If more than one technician can be removed, then one of these is randomly selected for removal. This process continues as long as technicians can be removed.

After removing tasks we also *compress* the solution. This means that we try to serve tasks earlier whenever possible. Our compress procedure only explores a limited number of possibilities. It does not allow tasks to change teams or to be served on earlier days, and it only examines whether the task can be served earlier on the same team because a task removed from the team has "left a hole". There are three reasons for this choice: (1) a compression mechanism that would investigate further options would be too time consuming, (2) it might not be a good idea to compress the solution too much as this would leave no room for the repair heuristic to insert the removed tasks (most of them would have to be served at the end of the schedule), and (3) it would be complicated to implement such a mechanism and simplicity is important in a heuristic.

Before invoking the removal heuristic the number of tasks Δ to remove is determined. This number is chosen randomly in the interval $[\max\{n^-, r^- n\}, \min\{n^+, r^+ n\}]$, where $n^-$and n^+ are the absolute minimum and maximum number of tasks we ever remove, and r^- and r^+ indicate the minimum and maximum proportions of the total number of tasks. For a sensible parameter setting, n^- and n^+ are only useful for very small and very large instances. Successors of removed tasks are included when calculating how many tasks have been removed.

4.1.1 Random destroy

Random destroy is the simplest destroy method. It selects Δ tasks at random and removes these. It works well as a diversification method.

4.1.2 Related destroy

Related destroy is similar to the removal method of the same name proposed by Shaw (1998) for the vehicle routing problem. For every pair of tasks (i, i') the coefficient $\gamma_{ii'}$ defined in Sect. 1 measures the similarity of i and i' in terms of their skill requirements. The algorithm works as follows. Initially, a task i is selected at random and the set S of tasks to remove is initialized as $S = \{i\}$. The heuristic then adds tasks to S as long as $|S| < \Delta$ in the following way: a random task i' from S is selected, the tasks in $N \setminus S$ are sorted in a list L according to their decreasing similarity with i', a random number $r \in [0, 1]$ is drawn, and the element at position $r^v(|L| - 1)$ in L is added to S. We index the positions in L from 0 and $v \geq 1$ is a parameter that controls how deterministic the removal method is. The larger v is, the more likely it is that the heuristic selects tasks similar to those already in S.

The goal of this heuristic is to remove similar tasks from the current schedule, hoping that the repair method will be able to group them together in a few teams, and hopefully freeing some technicians in the process. The related destroy method also increases the likelihood of filling the holes left by removed tasks by a similar, but different task during the repair process.

4.1.3 Last-random and last-related destroy

As observed earlier, only the last tasks of each priority type contribute to the objective function. The two destroy heuristics described in this section attempt to take advantage of this feature by removing some of the last tasks of each priority type. If the repair method is capable of moving at least some of these tasks to earlier positions without delaying other tasks, then one can hope for a reduction of the objective function value.

The heuristics first remove $\tau\Delta$ tasks that are the latest of one of the priority types ($0 < \tau \leq 1$), and then $(1 - \tau)\Delta$ tasks by selecting either random tasks (*last-random destroy*) or tasks that are related to the already removed tasks (*last-related destroy*) by using the principles described in Sect. 4.1.2. The idea behind not only removing the latest tasks but also some random or related tasks is to make room elsewhere in the plan for the latest tasks.

The removal of the latest tasks works as follows. First the number of planned tasks n_p of each priority type p is calculated. At each step the heuristic removes the latest task of priority p with probability $n_p/(n_1 + n_2 + n_3 + n_4)$, and n_p is updated. If several tasks tie to be the latest task of a certain priority, then the task with lowest index is removed first. The heuristic keeps removing the latest tasks until the number of removed tasks reaches $\tau\Delta$.

4.1.4 Whole team destroy

The *whole team destroy* heuristic attempts to remove complete teams from the solution. This is done to make it easier for the repair heuristic to regroup the technicians into new teams. The heuristic always attempts to remove two teams from the same day in order to create more opportunities for new teams. The heuristic repeats the following operation until Δ tasks have been removed. First a day is selected at random. If the day contains less than two teams, then the heuristic proceeds to the next day until a day with at least two teams is found. If no such day is found the heuristic terminates. When a day with two teams is identified the heuristic selects two of those teams at random and removes all tasks assigned to them.

4.2 Repair heuristics

The ALNS heuristic uses two variants of the construction heuristic described in Sect. 3 as repair heuristics: a version that uses the fixed parameter setting described in Sect. 3.3.2 and a version that uses randomized parameters also described in Sect. 3.3.2

4.3 Selection of removal and repair heuristics

This section describes how removal and repair heuristics are selected. We use the approach suggested by Ropke and Pisinger (2006a) with a slight simplification. The vectors ρ^d and ρ^r, introduced in the overview of ALNS heuristic, control the probabilities of choosing destroy and repair heuristics. These vectors contain components corresponding to the weights of the heuristic: ρ^d contains five components and ρ^r

contains four components. For example, ρ_j^d is the weight of the jth destroy heuristic. We denote the number of destroy and repair heuristics by η^d and η^r, respectively. The vectors determine the probability of choosing a given heuristic in line 4 by using the weight of the heuristic in a *roulette-wheel selection mechanism*. The probability ϕ_j^d of choosing the j^{th} destroy heuristic is

$$\phi_j^d = \frac{\rho_j^d}{\sum_{k=1}^{\eta^d} \rho_k^d},$$

and the probabilities for choosing the repair heuristics are determined in the same way.

The weights are adjusted automatically, based on the recorded performance of the heuristics. The idea is that the ALNS heuristic should *adapt* to the instance at hand since the best heuristic is not the same for all instances. Also, we may expect that one set of probabilities works well at the beginning of the search when it can be easy to find improvements, while another set of probabilities is necessary towards the end of the search when improvements are hard to find.

When an iteration of the ALNS heuristic is completed a score ψ for the destroy and repair heuristic used in the iteration is computed as

$$\psi = \begin{cases} \omega_1 & \text{if the new solution is a new global best,} \\ \omega_2 & \text{if the new solution is better than the current} \\ & \text{one,} \\ \omega_3 & \text{if the new solution is accepted,} \\ \omega_4 & \text{if the new solution is rejected,} \end{cases} \tag{21}$$

where $\omega_1, \omega_2, \omega_3$ and ω_4 are parameters satisfying $\omega_1 \geq \omega_2 \geq \omega_3 \geq \omega_4$. In our tests we have used $\omega_1 = 100, \omega_2 = 40, \omega_3 = 10, \omega_4 = 1$. The first matching option in (21) is always preferred (for example if the new solution is better than the current one, it would always be accepted, but we would still set $\psi = \omega_2$). Let a and b be the indices of the selected destroy and repair heuristics, respectively. The components corresponding to the selected destroy and repair heuristics in the ρ^d and ρ^r vectors are updated using equations (22) in line 12 of Algorithm 2:

$$\begin{aligned} \rho_a^d &= \lambda \rho_a^d + (1-\lambda)\psi, \\ \rho_b^r &= \lambda \rho_b^r + (1-\lambda)\psi, \end{aligned} \tag{22}$$

where $\lambda \in [0; 1]$ is the *decay* parameter that controls how sensitive the weights are to changes in the performance of the destroy and repair heuristics. A value of 1 means that the weights remain unchanged, while a value of 0 implies that historic performance has no impact: only the last attempt at using the heuristic counts. Obviously a value between the two extremes is preferable; in our tests we used $\lambda = 0.99$. Note that the weights that are not used at the current iteration remain unchanged. This dynamic weight adjustment mechanism was proposed by Ropke and Pisinger (2006a). The approach implemented in that reference is slightly more complicated than ours since it maintains a vector of heuristic weights and a vector of temporary scores with a component per heuristic. Our procedure only needs the vector of weights. However, we have no reason to believe that either approach dominates the other when it comes to selecting good values for the weights.

4.4 Variants and improvements

This section describes further refinements of the ALNS heuristic. All refinements aim at improving the solution quality obtained by the heuristic. In Sect. 4.4.1 an artificial objective is defined. In Sect. 4.4.2 a strategy for resetting the temperature in the simulated annealing heuristic is described, and Sect. 4.4.3 shows how the ALNS heuristic is divided into several phases.

4.4.1 Artificial objective function

One major difficulty of the TTSP is the flatness of its objective function. Because only the last tasks of each priority type count in the objective, many solutions share the same objective value, even though they can be very different. In this sense the objective value is not very useful for guiding a local search because it does not reveal whether a solution is easy or difficult to improve. As a remedy we introduce an artificial objective function which considers the ξ last tasks of the priority types 1, 2 and 3 and the ξ last tasks overall, where ξ is a parameter. Let t_1^ξ, t_2^ξ and t_3^ξ be average end time of the ξ last tasks of priority type 1, 2 and 3 respectively, and let t_4^ξ be the average end time of the ξ last tasks overall. The artificial objective function is then defined as

$$\sum_{j=1}^{4} w_j t_h + 0.1 \sum_{j=1}^{4} w_j t_h^\xi.$$

The first term is the original objective, while the second term takes more tasks into account. It is scaled in such a way that it does not contribute too much to the artificial objective.

The artificial objective is used when deciding whether to accept or reject a solution in line 6 of Algorithm 2. If a new solution is better than the global best solution according to the original objective, then the new solution is accepted no matter what the artificial objective indicates.

4.4.2 Restarts

The rules of the ROADEF challenge enforced a strict time limit on the run time of the heuristic. Each instance, no matter its size, should be solved within 20 minutes on a particular PC. The simulated annealing metaheuristic is set to perform short runs of 25,000 iterations (we chose this number of iterations as it was shown to be sufficient in the experiments performed in Ropke and Pisinger 2006a and in Pisinger and Ropke 2007). For small and medium size instances performing 25,000 iterations takes only a fraction of the allocated time and, in order to utilize the entire allowed time, the heuristic is restarted from the best found solution. We found that it was beneficial to use a reduced initial temperature T_0^r in the first run and an increased initial temperature T_0 in the following runs. There are two reasons for this. First, for several of the large instances it is not possible to perform 25,000 iterations within the time limit. A lower initial temperature is beneficial for these instances since a more intensifying search is performed right from the start. Second, the initial solution constructed by the heuristic of Sect. 3 is sometimes of very high quality and a high initial temperature will tend to destroy this solution without being able to reconstruct it. In this case it is preferable to intensify the search instead of diversifying it.

4.4.3 Prioritized strategy

Priority 1 tasks are more costly (in the objective function) than priority 2 tasks which, in turn, are more costly than priority 3 tasks, and so on. It therefore seems natural to focus on the tasks in the order of their priority, creating solutions that first serve priority 1 tasks, then priority 2 tasks, and so on. However, such a strategy is not always the best option. Figure 1 shows the best known solution to a small TTSP instance in which priority 2 tasks are finished before priority 1 tasks. It often pays to finish priority 2 or 3 tasks early if there only are a few of them compared to priority 1 tasks. In this case, serving the priority 2 tasks as early as possible instead of waiting until most or all priority 1 tasks have been served does not significantly affect the latest ending time of priority 1 tasks, but it can significantly decrease the ending time of priority 2 tasks.

This observation leads to an algorithm that tries different permutations of the insertion priorities. The algorithm could in principle examine the 24 permutations of the numbers 1, 2, 3 and 4, but we always insert priority 4 tasks as the last set of tasks and we limit the search to the six permutations of the numbers 1, 2 and 3. Little is lost through this simplification since priority 4 tasks seldom contribute significantly to the objective function. For each permutation $(\varphi_1, \varphi_2, \varphi_3)$ of $\{1, 2, 3\}$ the algorithm first constructs a solution that only contains tasks of priority φ_1 and their predecessors. This is done by applying the algorithm described in Sect. 3. After a solution has been constructed a short LNS phase is applied to obtain a better estimate of the solution quality that can be obtained with the current permutation. Only improving solutions are accepted and only 15 iterations are performed. The best solution from this LNS phase is used as starting point for the next phase in which all tasks from the previous phase are locked (they are not allowed to be moved from their current position), and tasks of priority φ_2 are inserted in the same way. The process continues until all tasks have been inserted and the next permutation is processed. The output of this algorithm is the permutation that yielded the best solution. This permutation is used as the input of the main part of the ALNS algorithm. Note that the permutation approach leads to a change in the outsourcing algorithm described in Sect. 3.3.1. The weights $\chi(p_i)$ are set according to the position of priority p_i in the permutation: for example, if the permutation is (3, 1, 2) then priority 3 tasks get a weight of 28^2, priority 1 tasks get a weight of 14^2 and priority 2 tasks get a weight of 4^2. This change was implemented to encourage the outsourcing of the tasks that are to be served first.

We now describe how the priority permutation is used in the main part of the ALNS algorithm. Let $(\varphi_1, \varphi_2, \varphi_3)$ be the best permutation found. Define four sets of tasks: B_1 is the set of all priority φ_1 tasks and their predecessors, B_2 is the set of all priority φ_2 tasks and their predecessors that are not in B_1, B_3 is the set of all priority φ_3 tasks and their predecessors that are not in $B_1 \cup B_2$, and $B_4 = N \setminus (B_1 \cup B_2 \cup B_3)$. The available run time for the heuristic is then split into seven stages. In stage 1 we only consider tasks from B_1 and optimize their placement using the ALNS heuristic. In stage 2 we start from the best solution identified in stage 1 and consider tasks from $B_1 \cup B_2$. Tasks from B_1 are locked: they cannot be moved from their current position. The ALNS heuristic will try to find good positions for the tasks in B_2 while respecting the placement of the tasks in B_1. In stage 3 we consider tasks from $B_1 \cup B_2$, and all tasks can be moved. These stages are summarized in Table 1. The time allocated for each stage is proportional to the number of new tasks considered (e.g. in stages 4 and 5 the ratio $|B_3|/|N|$ determines the allocated time). The temperature in the simulated annealing algorithm is reset to T_0^r every time a new stage begins.

At the start of stages 1, 2, 4 and 6 a new set of tasks have to be inserted. This is of course done with the construction heuristic from Sect. 3. We try the construction heuristic several times before applying the ALNS heuristic. The first time

Table 1 Stages in the prioritized ALNS strategy

Stage	Tasks considered
1	B_1
2	$B_1 \cup B_2$, tasks in B_1 are locked
3	$B_1 \cup B_2$
4	$B_1 \cup B_2 \cup B_3$, tasks in $B_1 \cup B_2$ are locked
5	$B_1 \cup B_2 \cup B_3$
6	N, tasks in $B_1 \cup B_2 \cup B_3$ are locked
7	N

the construction heuristic is attempted the default parameters are selected, while in the subsequent attempts the randomized parameters are used to create new solutions. The best solution found during these attempts serves as a starting point for the ALNS heuristic. We execute the construction algorithm several times since we have noticed that the construction algorithm is occasionally able to find good solutions that are hard to obtain with the ALNS heuristic. This staged ALNS heuristic involves some extra bookkeeping in the removal heuristics as we have to ensure that locked tasks are not removed, but this appears to be worthwhile.

It should be noted that the idea of trying permutations of the priorities was not part of our original solution method developed for the ROADEF challenge. The strategy was implemented as several of the competing teams reported good results with such a strategy.

5 Computational results

The algorithm was implemented in C++ and run on an AMD Opteron 250 computer (2.4 GHz) running Linux. The computer used in the competition was an AMD Athlon 64 3000+ (1.8 GHz) running Linux. The allotted time per run on this computer was 20 minutes. Running the heuristic on both computers revealed that 20 minutes on the competition computer roughly corresponded to 12 minutes on our computer. Consequently we have used a time limit of 12 minutes in the experiments performed in this paper.

5.1 Data sets

The organizers of the 2007 ROADEF challenge provided 30 test instances. These are grouped into three sets, A, B and X, of ten instances. The A instances were made available at the start of the competition and involve up to 100 tasks and 20 technicians. The B instances were revealed during the competition and contain larger instances with up to 800 tasks and 150 technicians. The X instances were kept secret until the evaluation phase and the heuristics were judged by their performance on these instances. Table 2 shows the characteristics of each instance. We report the number n of tasks, the number m of technicians, the number q of skill domains and the number p of skill levels.

5.2 Mathematical model

The motivation for including the mathematical model in Sect. 2 was mainly to give a precise definition of the problem. In this section we provide the results of a short test that indicates what problem sizes that can be solved with a commercial MIP solver when using the model. We used CPLEX 11 and the OPL 6.0 modeling language for the experiments which were performed on an Intel Xeon E5430 (2.66 GHz) processor running Windows Vista (64 bit). The CPLEX solver was single threaded and the standard parameter setting was used. Table 3 shows the results obtained on the four smallest instances, all from data set A. The table shows the best upper bound (UB) and lower bound (LB) obtained as well as the gap between the two values, calculated as (UB−LB)/UB. Each instance was allocated 24 hours of computation time. The first two instances were solved in less than one second while CPLEX was unable to prove optimality for the two last instances within the time limit. In order to obtain the results of Table 3 a few improvements had to be made to the model. As none of the four instances contains tasks of priority level 4, the upper bound on the number of days can be set to $\lceil u/(4 \times 120)\rceil$ where u is the best upper bound found by the heuristic described in this paper, this reduces the number of variables. Also, all variables y_{ikr}, for which $w_{p_i}(120(k-1)+d_i) > u$, can be eliminated as they cannot belong to an optimal solution.

Table 2 Characteristics of the instances in the three data sets

Instance	Dataset A				Dataset B				Dataset X			
	n	m	q	p	n	m	q	p	n	m	q	p
1	5	5	3	2	200	20	4	4	600	60	15	4
2	5	5	3	2	300	30	5	3	800	100	6	6
3	20	7	3	2	400	40	4	4	300	50	20	3
4	20	7	4	3	400	30	40	3	800	70	15	7
5	50	10	3	2	500	50	7	4	600	60	15	4
6	50	10	5	4	500	30	8	3	200	20	6	6
7	100	20	5	4	500	100	10	5	300	50	20	3
8	100	20	5	4	800	150	10	4	100	30	15	7
9	100	20	5	4	120	60	5	5	500	50	15	4
10	100	15	5	4	120	40	5	5	500	40	15	4

Table 3 Results from solving model (1)–(20) using CPLEX 11

Name	UB	LB	Gap (%)
A1	2340	2340.0	0
A2	4755	4755.0	0
A3	11880	9411.0	20.8
A4	13452	11428.8	15.0

Table 4 Results of the parameter tests

Configuration	Gap (%)
Permutation greedy	17.75
Normal	7.41
Remove half	7.15
Remove double	7.80
No artificial objective	7.99
Only random destroy	8.12
Only related destroy	8.64
Only fixed repair	8.69
No SA	9.20
Basic+	11.20
No stages	15.35
Basic−	16.40

5.3 Parameter setting

The ALNS heuristic and its various procedures are controlled by several parameters whose values were determined in a rather *ad hoc* manner. The parameters controlling the simulated annealing component are set as follows: $\zeta = 0.99965$, $T_0 = 500$ and $T_0^r = 50$. The parameters controlling the number of tasks to remove in each iteration have the following values: $r^+ = 0.4$, $r^- = 0.1$, $n^+ = 80$, $n^- = 15$. The removal heuristics define two parameters $\upsilon = 4$ (related destroy) and $\tau = 0.2$ (last-random and last-related destroy). The number of tasks to include in the artificial objective function is set to $\xi = 8$.

In order to asses the impact of the major features of the proposed heuristic ten different configurations of the heuristic were tested. In most of the configurations only one component of the heuristic was changed (as described below). The components not mentioned stay as described earlier.

- **No artificial objective:** The artificial objective function described in Sect. 4.4.1 was disabled and the original objective function was used instead.
- **No SA:** The simulated annealing acceptance criterion was disabled and only improving solutions were accepted (see Sect. 4).
- **Only random destroy:** The only destroy heuristic that was used was *random destroy* (see Sect. 4.1.1).
- **Only related destroy:** The only destroy heuristic that was used was *related destroy* (see Sect. 4.1.2).
- **Only fixed repair:** The only repair heuristic that was used was the insertion heuristic with fixed parameters (see Sect. 4.2).
- **No stages:** In this configuration the seven-stage process described in Sect. 4.4.3 and Table 1 was disabled and instead the ALNS heuristic only performed one stage where all tasks could be moved around freely. The initial solution was still constructed using the priority permutation approach described in the first two paragraphs of Sect. 4.4.3.
- **Remove half:** This configuration removes half the number of tasks per iteration. More precisely we perform the following change of parameters: $r^+ = 0.2$, $r^- = 0.05$, $n^+ = 40$, $n^- = 7$.
- **Remove double:** This configuration removes twice the number of tasks per iteration. More precisely we perform the following change of parameters $r^+ = 0.8$, $r^- = 0.2$, $n^+ = 160$, $n^- = 30$.
- **Basic+ and Basic−** Several features are disabled in these two configurations to represent the results that can be obtained with a simpler heuristic. In Basic+ the artificial objective and the simulated annealing acceptance criterion are disabled, the only destroy method is the random one and the only repair method is the insertion heuristic with fixed parameters. The Basic− configuration is similar to Basic+ except that the seven stages of the ALNS also are disabled as in configuration *No stages*.

The ten configurations of the heuristic were applied to the B data set. For each configuration and each instance 10 test runs were performed, each with a time limit of 12 minutes. The results are shown in Table 4. For each configuration we report the average gap over all ten instances. The gap for each instance is defined as $(z' - \bar{z})/\bar{z}$, where z' is the average objective (over the ten runs) and $\bar{z}$ is the best know solution to the instance. The table also contains entries showing the gap obtained with the standard configuration (*Normal*) and the construction heuristic that generates the initial solution for the ALNS heuristic (*Permutation greedy*). The table shows that the heuristic is relatively stable with respect to the number of tasks to remove and that it actually seems to be better to remove fewer tasks than what we do in the standard configuration. The results also show that solution quality deteriorates when features are turned off and that some features are more important than others. The seven-stage approach is especially important and without it the ALNS heuristic is not able to improve the initial solution by much. The results obtained by the simpler heuristics Basic− and Basic+ are not impressive and justify the more complicated heuristic proposed in this paper. If a simpler heuristic is sought then the best option seems to be to only include a single destroy method. The table shows that rather good performance is

Table 5 Results for greedy heuristics

Instance	Best	Greedy			Permutation greedy			
		Time (s)	Objective	Gap (%)	Time (s)	Objective	Gap (%)	
A	1	2340	0.0	3690	57.7	0.0	2340	0.0
A	2	4755	0.0	4755	0.0	0.0	4755	0.0
A	3	11880	0.0	16950	42.7	0.0	13710	15.4
A	4	13452	0.0	17520	30.2	0.0	13620	1.2
A	5	28845	0.0	42495	47.3	0.0	31215	8.2
A	6	18795	0.0	26775	42.5	0.0	20055	6.7
A	7	30540	0.0	36630	19.9	0.1	32520	6.5
A	8	16920	0.0	23280	37.6	0.1	18540	9.6
A	9	27348	0.0	40290	47.3	0.1	29664	8.5
A	10	38296	0.0	50640	32.2	0.1	40440	5.6
B	1	34395	0.0	93120	170.7	0.1	40620	18.1
B	2	15870	0.0	37425	135.8	0.2	18585	17.1
B	3	16020	0.1	47250	194.9	0.2	18360	14.6
B	4	23775	0.2	63705	167.9	1.2	30930	30.1
B	5	89700	0.4	148620	65.7	4.9	122640	36.7
B	6	26955	0.1	47850	77.5	0.4	31155	15.6
B	7	33060	0.3	43680	32.1	1.6	35820	8.3
B	8	33030	0.7	62160	88.2	4.2	40200	21.7
B	9	28200	0.0	34620	22.8	0.3	29760	5.5
B	10	34680	0.0	50820	46.5	0.2	38040	9.7
X	1	151140	1.8	272055	80.0	36.7	172695	14.3
X	2	7260	1.0	11700	61.2	3.3	9660	33.1
X	3	50040	0.1	67920	35.7	1.0	52800	5.5
X	4	65400	0.8	95520	46.1	4.1	68660	5.0
X	5	147000	2.1	235080	59.9	42.6	168330	14.5
X	6	9480	0.0	19350	104.1	0.2	12540	32.3
X	7	33240	0.1	51120	53.8	1.0	48960	47.3
X	8	23640	0.0	35020	48.1	0.4	28020	18.5
X	9	134760	1.4	213120	58.1	24.5	159960	18.7
X	10	137040	0.7	213240	55.6	12.5	158040	15.3
		1287856	0.3	2106400	65.4	4.7	1492634	14.8

obtained with only the very simple random destroy method compared to using all five destroy methods defined in this paper.

5.4 Experiments and results

Table 5 shows results from two variants of the greedy heuristic. The standard version described in Sect. 3 and the variant that attempts to insert tasks in different orders based on permutations of the three first priorities and performs a few LNS iterations (described in the first two paragraphs of Sect. 4.4.3). The table reports the time spent by each heuristic (columns 3 and 6), the solution obtained (columns 4 and 7) and the gap between the solution obtained and the best known solution (columns 5 and 8). The best known solution is reported in column 3. It is the best among all the solutions found in our experiments and all solutions obtained in the competition by the challengers. It is clear that the permutation variant by far produces the best results, but it does so by consuming much more time than the standard version. The standard version of the construction algorithm is relatively quick although an even faster heuristic would be preferred as it is an important component of the ALNS heuristic.

In Table 6 the ALNS heuristic is compared with the best two entries in the competition. The first two columns indi-

Table 6 Comparison of ROADEF challenge top three heuristics

Instance		Old best	New best	Hurkens		EsGaNo		ALNS	
				Obj.	Gap (%)	Obj.	Gap (%)	Obj.	Gap (%)
A	1	2340	2340	2340	0.0	2340	0.0	2340	0.0
A	2	4755	4755	5580	17.4	4755	0.0	4755	0.0
A	3	11880	11880	12600	6.1	11880	0.0	11880	0.0
A	4	13452	13452	13620	1.2	14040	4.4	13452	0.0
A	5	28845	28845	30150	4.5	29700	3.0	29355	1.8
A	6	18795	18795	20280	7.9	18795	0.0	18795	0.0
A	7	30540	30540	32520	6.5	30540	0.0	30540	0.0
A	8	16920	16920	18960	12.1	20100	18.8	17700	4.6
A	9	27692	27348	29328	7.2	28020	2.5	27692	1.3
A	10	38296	38296	40650	6.1	38296	0.0	38636	0.9
B	1	34395	34395	34710	0.9	34395	0.0	37200	8.2
B	2	15870	15870	17970	13.2	15870	0.0	17070	7.6
B	3	16020	16020	18060	12.7	16020	0.0	18015	12.5
B	4	25305	23775	26115	9.8	25305	6.4	23775	0.0
B	5	89700	89700	94200	5.0	89700	0.0	117540	31.0
B	6	27615	26955	30450	13.0	27615	2.4	27390	1.6
B	7	33300	33060	33300	0.7	38220	15.6	33900	2.5
B	8	33030	33030	35490	7.4	37440	13.4	33240	0.6
B	9	28200	28200	28200	0.0	32700	16.0	29760	5.5
B	10	34680	34680	34680	0.0	41280	19.0	35640	2.8
X	1	151140	151140	151140	0.0	188595	24.8	159300	5.4
X	2	7260	7260	9120	25.6	8370	15.3	8280	14.0
X	3	50040	50040	50400	0.7	50100	0.1	50400	0.7
X	4	65400	65400	65400	0.0	68120	4.2	66780	2.1
X	5	147000	147000	147000	0.0	183720	25.0	157800	7.3
X	6	9480	9480	10320	8.9	10440	10.1	9900	4.4
X	7	33240	33240	33240	0.0	37200	11.9	47760	43.7
X	8	23640	23640	23640	0.0	25480	7.8	24060	1.8
X	9	134760	134760	134760	0.0	159660	18.5	152400	13.1
X	10	137040	137040	137040	0.0	152040	10.9	140520	2.5
		1290630	1287856	1321263	5.6	1440736	7.7	1385875	5.9

cate the data set and the instance number. The third column presents the best known solution value of all the heuristics presented in the competition, the next column shows the best solution value when new solutions found by the ALNS heuristic is taken into account. The last six columns show the results from the top three heuristics. For each heuristic we report the objective of the solution found and the gap relative to the (new) best known solution. The columns with heading *Hurkens* report the results from the heuristic by Hurkens (2007) (winner), the columns *EsGaNo* report the results from the heuristic by the team of Estellon et al. (2008) (tied second place), and the last columns report the results from our ALNS heuristic. The last row is the sums of the objective and averages the gap columns. The results for the two competing heuristics are those obtained during the competition when the heuristic was run for 20 minutes on the computer provided by the organizers. The results for the ALNS were obtained by running the heuristic for 12 minutes on our test computer.

It is clear that when all instances are considered, the ALNS heuristic is very close to the winning heuristic in terms of relative gap. Table 7 reports the average gap of the three heuristics on the three test sets A, B and X. Here three different results for the ALNS heuristic are reported: (1) results for one run (as in Table 6), (2) average results over ten runs (3) best results over ten runs. The fairest comparison is

Table 7 Summary of ROADEF challenge top three heuristics

	Hurkens (%)	EsGaNo (%)	ALNS one run (%)	ALNS avg. (%)	ALNS best (%)
A	6.9	2.9	0.9	1.1	0.5
B	6.3	7.3	7.2	7.4	5.0
X	3.5	12.9	9.5	10.0	8.0
	5.6	7.7	5.9	6.2	4.5

between the ALNS one-run results and the results from the two competing heuristics. The results obtained as the best of ten runs are also interesting as they indicate that an improved solution quality is within reach of the ALNS heuristic if the time limit is increased. However, a direct comparison between these results and the competing heuristics would be unfair. A final remark is that the performance of Hurkens's heuristic on the X data set is very impressive.

6 Conclusion

We have modeled and solved a complex problem arising in the scheduling of technicians and tasks for a telecommunications company. This problem was the topic of the 2007 ROADEF Challenge. Because the model cannot be solved optimally for large instances, we have developed a heuristic consisting of a construction step, followed by an improvement step. The construction step quickly generates a feasible solution by successively defining teams and assigning tasks. The improvement step is an adaptive large neighborhood search metaheuristic that embeds several destroy and repair procedures and selects them based on past performance, according to a roulette-wheel principle. Overall the proposed methodology was highly successful and has enabled its authors to tie for second place in the challenge.

Acknowledgements This research was partially funded by the Canadian Natural Sciences and Engineering Research Council under grants 227837-04 and 39682-05, and by the strategic research workshops program of HEC Montréal. This support is gratefully acknowledged. Thanks are also due to Nadia Lahrichi for her technical support and to three anonymous referees for their valuable comments.

References

Ahuja, R. K., Ergun, Ö., Ergun, J. B., & Punnen, A. P. (2002). A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, *123*, 75–102.

Begur, S. V., Miller, D. M., & Weaver, J. R. (1997). An integrated spatial DSS for scheduling and routing home-health-care nurses. *Interfaces*, *27*(4), 35–48.

Bellenguez, O., & Néron, E. (2005). Lower bounds for the multi-skill project scheduling problem with hierarchical levels of skills. In E. K. Burke & M. Trick (Eds.), *Lecture notes in computer science: Vol. 3616. Practice and theory of automated timetabling V: 5th international conference, PATAT 2004* (pp. 229–243). Berlin: Springer.

Bellenguez-Morineau, O., & Néron, E. (2007). A branch-and-bound method for solving multi-skill project scheduling problem. *RAIRO—Operations Research*, *41*, 155–170.

Bertels, S., & Fahle, T. (2006). A hybrid setup for a hybrid scenario: combining heuristics for the home health care problem. *Computers & Operations Research*, *33*, 2866–2890.

Caramia, M., & Giordani, S. (2009). A new approach for scheduling independent tasks with multiple modes. *Journal of Heuristics*, *15*, 313–329.

Caseau, T., & Koppstein, P. (1992). A cooperative-architecture expert system for solving large time/travel assignment problems. In *International conference on databases and expert systems applications* (pp. 197–202).

Danna, E., & Perron, L. (2003). Structured vs. unstructured large neighborhood search: a case study on job-shop scheduling problems with earliness and tardiness costs. In *Lecture notes in computer science* (Vol. 2833, pp. 817–821). Berlin: Springer.

Dutot, P.-F., Laugier, A., & Bustos, A.-M. (2006). *Technicians and interventions scheduling for telecommunications*. France Telecom R&D, August 2006.

Estellon, B., Gardi, F., & Nouioua, K. (2008). Recherche locale haute performance pour la planification des interventions à France Télécom. In *JFPC 2008—Quatrièmes Journées Francophones de Programmation par Contraintes*. Nantes, France.

Eveborn, P., Flisberg, P., & Rönnqvist, M. (2006). Laps care—an operational system for staff planning of home care. *European Journal of Operational Research*, *171*, 962–976.

Hurkens, C. A. J. (2007). Incorporating the strength of MIP modeling in schedule construction. In G. Finke (Ed.), *ROADEF 2007, le 8ème Congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision*. Grenoble, France.

Kirkpatrick, S., Gelatt, C. D. Jr., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680.

Laborie, P., & Godard, D. (2007). Self-adapting large neighborhood search: application to single-mode scheduling problems. In *Proceedings of MISTA-07*, Paris, France.

Meyers, C., & Orlin, J. B. (2007). Very large-scale neighborhood search techniques in timetabling problems. In *Lecture notes in computer science* (Vol. 3867, pp. 24–39). Berlin: Springer.

Mladenović, N., & Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research*, *24*, 1097–1100.

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

Pisinger, D., & Ropke, S. (2007). A general heuristic for vehicle routing problems. *Computers & Operations Research*, *34*, 2403–2435.

Rios-Solis, Y. A., & Sourd, F. (2008). Exponential neighborhood search for a parallel machine scheduling problem. *Computers & Operations Research*, *35*, 1697–1712.

Ropke, S., & Pisinger, D. (2006a). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, *40*, 455–472.

Ropke, S., & Pisinger, D. (2006b). A unified heuristic for a large class of vehicle routing problems with backhauls. *European Journal of Operational Research*, *171*, 750–775.

Schrimpf, G., Schneider, J., Stamm-Wilbrandt, H., & Dueck, G. (2000). Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, *159*, 139–171.

Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In *Lecture notes in computer science: Vol. 1520. CP-98, fourth international conference on principles and practice of constraint programming* (pp. 417–431). Berlin: Springer.

Tsang, E., & Voudouris, C. (1997). Fast local search and guided local search and their application to British telecom's workforce scheduling problem. *Operations Research Letters*, *20*, 119–127.

Weigel, D., & Cao, B. (1999). Applying GIS and OR techniques to solve Sears technician-dispatching and home-delivery problems. *Interfaces*, *29*(1), 112–130.

Xu, J., & Chiu, S. Y. (2001). Effective heuristic procedures for a field technician scheduling problem. *Journal of Heuristics*, *7*, 495–509.