CEJOR (2011) 19:467–493
DOI 10.1007/s10100-010-0137-8

ORIGINAL PAPER

A tabu search tutorial based on a real-world scheduling problem

Ulrike Schneider

Published online: 6 March 2010

Abstract We apply a tabu search method to a scheduling problem of a company producing cables for cars: the task is to determine on what machines and in which order the cable jobs should be produced in order to save production costs. First, the problem is modeled as a combinatorial optimization problem. We then employ a tabu search algorithm as an approach to solve the specific problem of the company, adapt various intensification as well as diversification strategies within the algorithm, and demonstrate how these different implementations improve the results. Moreover, we show how the computational cost in each iteration of the algorithm can be reduced drastically from $O(n^3)$ (naive implementation) to $O(n)$ (smart implementation) by exploiting the specific structure of the problem (n refers to the number of cable orders).

Keywords Combinatorial optimization · Scheduling · Metaheuristics · Tabu search

1 Introduction

This article is about adapting and implementing the metaheuristic optimization algorithm tabu search to a concrete real-world scheduling application. While the research presented here was conducted for a particular problem, we point out that our work can be useful from a tutorial point of view in different or more general settings of combinatorial optimization problems, or where the corresponding objective function has a similar structure as in our model. The two main points in this regard are the tailoring of the tabu search algorithm involving the employment and adaptation of various tabu search strategies within the search, and the treatment of computational

U. Schneider (✉)
Institute for Mathematical Stochastics, University of Göttingen,
Goldschmidtstr.7, 37077 Göttingen, Germany
e-mail: ulrike.schneider@math.uni-goettingen.de

cost issues involving the structure of storing and updating the objective function in the implementation, both presented in Sect. 4.

The paper is organized as follows. Section 2 describes the problem and its formulation as a combinatorial optimization problem. In Sect. 3, we briefly motivate why we chose to apply a tabu search algorithm, describe the main ideas of this method, and define its basic concepts within the context of our scheduling problem. Different implementations of the algorithm are presented and compared in Sect. 4 where we examine beneficial as well as unprofitable features of the search and also address computational cost. Moreover, Sect. 4 contains a table summarizing all results. Finally, we recapitulate and conclude in Sect. 5, and discuss potential future research.

2 Problem formulation and modeling

This work started with the request of a company producing so-called single core cables for cars to optimize their production procedures. The production process basically amounts to encasing wire with a layer of insulation. The company gets daily orders for specific cables differing in length, diameter and type of wire, insulation material, color and other specifics such as the color of a marker to be placed at the end of the cable. After the order is complete, the different cable jobs are assigned to different machines "by hand" using common sense and experience with the objective to minimize "change-over costs" which occur by switching the production on a machine from one specific cable to another. These costs arise for instance if the color of the insulation material has to be changed, entailing associated cleaning costs, material consumption and idle times of the machine. Moreover, certain kinds of cables can be produced on specific machines only. Typically, about 300 cable jobs have to be assigned to approximately 10 machines. The company would like to automate this process by employing a computer program that, given a list of cable orders as input, can output the "optimal" machine allocation.

2.1 Search space

More concretely, we call this list of cable orders the *job list*, coded $J = \{1, \ldots, n\}$. Assume that we have M different machines, labeled $1, \ldots, M$. With a given job list J, a *feasible solution* is an assignment of the elements of J to an array

$$s(m,k), \quad k = 1, \ldots, n_m; \; m = 1, \ldots, M, \; \sum_{m=1}^{M} n_m = n, \tag{1}$$

where it has been taken into account that certain jobs can only be assigned to certain machines (see Fig. 1). We denote by $S(J)$ the *search* or *solution space*, i.e. the set of all feasible solutions for a given job list J. For formal reasons, we define $s(m, 0) := 0$ for $m = 1, \ldots, M$.

We denote a solution $s(m, k)$ with $k = 1, \ldots, n_m; m = 1, \ldots, M$.

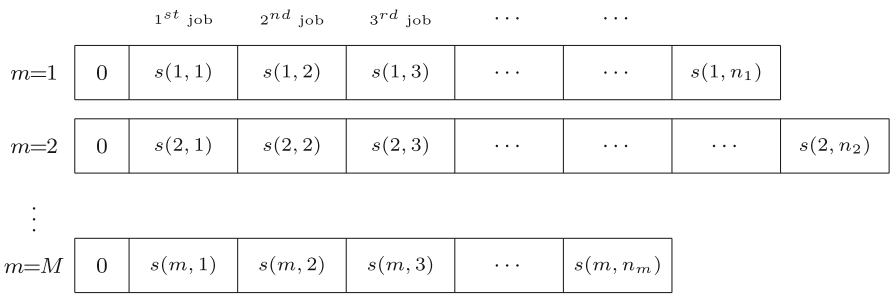

Fig. 1 A machine assignment

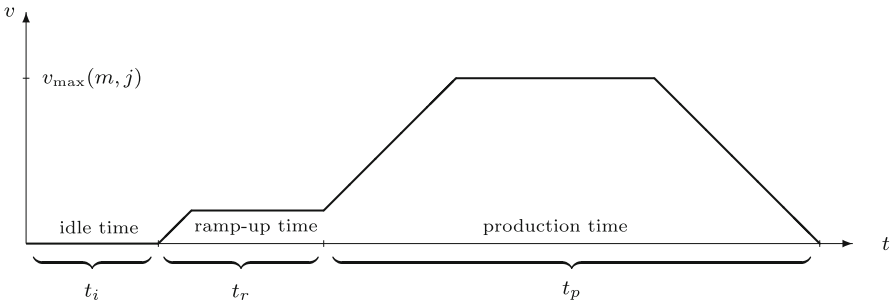

Fig. 2 The 3 time components of a job. $v_{\max}$ denotes the (maximum) speed at which job j can be produced on machine m

2.2 Objective function

The cost of a feasible machine assignment s is the sum of the cost of a job, taken over all given jobs. The cost of a specific job is composed of three different parts. First, there is the cost associated with the *re-tool time* t_i, when the machine is idle and has to be set up for the specific job characteristics. The second cost occurs during the *ramp-up time* t_r, when the machine is run at low speed until the produced cable has the desired quality. Finally, the third component is the cost arising during the *production time* t_p when the actual cable is produced. t_i and t_r depend on the machine m, the characteristics of the job j to be produced, as well as on the specifics of the job j_p produced before job j. t_p depends on m and j only. We model the cost of j linear in time in the following sense.

$$jobcost(m, j_p, j) = c_i(m)t_i(m, j_p, j) + c_r(m)t_r(m, j_p, j) + c_p(m)t_p(m, j),$$

where $c_i(m)$, $c_r(m)$, and $c_p(m)$ denote the cost arising during one time unit of t_i, t_r, and t_p, respectively. These costs are constant on each machine m. The *re-tool cost* c_i involves cost of labor, the *ramp-up cost* reflects the cost of material having to be discarded, and the *production cost* is based on the cost of running the machine. Figure 2 illustrates these three components, showing the speed v of machine m during the production of job j versus time.

As implied by the above, the cost of a specific job is determined by the machine on which it is to be produced and by the specifics of the job that has been assigned to the slot before that job. Note that we defined $s(m, 0) := 0$, and that $jobcost(m, 0, j)$ denotes the cost of job j assigned to the first slot on machine m.

The objective function $f : S(J) \to \mathbb{R}$ therefore, has the following structure.

$$f(s) = \sum_{m=1}^{M} \sum_{k=1}^{n_m} jobcost(m, s(m, k-1), s(m, k)) \tag{2}$$

The above structure of f will be key for efficient updates of f within the tabu search algorithm, see Sect. 4.2.

2.3 The problem

The problem can now be stated as follows.

$$\text{For a given job list } J, \text{ find } s_0 \in S(J) \text{ such that } f(s_0) = \min_{s \in S(J)} f(s). \tag{3}$$

2.4 A note on the complexity

We touch upon the subject of computational complexity here by noting the following considerations. Assume that $M = 1$, that is, we assign all jobs to one machine only. Let $J = \{1, \ldots, n\}$ be the job list and define the $(n+1) \times (n+1)$ matrix $D = (d_{ij})$ by

$$d_{ij} = \begin{cases} jobcost(1, i, j) & j \neq 0 \\ 0 & j = 0 \end{cases}$$

for $i, j = 0, \ldots, n$. We can now interpret the machine scheduling problem as a *traveling salesman problem* with cities $\{0, 1, \ldots, n\}$ and (asymmetric) distance matrix D. A *tour* $(j_0, j_1, \ldots, j_n)$ can be interpreted as a machine assignment by taking the city (job) j_{i+1} after city (dummy job) $j_i = 0$ to be assigned to the first slot on machine m,

$$s(m = 1, .) := (0, j_{i+1}, j_{i+2}, \ldots, j_n, j_0, \ldots, j_{i-1}).$$

The traveling salesman problem is very well known to be $\mathcal{NP}$-hard.

2.5 Classification

Due to its complexity, our optimization problem does not directly fit into the "regular" framework of classifying scheduling problems. For such a framework see, for example, Brucker (2007) who introduces a scheme based on Graham et al. (1979), where problems are classified by *machine environment*, *job characteristics*, and *optimality*

criterion, the latter one including objective functions such as bottleneck and makespan. We note, however, that in the context of this framework, the job characteristics of our model are specified by so-called *s-batching* (see Sect. 3) and that the machine environment corresponds to unrelated multi-purpose machines.

3 Using tabu search

The problem, as stated in (3) is a combinatorial optimization problem and these types of problems are often tackled with *metaheuristic search methods*.

Metaheuristic search methods are *local search* or *neighborhood search algorithms* which are iterative procedures that start with an *initial solution* $s_0 \in S$ and then browse through the search space, picking up better solutions on the way. They use the concept of a neighborhood structure, that is, a function $\mathcal{N} : S \to 2^S$ assigning to each solution $s \in S$ a *neighborhood* $\mathcal{N}(s) \subseteq S$ of solutions that are in some sense "close" to s. As a convention, we assume that $s \notin \mathcal{N}(s)$. With the notion of neighborhood, it is natural to define a *local minimum* as an element $s \in S$ that satisfies $f(s) \leq f(s')$ for all $s' \in \mathcal{N}(s)$. The basic framework of such a local search algorithm can be formulated as a generic *descent method* in Algorithm 1.

Algorithm 1 A generic descent method

1. Initialize: find an initial solution $s_0 \in S$ and set $s = s_0$, $f = f(s_0)$, $s^* = s$, $f^* = f$.
2. Neighborhood search step: find $s' \in N(s)$ such that $f(s') \leq f$ and set $s = s'$, $f = f(s')$. Additionally, if $f < f^*$, set $s^* = s$, $f^* = f$.
3. Stop if no such s' exists or some other stopping rule applies. Otherwise, go to 2.

The "answer" of the algorithm is the solution s^* with objective function value f^*. If in step 2, s' is found in such a way so that $f(s') < f(s'')$ for all $s'' \in \mathcal{N}(s)$, the algorithm is called a *steepest descent method*.

Obviously, the algorithms described above get trapped in the first local minimum they encounter whereas metaheuristic methods are local search methods that have strategies to go beyond local optima. They can constitute a favorable compromise between expensive exact methods and the mere local pretense.

All metaheuristic search methods use a basic neighborhood search as their foundation and can formally be brought into the framework of Algorithm 1 where the construction of the neighborhood, say $\mathcal{N}^*(s)$, can dynamically vary during the search process. Among those methods are tabu search, simulated annealing, genetic algorithms, neural networks, scatter search, and ant colony optimization. The difference in these methods can be seen as a difference in the way the neighborhood $\mathcal{N}^*(s)$ is defined and altered. For a more comprehensive treatment of metaheuristic techniques, see, for example, Reeves and Beasley (1993), Aarts and Lenstra (1997) and Blum and Roli (2003).

To solve our scheduling problem, we apply the metaheuristic technique *tabu search*. Tabu search has been successfully applied to a wide range of applications, and seems to be unrivaled in efficacy for a variety of scheduling problems. See, for example

Nowicki and Smutnicki (1996) and Grabowski and Wodecki (2004). For other metaheuristic approaches in scheduling problems we refer the reader to Tan et al. (2007), for example, and the references therein. Abdullah et al. (2007) contains a nice literature overview on these kinds of problems including algorithms with more than one neighborhood structure. A comprehensive overview on scheduling problems and solution approaches can be found in Leung (2004).

We now discuss the basic concepts involved in tabu search. A number of foundation ideas were first introduced in Glover (1977), with a more detailed and comprehensive description provided in Glover (1989, 1990). A general and extensive treatment can be found in Glover and Laguna (1997). The most important feature of tabu search is its emphasis on the use of *adaptive memory*—the dynamic adaption and update of the neighborhood structure guided by information gained throughout the search process. Such memory can either be *explicit* where entire solutions are stored, or it can record only certain *attributes* of solutions that have been visited. Usually four different structures are distinguished. *Frequency memory* stores information about how often chosen solutions exhibit a certain attribute, whereas *recency memory* keeps track of how lately this has been the case. *Quality memory* saves information on how "good" a particular attribute has proven to be, and with *influence memory* we retain evidence how solutions with a certain attribute have influenced the search process. Generally, frequency memory is employed as a *long term memory* concept in the sense that the information recorded throughout the search process. Recency memory falls under the use of *short term memory* which only stores what has happened in recent iterations. Depending on their use, quality and influence memory can fall under either category. These memory concepts are usually employed for either *intensification* purposes, that is to encourage the search to explore certain regions in more detail, or for *diversification* reasons with the goal of guiding the algorithm to previously unexplored areas of the search space. So-called *candidate lists* of "promising" neighbors can be used if exploring the entire neighborhood in each iteration becomes too computationally expensive.

3.1 Adapting basic tabu search elements

We describe the use of the very basic tabu search elements within the context of our machine scheduling problem: the program to carry out the optimization process is implemented in the C programming language and executes the following steps.

1. Read-in of job list J
2. "Batch" jobs with same specifics
3. Generate an initial solution
4. Tabu search phase
5. Output of relevant data

Clearly, the tabu search phase (4) constitutes the main part of the program and is treated in detail in Sect. 4 where the basic elements introduced in this section are developed to become more advanced throughout the different versions of the program.

Input and internal storage

First, the program reads in a text file with a list of the current cable orders. The input file contains the specifics of each cable, including length, type of wire, color of insulation material, and so on. Internally, the jobs are numbered $1, \ldots, n$ and stored in an object (array) $joblist$ that contains the relevant data for each job j.

Batching and initial solution

After reading in the current list of cable orders, jobs with exactly the same specifics except for length are batched into one job, since it is clear that these jobs should be produced consecutively. Afterwards, an initial solution is generated in the following way. The jobs are split into two lists L_1 and L_2, where L_1 contains the jobs which have to be produced on specific machines, and L_2 the remaining ones which can be assigned to any machine. Each list is then sorted by color of insulation material before the jobs from L_1 are allocated to admissible machines and the jobs from L_2 are put on the "remaining" spots (in the sense that in the initial solution the first $n - M\lfloor n/M \rfloor$ machines contain $\lfloor n/M \rfloor + 1$ jobs and the other ones contain $\lfloor n/M \rfloor$ jobs each). This way each machine should have about the same job load (neglecting unlikely cases where L_1 contains so many jobs that such a uniform assignment is not possible.)

This type of initial solution mimics the process in which cable jobs are manually assigned to the machines by the employees of the company.

In the following, we shall refer to a machine assignment or solution by $s = s(m, k)$ as introduced in (1) in Sect. 2.1.

Cost function

A routine *jobcost* has been implemented to compute the corresponding element of the cost function f in the notation of (2) in Sect. 2.2. To ensure that the computation of f is well defined in the sense that the same machine assignment always leads to the same value of the cost function, the routine *jobcost* is integer-valued in order to avoid rounding issues. This is important since the value of the objective function of a particular solution will be calculated or updated in different ways with the routine *jobcost* representing the smallest unit in these calculations.

Neighborhood structure

The neighborhood of a feasible solution $s \in S(J)$ are all solutions $s' \in S(J)$ which can be reached by exchanging two jobs in s. We call such an exchange a *swap move* μ and represent it by a 4-tuple $\mu = (m_1, k_1, m_2, k_2)$ where μ causes to switch $s(m_1, k_1)$ with $s(m_2, k_2)$. We write $s' = \mu[s]$ to emphasize that a move is acting on a solution s. Without loss of generality we assume $m_1 \leq m_2$. Additionally, if $m_1 = m_2$, we let $k_1 < k_2$. Observe that once an initial solution is generated, the structure of all subsequent solutions is locked-in in the sense that the number n_m of jobs on machine m does not change anymore. (This entails that if the initial solution exhibits a uniform workload it will be kept in subsequent solutions.) Later on, we will introduce another

representation, as well as an additional neighborhood structure defined by a different type of move.

In this work, we only consider *transition moves* that operate on a given solution, but it should be noted that the metaheuristic framework also encompasses *constructive* (and *destructive*) *moves* in which solutions can be generated "from scratch", as noted in Chapter 1.9.1 in Glover and Laguna (1997).

Guaranteeing feasibility

As described in Sect. 2, certain cable jobs can only be produced on certain machines. In view of this, given a solution s, we call a swap move $\mu = (m_1, k_1, m_2, k_2)$ a *feasible move* if it leads to another feasible machine assignment. The feasibility of μ therefore depends on whether it is possible to produce job $s(m_1, k_1)$ on machine m_2 *and* job $s(m_2, k_2)$ on machine m_1. We denote by $\mathcal{M}(s)$ the collection of all feasible swap moves for s to define the neighborhood $\mathcal{N}(s)$ as

$$\mathcal{N}(s) := \{s' = \mu[s] \mid \mu \in \mathcal{M}(s)\},$$

therefore, assuring that $\mathcal{N}(s) \subseteq S(J)$. In all versions of the program presented in Sect. 4, a move under consideration is first checked for feasibility (i.e. whether it entails moving a job to a machine where it cannot be produced). If this is the case, the particular move is not considered in the current iteration. In this way, we never need to establish feasibility for an entire solution "from beginning to end".

Tabu strategy

To avoid the search from cycling, the last T moves (not the explicit solutions) during the search process are set "tabu" and a *tabu list* contains the last T moves made. T is called the *tabu tenure*. If we are currently in iteration i in the search process and label the moves conducted so far by $\mu_1, \ldots, \mu_i$, then the tabu list $\mathcal{T}(i)$ in iteration i can by written as

$$\mathcal{T}(i) = \{\mu_i, \mu_{i-1}, \ldots, \mu_{\max(i-T+1,1)}\}.$$

$\mathcal{T}$ constitutes an example of recency memory used for diversification purposes.

Aspiration criterion

To avoid excluding obviously good solutions from the search by the tabu strategy, we use the standard aspiration function (improved-best). Given the current solution s, this criterion induces a set of "aspired" moves from the tabu list, denoted $\mathcal{A}(s, i) \subseteq \mathcal{T}(i)$ with

$$\mathcal{A}(s, i) = \{\mu \in \mathcal{T}(i) \mid f(\mu[s]) < f^*\}$$

which is used in all following versions of the program.

The actual neighborhood

With the notation introduced above, the actual neighborhood used in iteration i in the basic tabu search algorithm can be written as

$$\mathcal{N}^*(s,i) := \{s' \in \mathcal{N}(s) \mid \mu \in (\mathcal{M}(s) \setminus \mathcal{T}(i)) \cup \mathcal{A}(s,i)\}.$$

In the following, we will drop the iteration i in the notation and simply write $\mathcal{N}^*(s)$.

Neighborhood search step

In each iteration of the algorithm, a "best" neighbor s' is chosen as the current solution for the next iteration. If s denotes the current solution, a new solution s' is chosen such that

$$f(s') = \min_{s'' \in \mathcal{N}^*(s)} f(s'')$$

Clearly, the minimizer above does not need to be unique. We simply chose the first neighbor encountered in that iteration with minimal objective value.

Stopping criterion

The algorithm stops after a fixed number of iterations, say, N.

Output

In addition to the best machine assignment found, the program outputs certain parameters that describe the search process which are summarized in Tables 1 and 2 in Sect. 4.8.

4 The different versions

We now discuss different implementations of the tabu search algorithm to our machine scheduling problem. The versions are presented in the chronological order of their development where each version addresses particular problems of its predecessor. The last version is by no means "ultimate", yet each program outperforms the previous one and we briefly list all implementations to examine the elements that improved the search. A summary of effective and not so useful elements is given in Sect. 5.

Additionally, we discuss the computational cost for calculating the objective function and show how to take advantage of the specific structure of our problem. We include patches of the program in pseudo-code manner where we believe them to benefit the presentation.

The results of the different versions can be found in Table 1 in Sect. 4.8, details of the search process are listed in Table 2 in the same section. We refer the reader to these tables when discussing the outcome of the various programs in the following.

Table 1 Overview and results of all versions

	$ts1$	$ts2$	$ts3$	$ts4$	$ts5$	$ts6$	$ts7$
$f(s_0)$	152,587	152,587	152,587	152,587	152,587	152,587	152,587
$f(s^*)$	142,736	142,503	142,420	139,500	139,190	139,021	138,856
Found in iteration	85	97	639	89	701	899	979
Impr. rel. to $f(s_0)$	**6.46%**	**6.61%**	**6.66%**	**8.58%**	**8.78%**	**8.89%**	**9.00%**
Impr. rel. to prev. vers.	–	**0.15%**	**0.06%**	**1.92%**	**0.2%**	**0.11%**	**0.11%**
# Real improvements	85	97	109	89	188	231	165
With swap moves	85	97	109	–	99	142	72
With insert moves	–	–	–	89	89	89	93
Diversification	No	No	Yes	No	Yes	Yes	Yes
For swap moves	No	No	Yes	–	Yes	Yes	Yes
For insert moves	–	–	–	No	No	No	Yes
Intensification	No	No	No	No	No	Yes	Yes
For swap moves	No	No	No	–	No	Yes	Yes
For insert moves	–	–	–	No	No	No	No
# Function calls	$5.9 \cdot 10^9$	$4.0 \cdot 10^6$	$5.4 \cdot 10^6$	$7.2 \cdot 10^6$	$8.4 \cdot 10^6$	$8.1 \cdot 10^6$	$4.3 \cdot 10^6$
Reduced to	–	0.07%	0.09%	0.12%	0.14%	0.14%	0.07%
Run time approx.	11 h	40 s	40 s	40 s	40 s	40 s	20 s

All versions were run with a real-life instance job list of $n = 300$ jobs representing a typical daily workload (we found analogous results for other input lists) to be assigned to the $M = 11$ machines the company has available.

The construction of the initial solution described in Sect. 3.1 follows the spirit of the company's manual approach for which only verbal descriptions but no hard data exists. As a basis for comparison between the solutions suggested by procedure and

Table 2 Details about the search process of all versions

	$ts1$	$ts2$	$ts3$	$ts4$	$ts5$	$ts6$	$ts7$
# Swap moves	1,000	1,000	1,000	–	500	500	790
# insert moves	–	–	–	1,000	500	500	210
# Switches betw. nbhds	–	–	–	–	9	9	4
# 0-cost moves	915	903	889	911	812	323	96
# Uphill moves	0	0	0	0	0	0	62
# Downhill moves	0	0	1	0	0	221	362
# Tabu moves	0	0	0	0	0	0	62
# Pen. record active	–	–	879	–	299	149	19
With swap moves	–	–	879	–	299	149	0
With insert moves	–	–	–	–	–	–	19

what is currently done by the company, we looked at the improvement in the objective function between the initial solution of the algorithm (representing the company's behavior) and the best solution found by the procedure (representing our suggestion).

We used a total number of $N = 1{,}000$ iterations and a tabu tenure of $T = 40$. The choice of other search parameters is also listed within the discussion as well as in Table 3 in Sect. 4.8. If not noted otherwise, parameters are kept at the values same for subsequent versions. The results of the algorithm appeared to be robust with respect to similar values of these variables. The CPU times shown in Table 1 result from executing the program on a AMD Athlon(tm) 64 X2 Dual Core Processor with 3,000 MHz and 1,024 KB cache size. They are listed to give a general idea. The C code was compiled using *gcc*, the GNU C compiler.

4.1 ts1—The basic version

The basic version, $ts1$, is mainly implemented to serve as a starting point for the improved versions and for comparison and illustration purposes. In this version, a basic tabu search algorithm is carried out with the definitions from Sect. 3 as demonstrated in Algorithm 2 below. The tabu list is the only relevant array that needs to be initialized and updated in line 2 and line 14 in Algorithm 2.

Table 3 Parameter settings and versions they have been used in

	Value	$ts1$	$ts2$	$ts3$	$ts4$	$ts5$	$ts6$	$ts7$
Number of iterations N	1,000	×	×	×	×	×	×	×
Tabu tenure T	40	×	×	×	×	×	×	×
P_{fac} (see Sect. 4.3)	5			×		×	×	×
P_{start} (see Sect. 4.3)	70			×		×	×	×
$n_{\max}$ (see Sect. 4.4)	100				×	×	×	×
P_{lev} (see Sect. 4.6)	2						×	×
P_{gap} (see Sect. 4.6)	2						×	×
$C_{\max}$ (see Sect. 4.6)	5						×	×
C_{tot} (see Sect. 4.6)	$N/2$						×	×
T_{start} (see Sect. 4.6)	$2/3 \cdot N$						×	×
T_{sup} (see Sect. 4.6)	2						×	×
N_{fac} (see Sect. 4.7)	0.2							×
N_{prop} (see Sect. 4.7)	0.1							×

Algorithm 2 Tabu search algorithm ts1

1. Generate an $s_0 \in S$ and set $s = s_0$, $f = f(s_0)$, $s^* = s$, $f^* = f$, $f^*_{nb} = 10^9$.
2. Initialize relevant arrays.
3. FOR $i = 1 : N$ DO
4. FOR all feasible moves $\mu \in \mathcal{M}(s)$ DO
5. $f' = f(\mu[s])$
6. IF ($f' < f^*_{nb}$) AND ((μ is not tabu) OR ($f' < f^*$)) DO
7. $\mu^*_{nb} = \mu$, $f^*_{nb} = f'$
8. END (IF)
9. END (FOR)
10. $s = \mu^*_{nb}[s]$, $f = f^*_{nb}$, $f^*_{nb} = 10^9$
11. IF $f < f^*$ DO
12. $s^* = s$, $f^* = f$
13. END (IF)
14. Update relevant arrays.
15. END (FOR)

We start our discussion by looking at the computational cost and the evaluation of the cost function.

Computational cost

In each iteration with current solution s, the objective function is evaluated for each point $s' \in \mathcal{N}^*(s)$. In the general form of the objective function as shown in (2) in Sect. 2.2, the function *jobcost* that computes the cost of a specific job constitutes the smallest unit of f in a rather general sense and we therefore, use the number of function calls to this routine to assess the computational effort. We wish to emphasize that by this measure of computational cost, we do not refer to the concept of computational

complexity in view of $\mathcal{NP}$-hardness—we merely discuss the cost of each iteration of our algorithm (and not the number of steps required to find an optimal solution).

Pseudo-code 1 illustrates the naive evaluation of the objective function for a given solution s' as needed in line 5 of Algorithm 2.

Pseudo-code 1 Naive evaluation of the objective function

```
f' = 0
FOR m = 1 to M DO
    FOR k = 1 to n_m DO
        j = s'(m, k), j_prev = s'(m, k − 1)
        f' = f' + jobcost(m, j_prev, j)
    END (FOR)
END (FOR)
```

Clearly, for each evaluation of the objective function f for a given solution s, the routine *jobcost* is called $n_1 + \cdots + n_m = n$ times. In each iteration, the objective function is evaluated for every point $s' \in \mathcal{N}(s)$ (due to the aspiration criterion, we also have to evaluate the solutions reached by tabu moves). Disregarding the number of infeasible moves, this amounts to having to compute f for the $1/2 \cdot n(n-1)$ neighbors of s and thus yielding a computational effort of $O(n^3)$ function calls to the basic routine *jobcost* in each iteration.

Results

The results in Table 1 show that after generating the initial solution, in the first 85 steps the algorithm finds 85 new best solutions or *real improvements*, meaning solutions with lowest cost compared to all previously encountered solutions. The algorithm only finds neighbors with no change in the objective function in the remaining iterations. The total improvement in the objective function was 6.46% compared to the initial solution. The algorithm took more than 11 h.

4.2 ts2—Smart update

The output of $ts1$ shows that the search in its current form should not be viewed as *meta*heuristic since it gets stuck in the first local minimum it encounters. Before going further into this issue, we wish to address the computational cost of the algorithm to make the search amendable to testing different features.

Clearly, the comprehensive evaluation of the objective function as implemented in $ts1$ is unnecessary since the transition from one solution to another only results in minor changes of the current machine assignment. Instead of associating the cost $f(s')$ with each neighbor s' of s, we assign a cost value $c(s, \mu)$ to each move $\mu \in \mathcal{M}(s)$, defined as the change in the objective function occurring by μ,

$$c(s, \mu) := f(s') - f(s) \text{ for } s' = \mu[s].$$

Notation

For the sake of simplicity, we now label the *position* p of jobs on the machines from 1 to n, instead of using the *coordinates* (m, k) as before. Note that once an initial solution $s_0 \in S$ is generated, n_m, the number of jobs on machine m does not change in subsequent solutions, so that the coordinates of a position are fixed throughout the search. Using this notation, we now represent a move μ by a tuple (p, q) with $p < q$, indicating the exchange of the job with position p with the job on position q. This labeling of positions will prove very useful in storing and updating the cost of moves.

If not noted otherwise, by j_1 and j_2, we denote the corresponding jobs of the move (p, q), and write (m_1, k_1) and (m_2, k_2) for the respective coordinates of j_1 and j_2. We declare an $n \times n$ array of integers $swap\_cost[\,p\,][\,q\,]$ to store the cost of swap moves in a given iteration. If a move is infeasible, we assign a very large value to the corresponding entry in *swap_cost*.

As mentioned above, a neighbor $s' \in \mathcal{N}(s)$ differs "only slightly" from s, and therefore, the cost value of most moves does not change either from one iteration to another. If, for instance $n = 300$, 98% remain the same. More concretely, between 4 and 8 jobs are affected in each iteration, each of which is involved in $n - 1$ moves. For each of these moves, we need to update (not recalculate!) $c(s, \mu)$ with at most 8 function calls to *jobcost*, as is described in detail below.

A new tabu strategy

The basic purpose of a tabu strategy is to prevent revisiting the same or similar solutions to those that have been looked at before. In view of this, the previous strategy unnecessarily prohibited certain moves, since a move could in later iterations involve different jobs than it originally did. Moreover, the tabu concept could not impede jobs from being moved back to a previously occupied position by a different move. In addition, to establish the tabu status of a move, it had to be compared with all moves stored in the tabu list.

Based on these considerations, we employ a new strategy using an $n \times n$ array $swap\_tabu[\,j\,][\,p\,]$, which stores the iteration until which job j must not be brought back into position p. With this idea, the determination of tabu status as well as its update is quite easy. If we have chosen the move $\mu = (p, q)$ in the iteration i, we simply put

$$\begin{aligned} swap\_tabu[\,j_1\,][\,p\,] &= i + T \\ swap\_tabu[\,j_2\,][\,q\,] &= i + T \end{aligned}$$

In this sense, each neighbor $s' = \mu[s]$ under consideration has two possible tabu attributes, one for each job j_1 and j_2 that is moved under $\mu = (p, q)$. We set μ to be tabu, if both corresponding attributes are tabu-active, that is, if

$$\begin{aligned} swap\_tabu[\,j_1\,][\,p\,] &>= i \quad \text{AND} \\ swap\_tabu[\,j_2\,][\,q\,] &>= i \end{aligned}$$

Algorithm

In each iteration we choose the move $\mu = (p, q)$ such that *swap_cost*[p][q] is minimized (over all (p, q) with $p, q = 1, \ldots, n$ and $p < q$). For a move $\mu = (p, q)$ in Algorithm 2, we replace line 5 by the following.

$5'$ $\quad f' = f + swap\_cost[\,p\,][\,q\,]$

We note that also both arrays *swap_cost* and *swap_tabu* need to be initialized and updated in line 2 and line 14 of the algorithm, respectively. The fast update of *swap_cost* is described in the following.

From $\mathbf{O(n^3)}$ *to* $\mathbf{O(n)}$

In each iteration, the routine *jobcost* has to be called between 4 and 8 times (as described below) for each of $4(n-1)$ to $8(n-1)$ affected moves, resulting in a computational cost of at most $64(n-1) = O(n)$ function calls to the routine *jobcost*.

In Pseudo-code 2 we describe the update of a *single* entry *swap_cost*[p][q] in the cost array for which up to 8 function calls to the routine *jobcost* are necessary. We assume that (p, q) represents a "typical" swap move, where neither p nor q refers to a position at the end of a machine. (In that case, less calls to *jobcost* are involved.) The routine $swap\_coord(s, p)$ computes the coordinates (m, k) of the job on position p on the machine assignment s.

Pseudo-code 2 Fast update of swap move costs

$$
\begin{aligned}
(m_1, k_1) &= swap\_coord(p)\\
j_1prev &= s(m_1, k_1 - 1)\\
j_1cur &= s(m_1, k_1)\\
j_1next &= s(m_1, k_1 + 1)\\
\\
(m_2, k_2) &= swap\_coord(q)\\
j_2prev &= s(m_2, k_2 - 1)\\
j_2cur &= s(m_2, k_2)\\
j_2next &= s(m_2, k_2 + 1)\\
\\
old\_cost &= jobcost(m_1, j_1prev, j_1cur) + jobcost(m_1, j_1cur, j_1next) +\\
&\quad\ jobcost(m_2, j_2prev, j_2cur) + jobcost(m_2, j_2cur, j_2next)\\
new\_cost &= jobcost(m_1, j_1prev, j_2cur) + jobcost(m_1, j_2cur, j_1next) +\\
&\quad\ jobcost(m_2, j_2prev, j_1cur) + jobcost(m_2, j_1cur, j_2next)\\
\\
swap\_cost[\,p\,][\,q\,] &= new\_cost - old\_cost
\end{aligned}
$$

Results

When comparing $ts1$ and $ts2$, certainly the most obvious difference lies in the run time together with the fact that $ts2$ only needs 0.07% of the function calls of $ts1$ as a

consequence of the more effective evaluation or update of the objective function. The slightly better result in terms of solution quality is due to the new tabu strategy.

Note that the fast search has the merit of being able to easily search the entire neighborhood, making the use of elements such as candidate lists unnecessary.

4.3 ts3—With diversification

As mentioned before, $ts1$ as well as $ts2$ exhibit the problem of only choosing so-called *zero-cost moves* (moves that lead to no change in the objective function) after a series of real improvements. To remedy this, we implement a diversification strategy in this version to advance the search into new regions of the search space.

Diversification

The following strategy is based on the assumption that new (good) solutions can be found by choosing moves whose corresponding jobs have rarely been exchanged previously. The selection of such neighbors is encouraged by penalizing moves which switch jobs that have frequently been moved before. This is accomplished by declaring an array $pen\_rec[\,j\,]$ of size n that indicates how many times job j has been moved before. To update this array after an iteration, we simply need to add 1 to the two corresponding entries of the selected move. The cost of a move $\mu = (p, q)$ acting on jobs j_1 and j_2 is modified in the following way.

$$swap\_cost[\,p\,][\,q\,] + P_{\text{fac}} * (pen\_rec[\,j_1\,] + pen\_rec[\,j_2\,])$$

We maintain pen_rec during the entire search, but only use the penalty terms after a certain number of iterations, say $P_{\text{start}} = 70$ with no improvement in f. P_{fac} was set to 2. In order not to disturb the search when "things are going well", the terms are deactivated as soon as a neighbor with a lower value of f than the current solution is found. They are employed again after a non-improving move.

Algorithm

For $ts3$, line 5 of Algorithm 2 needs to be replaced by the following lines. Clearly, in addition to the move costs and the tabu array, the penalty record has to be initialized and updated in line 2 and line 14 of the algorithm, respectively. We also initialize the variable *gap* to be zero and *pen_active* to be false in line 2 of Algorithm 2.

$5'$ $(m_1, k_1) = swap\_coord(p), (m_2, k_2) = swap\_coord(q)$
$j_1 = s(m_1, k_1), j_2 = s(m_2, k_2), pen = 0$
IF $(gap \geq P_{\text{start}})$ OR $((pen\_active)$ AND $(gap \geq 1))$ DO
  $pen\_active =$ TRUE
  $pen = P_{\text{fac}} * (pen\_rec[\,j_1\,] + pen\_rec[\,j_2\,])$
END (IF)
$f' = f + swap\_cost[\,p\,][\,q\,] + pen$

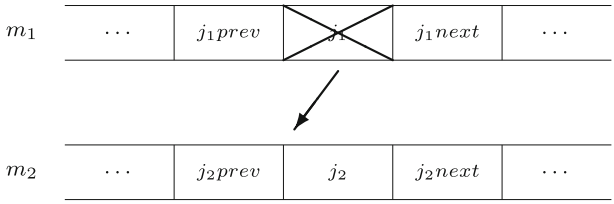

Fig. 3 An insert move

The variable *gap* needs to be updated in line 14 of Algorithm 2 in the following manner.

IF ($f' \geq f$) DO $gap = gap + 1$ END (IF)
ELSE DO $gap = 0$ END (ELSE)

Results

Unfortunately, $ts3$ it only brought a minor improvement in the results. The slightly longer run time can be attributed to the using pen_rec. However, it should be noted that the search was also forced to move "uphill" which is essential in advancing to unexplored regions of S.

4.4 ts4—A new neighborhood

Since the diversification strategy from Sect. 4.3 did not bring the desired effect, we revert to another concept of "diversification" by introducing a new and more dynamic neighborhood that allows the algorithm to search along completely different paths in S. As a first step, we consider this new neighborhood 'exclusively' without the previous swap moves.

New neighbors

The new neighborhood is defined by a different class of moves, the so-called *insert moves*. An insert move causes a job to be deleted from a certain position and to be inserted on another one as illustrated in Fig. 3. For computational simplification which should become apparent later on, we only consider moves that insert jobs on a machine different to the one from where it was deleted. This is done under the assumption that jobs within a machine can be sufficiently mixed by swap moves in later versions.

We represent an insert move ν by a tuple (j_1, j_2) where ν causes job j_1 to be deleted from its current position and to be inserted before job j_2.

It should be noted that there are n *delete positions* from which a job can be removed, but $n+M$ *insert positions* where it can be pasted, since the job can also be inserted *after* the last one on a machine. Since $\nu = (j_1, j_2)$ denotes that job j_1 should be inserted *before* j_2, we define additional dummy jobs $n+1, \ldots, n+M$ where $\nu = (j_1, j_2)$ with $j_2 = n + m$ indicates that job j_1 should be inserted to be the last job on machine m. Note that through insert moves, it is possible that machines are emptied completely.

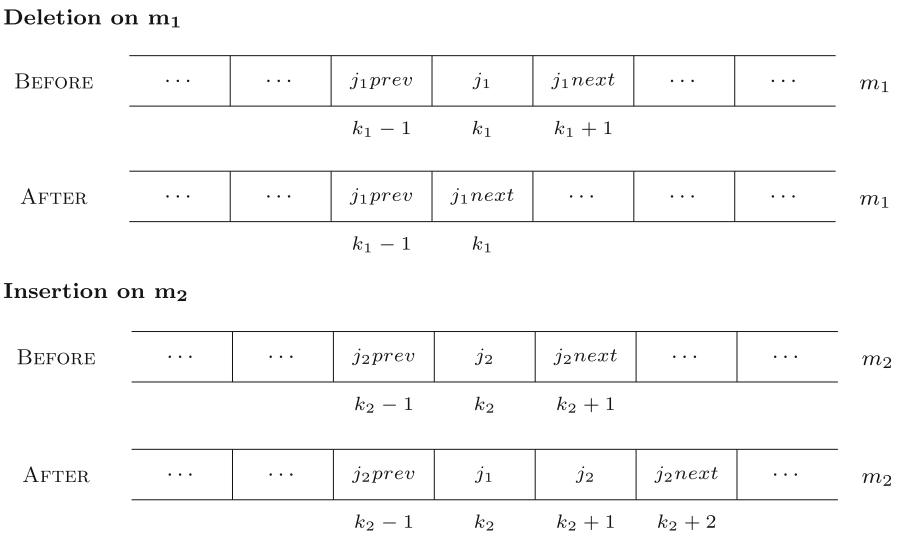

Fig. 4 The insert move $\nu = (j_1, j_2)$ acting on a machine assignment s

To ensure a somewhat uniform workload on each machine (which is favored in this problem), we only allow for a maximum number, say, $n_{\max}$ of jobs per machine. The extent of this restriction can be regulated by the parameter $n_{\max}$, where $n_{\max}$ should be determined depending on M. For the subsequent results, we had $n_{\max}$ set to 100. We maintain this restriction simply by viewing a move infeasible that would shift a job to a machine with a workload of already $n_{\max}$ jobs.

A tabu strategy for insert moves

Since insert moves shift many jobs without actually affecting them, the tabu strategy from Sect. 4.2 does not make sense in this type of neighborhood. It is rather significant for a job to which machine it is assigned and what job its predecessor is. Similar to *swap_tabu* in Sect. 4.2, the $M \times n \times n$ array $ins\_tabu[m][j\_prev][j]$ that indicates until which iteration j cannot be assigned to the slot after j_prev on machine m. The move described in Fig. 4 would be set tabu in iteration i if

$$\begin{aligned} ins\_tabu[m_1][j_1prev][j_1next] &\leq i \quad \text{OR} \\ ins\_tabu[m_2][j_2prev][j] &\leq i \quad \text{OR} \\ ins\_tabu[m_2][j][j_2next] &\leq i \end{aligned}$$

A neighbor reached by move ν therefore, has 3 tabu attributes and ν is set tabu if any of them is tabu-active. The update works analogously to the update of $swap\_tabu$ described in Sect. 4.2.

Computing and updating the cost of insert moves

In the following, for the insert move $\nu = (j_1, j_2)$ we denote by (m_1, k_1) and (m_2, k_2) the coordinates of job j_1 and j_2 on the current machine assignment s. The routine $ins\_coord(s, j)$ can compute the coordinates of job j in the solution s. We want to adapt a similar treatment of insert moves as was done for swap moves in Sect. 4.2. Since insert moves allow for a more dynamic neighborhood, they also require more thought on how to store and update their cost.

We first split the cost of an insert move $\nu = (j_1, j_2)$ into a *deletion cost* c_{del} and an *insertion cost* c_{ins} which denote the change in cost occurring by deleting the corresponding job j_1 and inserting j_1 before j_2, respectively. An essential observation is that these two types of costs can be computed independently of each other since we assumed that $m_1 \neq m_2$. Moreover, we observe that c_{del} depends on j_1, but not on j_2. We can therefore, write the cost of move $\nu = (j_1, j_2)$ as

$$\begin{aligned} c(s, \nu) &:= f(s') - f(s) \\ &= c_{\text{del}}(s, j_1) + c_{\text{ins}}(s, j_1, j_2), \end{aligned}$$

where $s' = \nu[s]$. Similar to the swap moves μ, we store and update the cost of insert moves ν in a given iteration for the current solution s throughout the search.

To this end, we define 3 arrays. One to store c_{del}, one to record c_{ins}, and one to keep track of infeasible moves. We start with the $n \times 1$ array $c\_del[\,j\,]$ which denotes the cost of deleting job j from its current position. We also define the $n \times (n + M)$ array $c\_ins[\,j_1\,][\,j_2\,]$ that stores the change in cost when inserting job j_1 before job j_2 in the current solution s. Finally, we also introduce the $n \times (n + M)$ array $infeas[\,j_1\,][\,j_2\,]$ to indicate whether the move $\nu = (j_1, j_2)$ is infeasible by holding the very large value *INVALID* or 0, respectively. Figure 4 illustrates the effect of an insert move in more detail.

For a given solution s, the deletion and insertion cost of the move $\nu = (j_1, j_2)$ can be calculated as shown in Pseudo-code 3.

Pseudo-code 3 Computation of the cost of an insert move

$$\begin{aligned}
(m_1, k_1) &= ins_coord(s, j_1) \\
(m_2, k_2) &= ins_coord(s, j_2) \\
j_1 prev &= s(m_1, k_1 - 1) \\
j_1 next &= s(m_1, k_1 + 1) \\
j_2 prev &= s(m_2, k_2 - 1)
\end{aligned}$$

$$\left.\begin{aligned} c\_del[\,j_1\,] = &- jobcost(m_1, j_1 prev, j_1) \\ &- jobcost(m_1, j_1, j_1 next) \\ &+ jobcost(m_1, j_1 prev, j_1 next) \end{aligned}\right\} \begin{array}{l} \text{subtract cost of deleted job } j_1 \\ \text{change in cost of job } j_1 next \end{array}$$

$$\left.\begin{aligned} c\_ins[\,j_1\,][\,j_2\,] = &+ jobcost(m_2, j_2 prev, j_1) \\ &- jobcost(m_2, j_2 prev, j_2) \\ &+ jobcost(m_2, j_1, j_2) \end{aligned}\right\} \begin{array}{l} \text{add cost of inserted job } j_1 \\ \text{change in cost of job } j_2 \end{array}$$

To calculate the array *infeas*, we assess that a move $v = (j_1, j_2)$ can become infeasible if j_1 "should not be put" on machine m_2 which can be the case for three different reasons. First, if $m_1 = m_2$, then we do not want to consider this move at all. Second, if $n_{m_2} = n_{\max}$, then we do not want to put another (any other) job on m_2. Third, it might not be possible to produce job j_1 on machine m_2 due to its job characteristics. Assume that initially, all entries of *infeas* have been set to 0. We can initialize the array according to Pseudo-code 4.

Pseudo-code 4 Initializing the infeasibility array

```
FOR j1 = 1 to n DO
    (m1, k1) = ins_coord(s, j)
    FOR m2 = 1 to M DO
        IF ((m1==m2) OR (nm2==nmax) OR (j1 cannot be produced on m2)) DO
            FOR k2 = 1 to nm2 + 1 DO
                infeas[ j1 ][ s(m2, k2) ] = INVALID
            END (FOR)
        END (IF)
    END (FOR)
END (FOR)
```

The update

Assume that we have chosen the move $v = (j_1, j_2)$ in a given iteration with current solution s. We now have to update *c_ins*, *c_del*, and *infeas*, as well as n_m, the number of jobs on a machine m, and of course the current solution s. As can be seen from Fig. 3, *c_del* has to be updated along the lines of Pseudo-code 3 for the following entries.

update $c\_del[\, j_1 prev\,]$, $c\_del[\, j_1\,]$, $c\_del[\, j_1 next\,]$, $c\_del[\, j_2 prev\,]$, $c\_del[\, j_2\,]$.

In Fig. 3 we detect that inserting a job before $j_1 next$, j_1, or j_2 has altered cost. We therefore, perform the following update according to Pseudo-code 3.

```
FOR j = 1 to n DO
    update c_ins[ j ][ j1 ], c_ins[ j ][ j1next ], c_ins[ j ][ j2 ]
END (FOR)
```

Lastly, we also need to update

$$n_{m_1} = n_{m_1} - 1$$
$$n_{m_2} = n_{m_2} + 1$$

Clearly, which moves are infeasible has also changed, so that *infeas* needs to be re-initialized (or updated, which entails a rather lengthy description which is why we skip it here).

Algorithm

Naturally, in each iteration we choose the move $v = (j_1, j_2)$ such that the corresponding cost of the move is minimized (over all (j_1, j_2) with $j_1 = 1, \ldots, n$, $j_2 = 1, \ldots, n + M$). For the algorithm we replace the corresponding lines of Algorithm 2 by the following.

3′ FOR all insert moves $v = (j_1, j_2)$ DO
5′ $f' = f + c\_del[j_1] + c\_ins[j_1][j_2] + infeas[j_1][j_2]$

Note that in this version, the arrays *ins_tabu*, *c_del*, *c_ins* and *infeas* need initialized and updated in line 2 and line 14 of Algorithm 2, respectively.

Computational cost

The update of insert move costs as described above requires $9n + 15$ function calls to *jobcost*, yielding again a computational effort of $O(n)$ in our measure of complexity.

Results

The introduction of insert moves brought an improvement of 8.58% in the objective function compared to the initial solution. This is a step up of almost 2 percentage points with respect to the previous version $ts3$. We note that similar to the versions with swap moves and no diversification strategy, only real improvements are found in the first 89 iterations before the search switches to zero-cost moves in the remaining steps.

4.5 ts5—One neighborhood is not enough

Since the introduction of insert moves brought a clear improvement in terms of the best solution found, we will keep this neighborhood in the following versions. To diversify the search further, we combine the use of both neighborhoods in $ts5$, at first without adapting both concepts to each other. The search simply alternates between 100 iterations with swap moves and 100 iterations with insert moves.

Tabu strategy

We implement the tabu strategy from Sect. 4.2 for swap moves and the tabu strategy from Sect. 4.4 for insert moves. In this version, the two strategies still ignore each other—the array *swap_tabu* is only updated after swap moves and the array *ins_tabu* only after insert moves.

Diversification

We use the diversification strategy of $ts3$ from Sect. 4.3 for swap moves. For the mean time, we neglect the fact that the search also picks solutions from a different neighborhood.

Algorithm

Basically, the algorithms of $ts3$ and $ts4$ are run for loops of 100 iterations each for a total of $N = 1{,}000$ iterations. At the end of each loop, the current solution is passed as initial solution to the next loop.

Computational cost

Additionally to the $O(n)$ function calls of *jobcost* per iteration, every 100 iterations we need to re-initialize *swap_cost* or *ins_cost* which entails a cost of $O(n^2)$.

Results

Combining both neighborhoods brought an additional improvement of 0.2 percentage points compared to the previous version. We also observe that the best solution was found later on in the search, although we also see that besides real improvements, only zero-cost moves where chosen. Since this version only serves as an intermediate step, we do not comment further on the results.

4.6 ts6—No zero moves

$ts6$ uses the neighborhood structures and tabu strategies from $ts5$. Additionally, we implement a supplementary tabu strategy for an intensification phase, on which also a new diversification strategy for swap moves is based. Moreover, we finally address the issue of zero-cost moves.

Intensification

Up to now, the tabu strategy held off jobs from certain positions, so that the search was kept away from particular areas of the search space and therefore, *diversified*. The following idea of a supplemental tabu strategy should help to *intensify* in certain regions of S. We use the $n \times n$ array $tabu\_sup[\,j\,][\,p\,]$ that indicates the iteration in which job j was assigned to position p where it is supposed to remain for at least $T_{\text{sup}} = 2$ iterations. We only activate *tabu_sup* after $T_{\text{start}} = 2/3 \cdot N$ iterations.

Diversification

So far, the diversification strategy for swap moves encouraged the search to choose moves whose component jobs have rarely been exchanged before. The new strategy is based on the idea that new good solutions can be found by encouraging the search to put jobs on positions where they have rarely been before. We employ the $n \times n$ $res\_freq[\,j\,][\,p\,]$ which stores the number of iteration job j has been on position p to keep track of the *residence frequency*. With the use of *tabu_sup*, the update of *res_ freq* is, in fact, very simple. If, in iteration i, we choose a move through which job j leaves

position p, we simply set

$$res\_freq[j][p] = res\_freq[j][p] + i - tabu\_sup[j][p].$$

After a large number of iterations, we do not want *res_freq* to distinguish between jobs whose residence frequency only differs by a couple of iterations. We therefore, introduce *penalty levels* P_{lev}. When we update *res_freq*, we therefore, also set

$$res\_freq[j][p] = P_{\text{lev}} * (res\_freq[j][p]/P_{\text{lev}}),$$

where "/ " denotes integer division. P_{lev} is set to 2. The penalty terms are applied the same way as in Sect. 4.3 by replacing the cost of a swap move $\mu = (p, q)$ with component jobs j_1 and j_2 by

$$swap\_cost[p][q] + P_{\text{fac}} * (res\_freq[j_1][p] + res\_freq[j_2][q])$$

Since the search typically has to walk across hills and plateaus to find better solutions, we keep the terms deactivated after an improving move for an additional number of $P_{\text{gap}} = 2$ admissible non-improving moves.

Prohibiting zero-cost moves

To address the issue of too many zero-cost moves, we simply prohibit them in the following way. Not more than $C_{\max} = 5$ consecutive zero cost moves are allowed. Moreover, to keep the search from repeatedly choosing $C_{\max}$ zero-cost moves in succession and one move with cost not equal to zero in between, we allow for a total of $C_{\text{tot}} = N/2$ zero-cost moves only, after which no such moves are admitted at all anymore.

Algorithm

As in $ts5$, the two neighborhood concepts still run independently of each other with the exception of sharing the strategy for banning zero-cost moves as described above. The implementation of $ts6$ is therefore, analogous as for $ts5$, with the following adaption incorporating $ts3$ as a basis for swap moves. *res_ freq* is employed instead of *pen_rec*, and additionally *sup_tabu* needs to be initialized and updated. Moreover, the supplemental tabu strategy has to be included when checking the tabu status of a swap move. Obviously, for the zero-cost move strategy, two variables need to be maintained during the search, one for the current consecutive number of zero-cost moves and one for the total number of such moves.

Results

$ts6$ was able to find slightly better solutions than $ts5$. It is noticeable that the number of real improvements increased for swap moves, but remained the same for insert moves

compared to the previous version. Clearly, this fact suggests a diversification strategy for insert moves which is implemented in the following version.

4.7 ts7—Advanced diversification

In the last version of the program, we want to better adapt the succession of swap and insert move, where we finally implement a tabu and diversification strategy that can be employed for both types of moves.

Neighborhood

We switch between the two types of moves not after a fixed number of iterations, but alter between them dynamically depending what kind of solutions could be found in a particular type of neighborhood. More concretely, we stay with a type of move for at least N_{min} iterations. After that, we switch to the other type of move once the gap between current iteration and the iteration with the last real improvement has become greater than N_{gap}. The admissible size of the gaps increases throughout the search by setting it to $N_{\text{gap}} = N_{\text{fac}} * i$ where i denotes the current iteration and $0 < N_{\text{fac}} < 1$. We used $N_{\text{fac}} = 0.2$.

Zero-cost moves and intensification

We use *tabu_sup* and the prohibition of zero-cost moves in the same way as in the previous version.

Diversification

We wish to encourage the search to assign jobs to positions where they can contribute to yield better solutions. Aside from the type of the job itself, the machine and the job assigned to the previous slot are crucial for the cost of a job. We use the $M \times n \times n$ array $freq[m][j\_prev][j]$ to store how many iterations job j has spent on machine m after job j_prev. The update works analogously to the update of *res_ freq* using *ins_tabu*, as well as the way the resulting penalty terms are utilized.

Algorithm

We give the skeleton of the algorithm alternating between the neighborhoods in Pseudo-code 5. The individual components for each neighborhood can be gleaned from previous versions.

Computational cost

As in $ts5$ and $ts6$, we have an additional computational cost of $O(n^2)$ each time there is a switch between neighborhoods. Similar to the strategy in $ts5$ and $ts6$, we can control this additional cost by setting the parameter $N_{\text{min}} = N_{\text{prop}} * N$ (which yields

Pseudo-code 5 Dynamic switch between neighborhoods

```
i = 0
WHILE (i ≤ N) DO
    initialize relevant arrays for insert moves
    DO
        pick a new neighbor (through insert move)
        update relevant arrays for insert moves
        i = i + 1
    WHILE (check whether to stay with insert moves) END(DO)
    initialize relevant arrays for swap moves
    DO
        pick a new neighbor (through swap move)
        update relevant arrays for swap moves
        i = i + 1
    WHILE (check whether to stay with swap moves) END(DO)
END(WHILE)
```

a number of a total $1/N_{\text{prop}} - 1$ change-overs) and therefore, keep the overall cost at $O(n)$ function call to *jobcost* per iteration. We set $N_{\text{prop}} = 0.1$.

Results

The final version $ts7$ brought an improvement of 9% compared to the initial solution, which is an enhancement of 0.11 percentage points to the previous version $ts6$. For the first time, most real improvements were found during insert moves, even though almost 80% of all solutions were found in swap move neighborhoods. This suggests that the diversification strategy for insert moves works very well. On the other hand, the penalty terms were not utilized during swap moves, which indicates that the criterion of when to activate the diversification strategy should adapted to each type of neighborhood separately. The switch between neighborhoods occurred 4 times.

4.8 Comparison of all versions

All results from the previous sections are summarized in Table 1, details of the search process are listed in Table 2. The values for the different parameters can be found in Table 3.

5 Conclusions

We applied a tabu search algorithm to the concrete scheduling problem of a company producing cables for cars after modeling it as a combinatorial optimization problem. We implemented and examined various features of tabu search adapted to our specific problem to improve the results which were listed in different program versions throughout Sect. 4. Program $ts7$ is the best version of our algorithms.

Although we were dealing with assigning cable jobs to machines as an underlying application, our considerations for the fast update of f presented in Sect. 4.2 and

Sect. 4.4 can be useful for any local search problem where the objective function has the same structure as (2) in Sect. 2.

A limitation of our work clearly lies in testing our ideas on only one problem instance.

Useful elements

The most successful implementation compared to the basic version was the introduction of an additional neighborhood defined by insert moves, as presented in version $ts4$. The switch between swap and insert moves in $ts5$, eventually employed dynamically in $ts7$ also brought improvements. Other profitable features were the banning of zero-cost moves and the intensification using the supplemental tabu strategy introduced in $ts6$. The diversification based on residence frequency records in $ts7$ could directly account for the structure of the objective function and proved to be a valuable adaptation.

And not so useful elements

Of little help was the diversification strategy from $ts3$ that penalized swap moves whose component jobs have been moved often before. This information seems to have little association with the solutions actually visited during the search. Moreover, adding penalty levels and keeping the terms deactivated for several iterations after an improving move, as done in $ts6$, had no influence on the results.

Possible extensions and outlook

Certainly, there is room left for various improvements of the implementations from Sect. 4. A first step could be devising an intensification strategy that can be employed for both swap and insert moves which takes into account the structure of the objective function (in a similar way as the common diversification strategy in $ts7$). Another means of intensification could be a solving M individual TSP problems on each machine in parallel, carried out every few iterations. An implementation of a fast update of *ins_cost* and *swap_cost* after "the other" type of move could facilitate an arbitrarily dynamic switch between neighborhoods. Also, a stopping criterion based on the gaps between real improvements rather than a fixed number of iteration would be an interesting extension.

In terms of the model, a uniform work load of the machine (which is desired in this context) could be better taken care of by aiming to put equal total length, not total number of jobs on each machine.

Clearly, to obtain robust results of the search, more comprehensive testing on different instances will be necessary.

Acknowledgments The author would like to acknowledge the help and guidance of her master's thesis advisor Arnold Neumaier on the underlying research of this paper, as well as valuable input from Fred Glover, Manuel Laguna, and an anonymous referee.

References

Aarts E, Lenstra JK (eds) (1997) Local search methods in combinatorial optimization. Wiley, Chichester, New York

Abdullah S, Burke EK, McCollum B (2007) Using a randomised iterative improvement algorithm with composite neighbourhood structures for the university course timetabling problem. In: Doerner KF, Gendreau M, Greistorfer P, Gutjahr WJ, Hartl RF, Reimann M (eds) Metaheuristics—progress in complex systems optimization. Springer, New York, pp 153–169

Blum C, Roli A (2003) Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput Surv 35:268–308

Brucker P (2007) Scheduling algorithms, 5th edn. Springer, Berlin

Glover F (1977) Heuristics for integer programming using surrogate constraints. Decis Sci 8:156–166

Glover F (1989) Tabu search—part I. INFORMS J Comput 1:190–206

Glover F (1990) Tabu search—part II. INFORMS J Comput 2:4–32

Glover F, Laguna M (1997) Tabu search. Kluwer, Boston

Grabowski J, Wodecki M (2004) A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion. Comput Oper Res 31:1891–1909

Graham RL, Lawler EL, Lenstra JK, Rinnooy Kan AHG (1979) Optimization and approximation in deterministic sequencing and scheduling: a survey. Ann Discrete Math 5:287–326

Leung J (2004) Handbook of scheduling: algorithms, models, and performance analysis. CRC Press, Boca Raton

Nowicki E, Smutnicki C (1996) A fast taboo search algorithm for the job shop scheduling problem. Manage Sci 42:797–813

Reeves CR, Beasley JE (1993) Introduction. In: Reeves CR, Beasley JE (eds) Modern heuristic techniques for combinatorial problems. Wiley, New York, pp 1–19

Tan KC, Burke E, Lee TH (2007) Feature cluster: evolutionary and meta-heuristic scheduling (guest editorial). Eur J Oper Res 177:1852–2118