OR Spectrum 29:513–533 (2007)
DOI 10.1007/s00291-006-0052-5

REGULAR ARTICLE

Tobias Brueggemann · Johann L. Hurink

Two very large-scale neighborhoods for single machine scheduling

Published online: 3 June 2006

Abstract In this paper, the problem of minimizing the total completion time on a single machine with the presence of release dates is studied. We introduce two different approaches leading to very large-scale neighborhoods in which the best improving neighbor can be determined in polynomial time. Furthermore, computational results are presented to get insight in the performance of the developed neighborhoods.

Keywords Very large-scale neighborhoods · Local search · Single machine

MSC Classification 90B35 · 68M20

1 Introduction

Many optimization problems in the practical world are computationally intractable. For these problems, it costs too much time to solve them to optimality. Hence, there is need for practical approaches to solve such problems. A way to achieve this is the development of heuristic (approximation) algorithms that are able to find satisfactory solutions within a reasonable amount of computation time. In the literature concerning heuristic algorithms, two different classes can be distinguished. The first class of heuristic algorithms consists of constructive algorithms. These algorithms build solutions by assigning values to one or more decision variables at a time. The second class is the improvement algorithms that start with a feasible solution and iteratively try to advance to a better solution. In this class, local search and neighborhood search algorithms play a crucial role.

A local search heuristic starts, roughly spoken, with some solution and iteratively replaces the current solution by some solution in a neighborhood of this solution. Thus, for a local search approach, a method for calculating an initial

T. Brueggemann (✉) · J. L. Hurink
Department of Applied Mathematics, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail: t.brueggemann@math.utwente.nl, j.l.hurink@math.utwente.nl

solution, a neighborhood structure of a given solution and a method to select a solution from the neighborhood of a given solution are needed.

The neighborhood structure of a given solution has an important influence on the efficiency of the local search heuristic. It determines the navigation through the solution space during the iterations of the local search method, and the computation time of one iteration is affected by the choice of the neighborhood structure as well. Therefore, one expects that the size of the neighborhood has influence on the quality of the final solution of a local search approach, because a larger neighborhood covers more solutions and, of course, affects the running time. Therefore, a compromise between size, quality, and running time has to be found.

A possible way to do this is to restrict the neighborhood of a solution to promising solutions, i.e., to solutions which may have a good objective value. Another possibility is to develop efficient methods to find the best solution in a given neighborhood, which is often an interesting optimization problem itself.

Over the last time, very large-scale neighborhoods that can be exhausted in reasonable time were considered. These very large-scale neighborhoods mostly contain an exponential number of solutions but allow a polynomial exploration. A nice survey about very large-scale neighborhood techniques is given by Ahuja et al. (2002). They categorize those into three not necessarily distinct classes. Their first category of neighborhood search algorithms consists of variable-depth methods. These algorithms partially exploit exponential-sized neighborhoods using heuristics. The second category consists of improvement algorithms based on network flow. These methods use network flow techniques to identify improving neighbors. Finally, their third category consists of neighborhoods for $\mathcal{NP}$ -hard problems obtained by considering subclasses or restrictions that can be solved in polynomial time.

Although the concept of very large-scale neighborhoods sounds promising, the practical relevance of these neighborhoods is not so clear (see, e.g., Hurink 1999). In this paper, we develop two very large-scale neighborhoods for a single-machine scheduling problem and study their usage in local search. The goal is to present for one problem different concepts to reach very large-scale neighborhoods and to get more insight under which conditions of very large-scale neighborhoods may be of practical use.

To be more precise, we present two different approaches for obtaining very large-scale neighborhoods for the problem of scheduling n jobs with release dates r_j and processing times p_j on a single machine to minimize total completion time $\sum C_j$ without preemption. In the classical scheduling notation by Graham et al. (1979), this problem is denoted by $1|r_j|\sum C_j$. The considered problem is $\mathcal{NP}$ -hard in the strong sense as stated in Lenstra et al. (1977). Since for a fixed sequence π there is an efficient method for calculating the best schedule in $\mathcal{O}(n)$, local search may be applied by considering sequences as solutions.

The first neighborhood we present is an extension of the adjacent pairwise interchange neighborhood (API). This extension is based on the idea of combining independent operators. Congram et al. (2002) and Potts and van de Velde (1995) applied the idea of combining independent SWAP operators to the single-machine total weighted tardiness scheduling problem and the traveling salesman problem, respectively. They call their approach *iterated dynasearch*. In the paper of Congram et al. (2002), the authors show that the size of their neighborhood is $\mathcal{O}(2^{n-1})$ and they

give a dynamic programming recursion to find the best neighbor in $\mathcal{O}(n^3)$. Hurink (1999) applies compounded API operators in the context of single-machine batching problems, and he shows that an improving neighbor can be obtained in $\mathcal{O}(n^2)$ by calculating the shortest path in an improvement graph, that is a structure, defined by the possibility of combining API operators and their change of the objective value.

For problem $1|r_j|\sum C_j$, the independency of changes gets a bit more complicated due to the presence of release dates. Therefore, we first examine in which situations we may combine several API operators to modify a sequence π describing a solution for $1|r_j|\sum C_j$. We exploit the locality of API operators, indicating that such an operator causes only small changes to a schedule. It turns out, that the problem of finding a best-combined move to a neighboring solution can be solved by calculating the shortest path in an improvement graph, as described by Hurink (1999). According to Ahuja et al. (2002), this extension of the API neighborhood belongs to their second category of very large-scale neighborhoods.

The second neighborhood we introduce is based on a dominance rule that may also be used in a branch-and-bound algorithm for solving the considered problem. This dominance rule states that for a given solution, the problem is locally not very different from its relaxation $1||\sum C_j$, which can be solved by the *shortest processing time first* (SPT) rule from Smith (1956). This second neighborhood belongs to the third category of Ahuja et al. (2002).

The outline of this paper is as follows. In Section 2, we give a brief description of the problem and introduce some notations. Section 3 describes the two very large-scale neighborhoods and their main conceptual differences. In Section 4, we give some computational results for these neighborhoods and discuss the possibilities and limitations of the two concepts. Finally, some concluding remarks are given.

2 Problem description

We consider the single machine scheduling problem, where n jobs $1,\ldots,n$ with nonnegative release dates $r_1,\ldots,r_n$ and processing times $p_1,\ldots,p_n$ are given. A job j is not available before time r_j and needs to be processed for p_j time-units without preemption. Without loss of generality, we reorder the jobs such that $r_1 \leq \ldots \leq r_n$ and, if $r_j = r_{j+1}$, that $p_j \leq p_{j+1}$.

A schedule for this problem can be described by a vector $S = (S_1,\ldots,S_n)$ of starting times. It is called a *feasible schedule*, if and only if:

- $S_j \geq r_j$ for $j = 1,\ldots,n$,
- Either $S_j \geq S_i + p_i$ or $S_i \geq S_j + p_j$ for all pairs $i,j = 1,\ldots,n$ with $i \neq j$.

Furthermore, by C, we denote the vector of completion times for a feasible schedule S, i.e., $C_j := S_j + p_j$ for $j = 1, \ldots, n$. The goal is to find a feasible schedule S, such that the objective function

$$f(S) := \sum_{j=1}^{n} C_j$$

is minimized.

A solution of this problem can be characterized by a sequence of the jobs, which represents a processing order of the jobs. For a given sequence π, the corresponding feasible schedule S is given by:

$$\begin{aligned} &S_{\pi(1)} := r_{\pi(1)} \text{ and} \\ &S_{\pi(j)} := \max\{r_{\pi(j)}, S_{\pi(j-1)} + p_{\pi(j-1)}\} \text{ for } j = 2, \ldots, n. \end{aligned} \tag{1}$$

The calculation of this schedule needs $\mathcal{O}(n)$ time. From now on, let $\pi = (\pi(1), \ldots, \pi(n))$ be a given sequence and S^π the corresponding feasible schedule. Often, we omit π if it is clear which sequence is considered.

3 Very large-scale neighborhoods

In this section, we present two very large-scale neighborhoods for the problem $1|r_j|\sum C_j$. The neighborhoods rely on two different principles. First, in Section 3.1, we build up neighbored solutions by combining several independent pair-interchange operators to one compounded neighborhood operator. Next, in Section 3.2, we use a reordering of subsequences as the base of building up a neighborhood structure. Both neighborhoods can have an exponential (in n) number of neighbors and allow efficient exploration. Finally, in Section 3.3, we compare the two approaches.

3.1 Compounded API

In this subsection, we develop a neighborhood, which is based on adjacent-pair interchanges (API). First, we analyze the effects of a single API operator. We examine which jobs are affected and how the objective function is influenced. Later on, this will be used to combine several API operators to a compounded operator, which results in a neighborhood that may have exponential size and can be searched in polynomial time. Before presenting the mentioned results, we first introduce some notations.

If we consider a schedule S^π, this schedule decomposes uniquely in a set of so-called blocks. Hereby, a block consists of jobs that are scheduled without idle-times, such that the first job of its block starts after an idle period at its release date and all other jobs start at the completion time of their predecessor. We denote by $b(\pi)$ the

number of blocks and by $B_1, \ldots, B_{b(\pi)}$ the blocks of the form $B_\beta = \{\pi(i_\beta), \ldots, \pi(i_\beta + k_\beta)\}$ with $i_\beta + k_\beta + 1 = i_{\beta+1}$.

In addition, we denote by g_β the amount of idle-time between the jobs $\pi(i_\beta)$ and $\pi(i_{\beta-1} + k_{\beta-1})$, if $\beta \geq 2$, or the idle-time before the job $\pi(1)$, if $\beta = 1$. In Fig. 1 an example for blocks and their gaps in a schedule is given.

The neighborhood $\mathcal{N}_{API}$ of a sequence π consists of all sequences received by applying one of the adjacent-pair interchange operators $API_1, \ldots, API_{n-1}$, where the operator API_j interchanges the elements in positions j and $j+1$ of a sequence, i.e.,

$$API_j(\pi) := (\pi(1), \ldots, \pi(j-1), \pi(j+1), \pi(j), \pi(j+2), \ldots, \pi(n)).$$

In the following, we examine how a single API operator affects a given solution. Considering the two jobs $\pi(j)$ and $\pi(j+1)$ involved in API_j, there are different cases to handle, depending on the position of job $\pi(j)$ in the block and the release date $r_{\pi(j+1)}$ of job $\pi(j+1)$.

Consider an operator API_j where index j belongs to a position of block $B_\beta = \{\pi(i_\beta), \ldots, \pi(i_\beta + k_\beta)\}$. Clearly, if $j = i_\beta + k_\beta$, the application of API_j leads to an increase of the objective value, since the job $\pi(i_\beta + k_\beta + 1)$ is the first job of the next block and, therefore, starts at its release date. Thus, if we are interested in operators API_j, which may lead to better solutions, we only have to consider jobs j which are not the last job of a block. To calculate the consequences of the exchange of the jobs $\pi(j)$ and $\pi(j+1)$, let S' be the schedule corresponding to $API_j(\pi)$. Furthermore, let

1. $d_1 := S'_{\pi(j)} - S_{\pi(j)}$.
2. $d_2 := S_{\pi(j+1)} - S'_{\pi(j+1)}$.
3. $d_3 := C'_{\pi(j)} - C_{\pi(j+1)}$.

Herewith, d_1 describes the absolute value of the change in starting time of job $\pi(j+1)$, d_2 gives the change for job $\pi(j)$, and d_3 presents the effect of the exchange for the succeeding jobs $\pi(j+2), \ldots, \pi(n)$ (see Fig. 2). The value d_2 is given by

$$d_2 = \begin{cases} \min\{p_{\pi(j)}, S_{\pi(j+1)} - r_{\pi(j+1)}\} & \text{for } j \neq i_\beta \text{ and} \\ \min\{p_{\pi(j)} + g_\beta, S_{\pi(j+1)} - r_{\pi(j+1)}\} & \text{for } j = i_\beta. \end{cases}$$

Furthermore, by scheduling job $\pi(j)$ directly after the finishing of job $\pi(j+1)$, the starting time $S'_{\pi(j)}$ of job $\pi(j)$ becomes $S'_{\pi(j)} = S_{\pi(j)} + p_{\pi(j)} - d_2 + p_{\pi(j+1)}$. Taking into account the release date of job $\pi(j)$, this leads to

$$d_1 = \max\{p_{\pi(j)} + p_{\pi(j+1)} - d_2, r_{\pi(j)} - S_{\pi(j)}\}.$$

Based on these considerations, d_3 becomes:

$$d_3 = d_1 - p_{\pi(j+1)}.$$

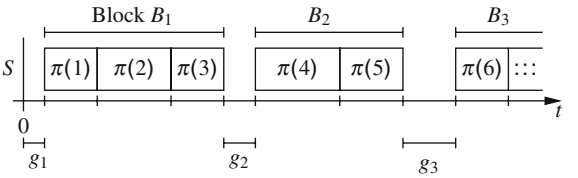

Fig. 1 Blocks and gaps in a schedule S

If $j > i_\beta$, d_3 is always greater or equal to 0. However, if $j = i_\beta$, d_3 may also be negative due to the gap g_β before job $\pi(i_\beta)$. The parameter d_3 is useful to calculate the effects of API_j on the jobs $\pi(j+2), \ldots, \pi(n)$.

Applying operator API_j changes the objective value by

$$\begin{aligned} \delta_j := f(S') - f(S) &= \sum_{\mu=1}^{n} (S'_{\pi(\mu)} + p_{\pi(\mu)} - S_{\pi(\mu)} - p_{\pi(\mu)}) \\ &= d_1 - d_2 + \sum_{\mu=j+2}^{n} S'_{\pi(\mu)} - S_{\pi(\mu)}. \end{aligned} \tag{2}$$

To calculate δ_j, it remains to calculate

$$penalty := \sum_{\mu=j+2}^{n} S'_{\pi(\mu)} - S_{\pi(\mu)}.$$

As mentioned before, the effects of API_j on the jobs $\pi(j+2), \ldots, \pi(n)$ are characterized by d_3. If $d_3 = 0$, then $penalty := 0$. Otherwise, not only jobs of B_β but also jobs of subsequent blocks may be involved. If $d_3 > 0$, all jobs $\pi(\mu)$ with $j+2 \le \mu \le i_\beta + k_\beta$ have to be shifted by d_3 units to keep a feasible schedule. This results in $penalty := d_3(i_\beta + k_\beta - j - 1)$. If $d_3 > g_{\beta+1}$, the shift also affects the next block. Thus, we have to shift all jobs $\pi(i_{\beta+1}), \ldots, \pi(i_{\beta+1} + k_{\beta+1})$ of the next block by $d_3 := d_3 - g_{\beta+1}$, resulting in $penalty := penalty + d_3 k_\beta$. Such a shifting of complete blocks has to be repeated until the gap is sufficiently big.

On the other hand if $d_3 < 0$, we have to shift jobs to the left in the new schedule, beginning with job $\pi(j+2)$. The involved jobs are only the jobs $j+2, \ldots, i_\beta + k_\beta$, and the calculation of $penalty$ has to be done by calculating the new starting times

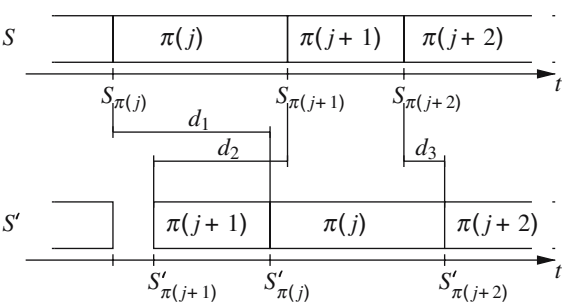

Fig. 2 Effect of $API_j(\pi)$

for all these jobs by applying Eq. 1. Note that in this case, the *penalty* is negative and the block B_β may split into several new blocks.

All in all, the effects of such an API operator are computable in time $\mathcal{O}(n)$; however, in the average, we expect a much lower running time. The neighborhood $\mathcal{N}_{API}$ has size $\mathcal{O}(n)$. Hence, we need in the worst case $\mathcal{O}(n^2)$ to compute the best solution in $\mathcal{N}_{API}$.

In the following, we investigate the possibility of combining API operators. More precisely, we search for pairs of operators API_i and API_j, where the consecutive application of these two operators to a sequence π leads to a change of $\delta_i + \delta_j$ in the objective value. Such an independence of operators allows to evaluate the effects of a combined execution of the operators, based on the effects of the single operators and, therefore, allows a combined execution of several different APIs in one iteration of the local search algorithm leading to a compounded neighborhood. To find candidates for combined operators, we have to analyze which APIs do not have an effect on each other.

Consider indices i and j with $1 \leq i < j \leq n-1$ and $i+2 \leq j$. We are interested in those cases where the effect of $API_j(API_i(\pi))$ and $API_i(API_j(\pi))$ are equal to $\delta_i + \delta_j$, i.e., we look for indices i and j where

$$f(S^\pi) + \delta_i + \delta_j = f(S^{API_j(API_i(\pi))}).$$

A sufficient condition for this independency is that after applying API_i to π, the resulting schedule around the jobs in position j and $j+1$ must be the same as in π. To formalize this, we introduce a variable F_i denoting the first position after $i+1$ where the application of API_i to π has no effect.

If by applying $API_i(\pi)$ the interval which is occupied by the jobs $\pi(i)$ and $\pi(i+1)$ does not change, we have $F_i = i+2$. On the other hand, if the starting time of job $\pi(i+2)$ or the idle period before $\pi(i+2)$ is changed by applying API_i to π, we determine the index of the last affected job

$$L_i := \max\{k : i \leq k \leq n, S^\pi_{\pi(k)} \neq S^{API_i(\pi)}_{\pi(k)}\},$$

and we define $F_i := L_i + 2$ (we have to add 2 , since there might be a change in the amount of idle time before $\pi(L_i+1)$ after applying API_i). The value for L_i can easily be determined during the calculations of *penalty* .

Based on the above considerations, we call API_j *π -independent of* API_i if

1. Neither $\pi(i)$ nor $\pi(j)$ is a last job of its block.
2. $j \geq F_i$.

In addition, we call API_i and API_j π -independent if either API_j is π -independent of API_i or API_i is π -independent of API_j.

Summarizing, for two π -independent operators API_i and API_j, we have

$$f(S^\pi) + \delta_i + \delta_j = f(S^{API_j(API_i(\pi))}).$$

An important property of π -independency is its *transitivity*. If for $i < j < k$ API_j is π -independent of API_i and API_k is π -independent of API_j, then API_k is also π -

independent of API_i. Thus, we call a set $M \subseteq \{1, \ldots, n-1\}$ π-independent if the API operators belonging to the elements of M are pairwise π-independent.

Hence, for a given π-independent set $M := \{v_1, \ldots, v_k\}$, we can calculate the objective value of the schedule $API_{v_k} \circ API_{v_{k-1}} \circ \ldots \circ API_{v_1}(\pi)$ by

$$f(S^{API_{v_k} \circ API_{v_{k-1}} \circ \ldots \circ API_{v_1}(\pi)}) = f(S^{\pi}) + \sum_{i \in M} \delta_i.$$

We define the neighborhood $\mathcal{N}_{CAPI}$ (CAPI is short for *compounded adjacent-pair interchange*) of a sequence π to consist of all possible sequences resulting from applying all possible combinations of π-independent operators to π. This neighborhood at least contains all API-operators, i.e., $\mathcal{N}_{API}(\pi) \subseteq \mathcal{N}_{CAPI}(\pi)$, but there may generally be an exponential number of sequences in $\mathcal{N}_{CAPI}$.

In the following, we develop an efficient method to calculate a set of independent operators that gives the best gain in the objective value over all possible independent operators. For this, we define a structure called improvement graph, which depends strongly on the given sequence π.

For a given sequence π, let $G^{\pi} = (V, A^{\pi})$ be a graph with vertices $V = \{0, 1, \ldots, n-1, *\}$ and a set $A^{\pi} \subseteq V \times V$ of directed arcs, where each arc $(i, j) \in A^{\pi}$ receives a cost c_{ij}. The vertices 0 and $*$ are called source and sink, respectively. The set A^{π} contains the following arcs.

- Arcs $(0, i)$ with cost $c_{0i} = \delta_i$ for all $1 \leq i \leq n-1$ where $\pi(i)$ is not a last job of a block in S^{π}
- Arcs (i, j) with cost $c_{ij} = \delta_j$ for all $1 \leq i < j \leq n-1$ where API_j is π-independent of API_i
- Arcs $(i, *)$ with cost $c_{i*} = 0$ for all $0 \leq i \leq n-1$

An arc leading to a vertex $i \leq n-1$ corresponds to an application of the operator API_i. Furthermore, a directed path $P = (0, v_1, \ldots, v_k, *)$ with $1 \leq v_i \leq n-1$ corresponds to a combined operator $API_{v_k} \circ API_{v_{k-1}} \circ \ldots \circ API_{v_1}(\pi)$ of π-independent operators, and the sum of the costs of the arcs on the path describes the gain to the objective value. Hence, if we have a shortest directed path from the source to the sink, this determines the best possible combined operator of π-independent operators.

Because our graph has no directed cycles, we can use Dijkstra's algorithm to obtain the shortest path in this graph. With this algorithm, we are able to calculate the best possible combined operator of π-independent API_i operators in $\mathcal{O}(n^2)$.

3.2 Neighborhood PAV

In this subsection, we develop a very large-scale neighborhood based on a dominance criteria for the problem $1|r_i|\sum C_i$. The basis of this approach is that each schedule defines in a unique way so-called peaks and valleys. Here, jobs with high indices form the peaks in the sequence and the indices of the jobs between two

peaks form a valley. More precisely, the peaks of a sequence π are a unique set of indices $1 = i_1 < \ldots < i_k < i_{k+1} = n+1$ with

$$\begin{array}{ll} \pi(j) < \pi(i_\mu) & \text{for } j = i_\mu + 1, \ldots, i_{\mu+1} - 1, \\ \pi(i_\mu) < \pi(i_{\mu+1}) & \text{for } \mu = 1, \ldots, k. \end{array} \tag{3}$$

Furthermore, we have $\pi(i_k) = n$.

From now on, let π be a given sequence of jobs and $i_1, \ldots, i_k, i_{k+1}$ be the peaks belonging to π. We define sets $V_\mu := \{\pi(i_\mu + 1), \ldots, \pi(i_{\mu+1} - 1)\}$ for $\mu = 1, \ldots, k$. Observe, that

$$\bigcup_{\mu=1}^{k} V_\mu \cup \{\pi(i_1), \ldots, \pi(i_k)\} = \{1, \ldots, n\}.$$

The sets V_μ are called *valleys* and contain the jobs between the peaks. In the following lemma, we show three important properties for the jobs of a valley in the schedule S^π.

Lemma 1

(P1) In the schedule S^π, the job $\pi(i_\mu)$, together with the jobs of a valley V_μ, are scheduled without idle times, i.e.,

$$S^\pi_{\pi(k)} = C^\pi_{\pi(k-1)} \text{ for } k = i_\mu + 1, \ldots, i_{\mu+1} - 1.$$

(P2) Within the set of all sequences resulting from π by a reordering of the jobs of a valley V_μ, the sequence obtained by reordering these jobs by nondecreasing processing times has minimal objective value.

(P3) Reordering the jobs of $V_\mu \cup \{\pi(i_\mu)\}$ by nondecreasing processing times does not increase the objective value.

Proof

(P1) The properties of the peaks in Eq. 3 imply that for all $l \in \{i_\mu + 1, \ldots, i_{\mu+1} - 1\}$ holds $l < i_\mu$, i.e., $r_{\pi(l)} \leq r_{\pi(i_\mu)}$. This again implies $C_j \geq r_i$ for all $j, i \in V_\mu$ leading to $S^\pi_{\pi(l)} = \max\{r_{\pi(l)}, C^\pi_{\pi(l-1)}\} = C^\pi_{\pi(l-1)}$ for all $l \in \{i_\mu + 1, \ldots, i_{\mu+1} - 1\}$.

(P2) Due to (P1) of the lemma, the jobs of V_μ are scheduled in S^π in the interval

$$I = [S^\pi_{\pi(i_\mu+1)}, S^\pi_{\pi(i_\mu+1)} + \sum_{j \in V_\mu} p_j].$$

Furthermore, we have $r_j \leq S^\pi_{\pi(i_\mu)}$ for all $j \in V_\mu$. Thus, reordering the jobs of V_μ still allows to schedule the jobs of V_μ in I. According to Smith's rule, a reordering to nondecreasing processing times is best possible.

(P3) Due to (P2) of the lemma, we first may reorder the jobs of V_μ by nondecreasing processing times without increasing the objective value. Since

$r_j \leq r_{\pi(i_\mu)}$, interchanging job $\pi(i_\mu)$ with jobs of V_μ, which have a smaller processing time, leads to a decrease of the objective value. □

Consider a given initial sequence π, peaks $i_1, \ldots, i_k, i_{k+1}$, and valleys $V_1, \ldots, V_k$. Based on (P2) of the lemma, it is possible to find a best sequence respecting the peaks and valleys in time $\mathcal{O}(n \log n)$ by simply sorting the jobs of V_μ by increasing processing times. If we allow a change in the peaks and valleys, we might even get a better solution by sorting the whole sets $V_\mu \cup \{\pi(i_\mu)\}$. In the latter case, it may happen that the new sequence does not have the same peaks and valleys as before and, therefore, again may be optimized by resorting the valleys.

In summary, if peaks $i_1, \ldots, i_k, i_{k+1}$, and valleys $V_1, \ldots, V_k$ are given, we can easily find an optimal sequence π that respects these peaks and valleys. However, as the next theorem shows, it is not easy to find an optimal sequence π respecting the given positions $i_1, \ldots, i_k, i_{k+1}$ of the peaks and corresponding jobs $\pi(i_1), \ldots, \pi(i_n)$ without the knowledge of the valleys.

Theorem 2 *Let* $1 = i_1 < \ldots < i_k < i_{k+1} = n + 1$ *be a given set of integers and* $\pi(i_1) < \ldots < \pi(i_k)$ *the jobs to be scheduled on the positions* $i_1, \ldots, i_k$. *The problem of completing the sequence* π *so that the objective function* $\sum C_j$ *is minimized and*

$$\pi(j) < \pi(i_\mu) \text{for } j = i_\mu + 1, \ldots, i_{\mu+1} - 1$$

is $\mathcal{NP}$ *-hard in the strong sense*.

Proof The proof is given in Appendix.

Based on the properties of peaks and valleys given in Lemma 1, a second very large-scale neighborhood called *PAV* (short for *peaks and valleys*) is defined. Given a solution π with corresponding peaks and valleys, we define the neighborhood $\mathcal{N}_{PAV}$ of a sequence π to contain all reorderings of the valleys. Hence, $\mathcal{N}_{PAV}$ may have an exponential size depending on the instance and the sequence. According to (P2) of Lemma 1, the best neighbor in $\mathcal{N}_{PAV}$ can be determined by sorting the jobs of the valleys by nondecreasing processing times in $\mathcal{O}(n \log n)$.

The neighborhood $\mathcal{N}_{PAV}$ is not directly suited for an iterative improvement process. If the best neighbor in $\mathcal{N}_{PAV}$ is determined, we are in a local optimum, i.e., (P2) of Lemma 1 does not yield any further improvement. To circumvent this, (P3) of Lemma 1 can be applied to the best sequence of $\mathcal{N}_{PAV}$ to obtain a different peak and valley structure. The peak in front of a valley has to be inserted in the valley on the basis of its processing time (which can be realized in linear time). This process can be repeated and stops if all peaks have a processing time smaller or equal to the minimum processing time in their valley.

3.3 Comparison of the neighborhoods

The two approaches, $\mathcal{N}_{CAPI}$ together with the block structure and $\mathcal{N}_{PAV}$ with its peaks and valleys, are somehow related. The jobs that belong to a block are processed without idle times and so are the jobs of a valley defined by the peaks. The difference results from the fact that valleys for two adjacent peaks may be processed without idle time in between, whereas two blocks are always separated by an idle time. Therefore, there are at least as many valleys as blocks for a given sequence π, but there may be more.

However, the underlying ideas to develop the very large-scale neighborhoods are quite different. The neighborhood $\mathcal{N}_{CAPI}$ is built upon the simple neighborhood $\mathcal{N}_{API}$ and has, in principle, the same navigation behavior as that neighborhood. The only difference to $\mathcal{N}_{API}$ is that the API operators are not chosen and executed sequentially but in parallel. Thus, from a quality point of view, we may expect from $\mathcal{N}_{CAPI}$ only a better behavior than from $\mathcal{N}_{API}$, if the parallel choice fits better to the problem. From a computational point of view, both neighborhoods have a worst case complexity of $\mathcal{O}(n^2)$ to compute a best neighbor. However, because the chosen operator of $\mathcal{N}_{CAPI}$ may contain several API operators, one may suppose that the total time to reach a certain solution quality may be shorter for algorithms using $\mathcal{N}_{CAPI}$. Both of these aspects are investigated via computational tests, which are reported in the next section.

The neighborhood $\mathcal{N}_{PAV}$ is based on a local priority criteria, which allows the interchange of two jobs under certain conditions (the first job has a larger processing time and, after the interchange, the first scheduled job does not start later). Thus, in principle this neighborhood also relies on $\mathcal{N}_{API}$. However, in contrast to $\mathcal{N}_{CAPI}$, we do not restrict to independent operators but allow a complete reordering of certain subsets of jobs. This may indicate that an algorithm using $\mathcal{N}_{PAV}$ is able to reach (good) local optima in short time. On the other hand, if one has reached a local optima, $\mathcal{N}_{PAV}$ may not be a good choice to navigate further, because within this neighborhood a job is interchanged only with successors with smaller processing times; i.e., we have a monotone behavior. Note, that under $\mathcal{N}_{API}$ also an interchange with a succeeding job with larger processing time may lead to an improving neighbor. Again, computational results have to give insight to these questions.

4 Computational results

In this section, we report on computational experiments to indicate how the two approaches perform regarding solution quality and running time for small and large instances. Furthermore, for smaller instances, we compare the results with optimal solutions found by a branch-and-bound algorithm. We use the branch-and-bound algorithm of Yanai and Fujie (2004) to obtain these exact solutions. As an initial heuristic, we use, besides some simple methods, the *APRTF-heuristic*, which was presented by Chu (1992).

The APRTF-heuristic iteratively extends a partial schedule by

1. Either scheduling a non-planned job having minimal earliest starting time (ties broken by choosing the job with minimal processing time)
2. Or scheduling a non-planned job having minimal sum of earliest starting time and earliest completion time

The second option is chosen if either the release date of the corresponding job is smaller or equal to the earliest starting time of the job corresponding to the first option, or if some dominance criteria is fulfilled. For details, we refer to the work of Chu (1992). The two simple initial heuristics are called *RSORT* and *RND*. The first sorts all jobs by nondecreasing release dates and the second sorts the jobs randomly. APRTF is superior over RSORT regarding solution quality, and RSORT again retrieves better results than RND.

To get some insight in the quality of the obtained solutions, we use a lower bounding scheme *LB* resulting from a relaxation of the problem by allowing preemption. In this case, the problem becomes solvable in time $\mathcal{O}(n \log n)$ by the *shortest remaining processing time* (SRPT) priority rule (see Baker 1974). The optimal solution of the relaxed problem given by the SRPT rule has been shown by Ahmadi and Bagchi (1990) to be a good lower bound for the considered problem regarding running time and quality.

The problem instances are generated as described by Yanai and Fujie (2004) and Chu (1992). The processing times are uniformly randomly chosen between 1 and 100. The release dates are generated between 0 and $\frac{101n\lambda}{2}$, where λ is a parameter indicating somehow the density of the problem and n denotes the number of jobs. The tests done by Yanai and Fujie (2004) show that the hardest instances are received for $0.6 \leq \lambda \leq 1.2$. To get an idea of the influence of λ, we show in Fig. 3 the average optimal values of 100 randomly generated instances with $n = 100$ jobs and different values of λ. With a constant number of jobs, the optimal objective value increases with λ due to the increasing range of release dates.

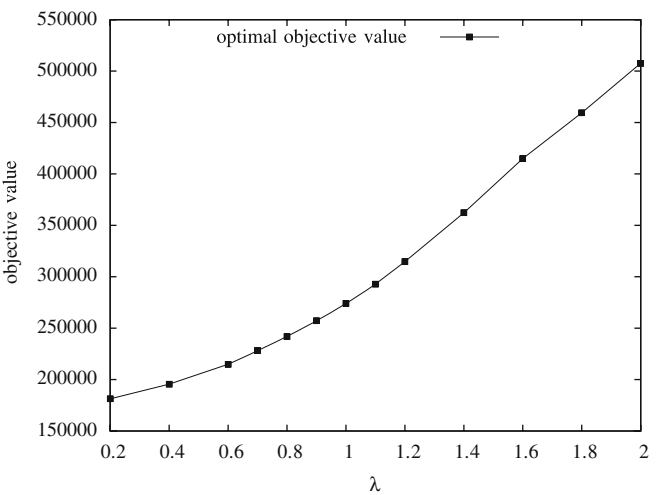

Fig. 3 Average objective values of optimal solutions

To get an indication of the effectiveness of the neighborhoods in practice, we implemented them in ANSI-C with an iterative improvement method. We have chosen not to use tabu search or simulated annealing as we are interested in the structural behavior of the neighborhoods and not in the potentials of local search methods. In the following, we denote with *BAPI* (*best* API), *CAPI*, and *PAV* the iterative improvement process using the neighborhood $\mathcal{N}_{API}$, $\mathcal{N}_{CAPI}$, and $\mathcal{N}_{PAV}$, respectively. Hereby, each method advances in every iteration to the best solution in the corresponding neighborhood as introduced. In addition, because the best solution of the neighborhood $\mathcal{N}_{PAV}$ constitutes a local optimum, we use the prior described method to receive a new peak and valley structure in each iteration of PAV. As a fourth approach, we combine PAV and BAPI. This means, that we apply PAV until we receive a local optimum and afterwards, advance to the best improving neighbor in $\mathcal{N}_{API}$ (i.e., applying one BAPI move). This algorithm we call PAVBAPI. The computational tests were done on a PC with an Intel Pentium IV processor running at 2.4 GHz.

In a first series of tests, we use the APRTF-heuristic as initial solution for the iterative improvement methods. In Fig. 4, the performance of the considered approaches are given, besides the results of some other approaches, which are described further on. For $n = 100$ and different values of λ, the average absolute deviation of the objective values of the solutions to the optimal value for 100 randomly generated instances is given. Since a solution received by the APRTF-heuristic is mostly locally optimal in the neighborhood $\mathcal{N}_{PAV}$, the values obtained by PAV are almost identical in average to the values of APRTF. The figure also shows that BAPI, CAPI, and PAVBAPI perform nearly identical in average. The corresponding solutions are similar in structure, and there is not much difference to the initial solution. This may be caused by the fact that APRTF often delivers local optima or a solution of good quality.

BAPI and CAPI arrive at solutions of similar quality. This indicates that, for problem $1|r_i|\sum C_i$, the possibility of the very large-scale neighborhood $\mathcal{N}_{CAPI}$ to combine several moves and to look at their overall performance does not help for navigation. Because BAPI also delivers solutions of similar quality compared to

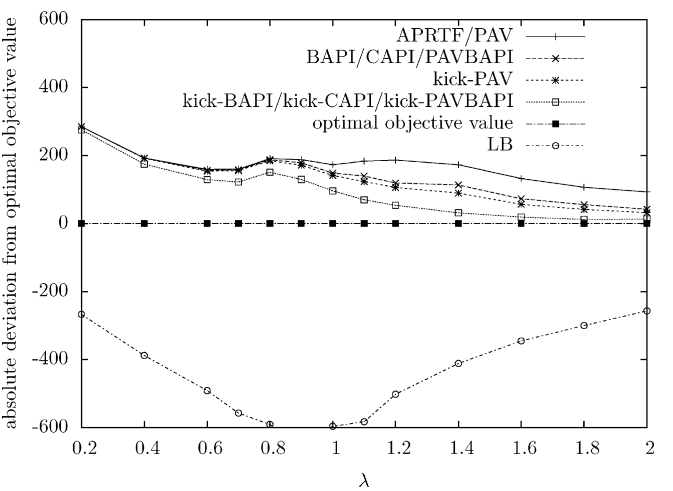

Fig. 4 Iterative improvement using APRTF to obtain initial solutions

PAVBAPI, it seems that mainly BAPI moves are applied in PAVBAPI, if an initial solution of good quality is used.

To get more insight in the navigational behavior of the neighborhoods around solutions of good quality, we added an extra component to the local search approach. After iterative improvement stops, we give the resulting local optima a kick and restart the iterative improvement procedure. This kick simply takes one of the jobs not starting at its release date and reinserts it at a position in the sequence so that this job will start at its release date. By doing so, we slightly perturb the solution and arrive at a different, but not necessarily better, solution compared to the original local optimal solution. Then we again start iterative improvement possibly arriving at a better solution. We apply the kick to every job where it is possible, and the best-received solution from this kick followed by iterative improvement is taken as the next solution. We iterated this process and stopped if no kick to a job followed by iterative improvement leads to a better solution. In the following, we call the corresponding methods *kick-BAPI*, *kick-CAPI*, *kick-PAV*, and *kick-PAVBAPI*.

The average solution quality received by using the kick method is also presented in Fig. 4. Here, one can see that the kick method generally has a big impact on solution quality. Although PAV was hardly able to improve the initial solution given by the APRTF-heuristic, it now improves this solution considerably. The other methods kick-BAPI, kick-CAPI, and kick-PAVBAPI perform nearly the same, but much better than kick-PAV.

The above-mentioned test gives some indications of the navigational behavior of the neighborhoods in regions of high-quality solutions. In a further series of tests, we investigate how they behave if a weak initial solution obtained by RSORT is chosen. In Fig. 5, we compare the initial solutions and local optima with the optimal solution for the generated instances. Due to the nature of PAV, an initial solution received by RSORT already constitutes a local optimum. By looking at $\lambda \geq 0.6$, the methods BAPI and CAPI perform the same and, this time, PAVBAPI delivers solutions of better quality compared to BAPI. By using the kick methods,

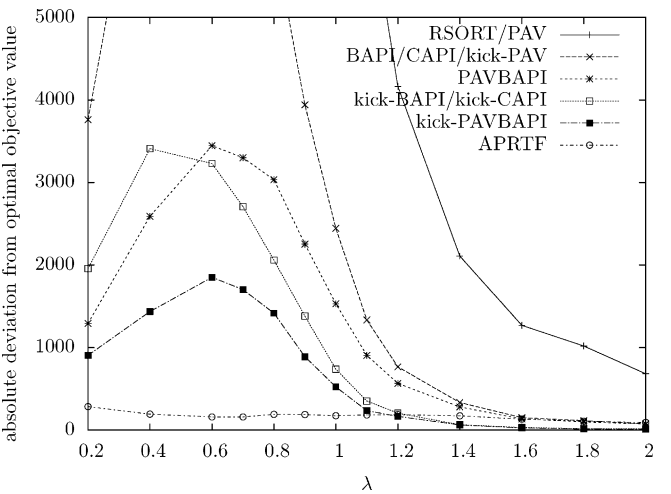

Fig. 5 Iterative improvement using RSORT to obtain initial solutions

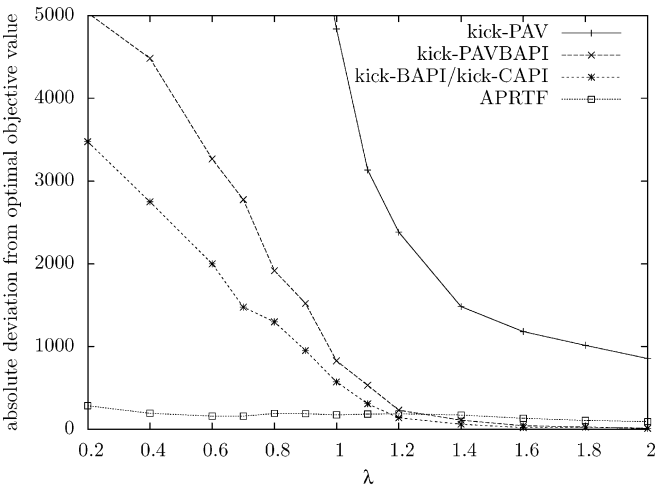

Fig. 6 Iterative improvement using RND to obtain initial solutions

we get the same ranking between the different neighborhoods except that the differences to the optimal values become smaller.

At last, we conducted tests using RND to obtain initial solutions. The initial solutions are of very bad quality, and the differences to the optimal solutions are big. The simple methods BAPI, CAPI, and PAV (as well as PAVBAPI) very soon get stuck in a local optima and, thus, deliver solutions of bad quality. BAPI, CAPI, and PAVBAPI perform superior over PAV alone and give solutions of comparable quality. They reduce the gap of the initial solutions to the optimum by roughly 50% . In Fig. 6, we only present the results using the kick methods. The solutions of the simple methods are far off limits. We again get a situation, where kick-BAPI and kick-CAPI perform the same and slightly better than kick-PAVBAPI. One observation is that the kick methods are successful in improving on randomly generated initial solutions.

The additional tests confirm that the navigation behavior of the very large-scale neighborhood $\mathcal{N}_{CAPI}$ is not better than that of the underlying basic neighborhood $\mathcal{N}_{API}$. Always taking in a greedy way the best solution of the neighborhood $\mathcal{N}_{API}$ does not generally cut off possible moves, which are contained in the best solution of $\mathcal{N}_{CAPI}$. Furthermore, a priority-based, very large-scale neighborhood is not very strong by itself. However, to incorporate these priorities in other neighborhoods (either via switching between the neighborhoods or by directly incorporating them) seems to be a good idea. The presented results, until now, only give a qualitative judgment of the neighborhoods. Although the very large-scale neighborhoods were not superior in this direction compared to $\mathcal{N}_{API}$, they still may be efficient if they

n	BAPI	CAPI	PAV	PAVBAPI
100	0.018	0.022	0.000	0.002
200	0.159	0.242	0.000	0.030
500	2.837	6.244	0.000	0.300
1000	24.715	77.086	0.003	1.594

Fig. 7 Average running time per instance averaged over λ

n	BAPI	CAPI	PAV	PAVBAPI
100	1669	1466	7	249
200	7037	6439	9	657
500	46674	44291	11	2195
1000	195104	189052	12	5272

Fig. 8 Average number of iterations per instance averaged over λ

reduce the computational effort. Therefore, in the following we investigate this aspect in more detail.

In Fig. 7, the average running time is presented for iterative improvement using an initial solution received by RND on instances with 100, 200, 500, and 1000 jobs. The outcome is averaged over all considered values for λ. It can be seen that the running times for CAPI is roughly twice as long as BAPI, and PAV and PAVBAPI are fast compared to the other methods.

Thus, also in this aspect, the very large-scale neighborhood $\mathcal{N}_{CAPI}$ does not outperform its simple counterpart $\mathcal{N}_{API}$. To get an explanation for this, we calculate the average number of iterations needed for iterative improvement to reach a local optimal solution (see Fig. 8) and the average number of API moves, which were contained in one CAPI move for $n = 1000$ and different values of λ (see Fig. 9). The results show that almost the same number of iterations is needed for BAPI and CAPI. Furthermore, CAPI is not able to significantly combine several API moves.

Finally, we calculated for $n = 1000$ in cases that the improvement of a solution in the combined approach PAVBAPI was achieved by BAPI (see Fig. 10). The average need for BAPI increases for higher values of λ because of the structure of these instances. To rearrange a local optima with respect to $\mathcal{N}_{PAV}$ by BAPI moves to achieve a solution, which is no longer locally optimal with respect to $\mathcal{N}_{PAV}$, in average more than one step of BAPI is needed. Hence, it takes some iterations using BAPI before PAV can again improve the given solution. This is specially the case for instances with higher values of λ. These instances have a wider range of release dates; hence, it is more likely to receive empty valleys. And if a nonempty valley is obtained by BAPI, mostly either the valley and the corresponding peak are already sorted by processing times or it would not improve the objective value if this sorting is done by PAV.

5 Conclusions

The very large-scale neighborhood $\mathcal{N}_{CAPI}$ received by combining independent API moves, does not automatically lead to solutions of better quality. In addition, the hope for a faster running time of iterative improvement because of the execution of several moves did not come out at once. In fact, $\mathcal{N}_{CAPI}$ is not able to combine enough moves in any testing to beat $\mathcal{N}_{API}$ regarding computational time.

λ	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.4	1.6	1.8	2.0
API	1.02	1.02	1.02	1.03	1.03	1.03	1.04	1.05	1.05	1.06	1.07

Fig. 9 Average moves per iteration in *CAPI*-neighborhood

λ	0.4	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.4	1.6
%	67.9	69.5	69.0	69.1	69.2	70.4	70.3	70.9	72.3	72.5

Fig. 10 Average amount of cases the improvement is made by BAPI in PAVBAPI

Furthermore, the second very large-scale neighborhood $\mathcal{N}_{PAV}$ alone is not able to deliver good results. This neighborhood is only useful to improve *unstructured* feasible solutions to some better solution in a very fast way. It does not succeed on solutions of good quality since these solutions have a near optimal structure regarding $\mathcal{N}_{PAV}$. But in combining $\mathcal{N}_{API}$ and $\mathcal{N}_{PAV}$, iterative improvement is able to improve an initial solution of bad quality in a fast way in comparison to the stand-alone neighborhood $\mathcal{N}_{API}$ and reaches solutions of comparable quality. Therefore, $\mathcal{N}_{PAV}$ is only useful in combination with other neighborhoods.

Since iterative improvement is not a clever local search algorithm and depends strongly on the initial solution, we added a simple restart technique by perturbing the received local optima. By doing this, we observed a vast increase in the solution quality. Hence, we expect that with other more elaborated methods, as simulated annealing or tabu search, it will be possible to receive even better local optima with the two introduced neighborhoods.

The results of this paper somehow confirm the conclusions drawn by others on the practical use of very large-scale neighborhoods (see, e.g., Hurink 1999). Very large-scale neighborhoods are not good for making local search efficient beforehand. On the basis of our experiences, we may conclude that the size of the neighborhood does not guarantee a better quality. Very large-scale neighborhoods may be successful only if structural properties of the considered problem make it useful to combine different neighborhood operators. This means that these combined neighborhood operators lead to different and better solutions as a sequence of (greedy) chosen neighborhood steps in the underlying basic neighborhood (for our problem, this was not the case!). On the other hand, large neighborhoods resulting from dominance rules are not useful as stand-alone methods. They may help to improve the quality only when combined with other neighborhoods.

A second possible advantage of very large-scale neighborhoods, consisting of combined operators of a basic neighborhood, can be speed in computational time. On the basis of our results, we may conclude that this can only be the case if most of the executed combined neighborhood operators combine several basic neighborhood operators, and if the extra computational effort for searching the combined neighborhood in comparison with the basic neighborhood is not large.

All in all, we suggest to develop and use very large-scale neighborhoods of the considered types only if problem-specific properties or computational arguments give an indication beforehand that the very large-scale neighborhoods have some potential to be a success.

Acknowledgements The authors are grateful to the anonymous referees for their helpful comments on an earlier draft of the paper.
Tobias Brueggemann is supported by the Netherlands Organization for Scientific Research (NWO) grant 613.000.225 (Local Search with Exponential Neighborhoods). Johann L. Hurink is supported by BSIK grant 03018 (BRICKS: Basic Research in Informatics for Creating the Knowledge Society).

Appendix

Review on complexity

We present the $\mathcal{NP}$ -hardness proof for Theorem 2. The proof is based on the $\mathcal{NP}$ -hardness proof of the considered scheduling problem. In Lenstra et al. (1977), it is mentioned that a reduction of 3 -Partition to the decision version of the scheduling problem $1|r_j \geq 0|\sum C_j$ can be obtained by adapting the transformation of Knapsack to $1|r_n \geq 0|\sum C_i$ presented by Rinnooy Kan (1976). This can be carried out in a straightforward way and leads to a similar construction.

We prove that 3 -Partition can be reduced pseudopolynomially to $1|r_j \geq 0|\sum C_j$. For 3 -Partition positive integers $a_1, \ldots, a_{3t}$ and b are given with

$$\frac{b}{4} < a_j < \frac{b}{2}, j = 1, \ldots, 3t \text{ and}$$
$$\sum_{j=1}^{3t} a_j = tb.$$

It is asked if there exists a partition of $T = \{1, \ldots, 3t\}$ into pairwise disjoint sets $T_1, \ldots, T_t \subseteq T$, such that

$$\sum_{j \in T_i} a_j = b$$

for all $i = 1, \ldots, t$. We define $\overline{b} := b + 1$.

The instance of $1|r_j \geq 0|\sum C_j$ corresponding to a given instance of 3 -Partition is defined as follows. The set of jobs is given by $J = S \cup B$ with

$$\begin{array}{rcl} S & := & \{1, \ldots, 3t\}, \\ B_i & := & \{(i,k) : k = 1, \ldots, m\}, i = 0, \ldots, t-1, \\ B_t & := & \{(t,k) : k = 1, \ldots, M\}, \\ B & := & B_0 \cup \ldots \cup B_t, \end{array}$$

where the concrete values of m and M are

$$\begin{array}{rcl} m & := & \frac{3\overline{b}}{2} t(t+1) + 1, \\ M & := & m\frac{3\overline{b}}{2} t(t+1) = \frac{9\overline{b}^2}{4} t^2(t+1)^2 + \frac{3\overline{b}}{2} t(t+1). \end{array}$$

The jobs have the following release dates r_j and processing times $p_j, j \in J$:

$$\begin{array}{rcl} r_j & := & 0 \text{ and } p_j := a_j \text{ for } j \in S, \\ r_{(i,k)} & := & i\overline{b} + (k-1)\varepsilon \text{ and } p_{(i,k)} := \varepsilon \text{ for } (i,k) \in B, \end{array}$$

with $\varepsilon := \frac{1}{m}$. It is asked if a schedule of the jobs exists with an objective value of at most

$$f^* = \frac{\overline{b}m^2 + 3\overline{b}m}{2m} t^2 + \frac{m^2 - \overline{b}m^2 + 2\overline{b}Mm + 3\overline{b}m + m}{2m} t + \frac{M^2 + M}{2m}.$$

Observe that if the jobs of all sets B_i are scheduled at their release dates, they leave free time windows $I_i := [i\overline{b} - b, i\overline{b},] i = 1, \ldots, t$, for the remaining jobs (Fig. 11).

Because of the release dates, the contribution f_{B_i} and f_{B_t} of jobs of B_i and B_t to the objective value can be bounded from below, as follows:

$$\begin{aligned} f_{B_i} &:= \sum_{k=1}^{m} C_{(i,k)} \geq \sum_{k=1}^{m} \left(r_{(i,k)} + p_{(i,k)}\right) \\ &= \overline{b}mi + \tfrac{m+1}{2} =: f^*_{B_i} \text{ for } i = 0, \ldots, t-1, \text{ and} \\ f_{B_t} &:= \sum_{k=1}^{M} C_{(t,k)} \geq \sum_{k=1}^{M} \left(r_{(t,k)} + p_{(t,k)}\right) \\ &= \overline{b}Mt + \tfrac{M^2+M}{2m} =: f^*_{B_t}. \end{aligned}$$

With this, we receive for the contribution of jobs of B to the objective value the following lower bound:

$$f_B := f_{B_t} + \sum_{i=0}^{t} f_{B_i} \geq \frac{\overline{b}m}{2} t^2 + \frac{2\overline{b}M - \overline{b}m + m + 1}{2} t + \frac{M^2 + M}{2m} =: f^*_B.$$

Furthermore, if all jobs from $B_0, \ldots, B_t$ start at their release date, we have $f_B = f^*_B$.

Now assume that we have a solution $T_1, \ldots, T_t$ of 3-Partition. In this case, we can schedule the jobs of $T_i \subseteq S$ completely in the corresponding window I_i and they contribute at most

$$\begin{aligned} f_S &:= \sum_{i=1}^{t} \sum_{j \in T_i} C_j \leq \sum_{i=1}^{t} \sum_{j \in T_i} i\overline{b} = \sum_{i=1}^{t} 3i\overline{b} \\ &= \tfrac{3\overline{b}}{2} t(t+1) := f^*_S \end{aligned}$$

to the objective value. Furthermore, since the jobs of S fit completely into the time windows $I_1, \ldots, I_t$, we can schedule the jobs of B at their release dates. Thus, their contribution to the objective value is given by f^*_B, and the objective value of the whole schedule is bounded by $f^*_S + f^*_B = f^*$.

Consider now that we have an optimal schedule for the jobs of J with objective value of $f^* = f^*_S + f^*_B$ at most. We show that 3-Partition has a feasible solution. Since all jobs of B have the same processing time, we may assume that, in the optimal schedule, the jobs of B are processed according to the order of their release dates.

Now, assume that, in the optimal schedule, two jobs (i, k) and $(i, k+1)$ from B are not scheduled next to each other. This implies that a subset $\overline{S} \subseteq S$ of jobs is

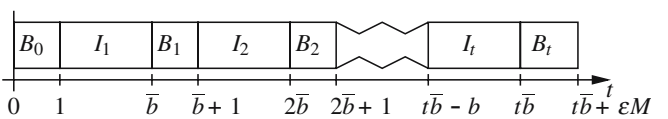

Fig. 11 Sets B and I

scheduled in between these two jobs. However, since the jobs of $\overline{S}$ have far larger processing times than the jobs from B and $r_{(i,k+1)} = r_{(i,k)} + p_{(i,k)} \leq C_{(i,k)}$, an interchange of job $(i, k+1)$ with $\overline{S}$ reduces the optimal value. As a consequence, all jobs of each set B_i, $i = 0, \ldots, t-1$ are scheduled together in a block of length 1, and the jobs of B_t are scheduled together in a block of length M/m.

Next we prove, that in each schedule with an objective value of at most $f^* = f_S^* + f_B^*$ all the jobs of B are scheduled at their release dates. Assume that job $(i, 1)$ is the first job of B not scheduled at its release date. This implies $S_{(i,1)} = r_{(i,1)} + \delta = i\overline{b} + \delta$, where $\delta \geq 1$, since all jobs from S have integer processing times. This leads to a contribution of the jobs from B_i of at least $f_{B_i}^* + m\delta$. Since

$$m > \frac{3\overline{b}}{2} t(t+1) = f_S^*,$$

this leads to an objective value of more than $f_B^* + f_S^* = f^*$, which is a contradiction.

As a consequence for the optimal schedule, the jobs of S can only be scheduled in the intervals $I_1, \ldots, I_t$, and after $t\overline{b} + \varepsilon M$. If we can prove that the latter is not possible, we get that the optimal schedule contains a solution of 3-Partition.

Assume that there is at least one $j \in S$ with $C_j \geq t\overline{b} + \varepsilon M$. This leads to an objective value of at least $C_j + f_B^* \geq t\overline{b} + \varepsilon M + f_B^*$. Since $\varepsilon = \frac{1}{m}$ and

$$t\overline{b} + \frac{M}{m} > \frac{3\overline{b}}{2} t(t+1) = f_S^*,$$

this contradicts that the objective value of the given schedule is bounded by $f_S^* + f_B^*$.

In summary, 3-Partition pseudopolynomially reduces to $1|r_j \geq 0| \sum C_j$. Therefore, the decision problem of $1|r_j \geq 0| \sum C_j$ is $\mathcal{NP}$-complete in the strong sense.

In order to prove Theorem 2, we now identify the positions of the jobs of B as the indices $i_1, \ldots, i_k$ (of Theorem 2), and the corresponding jobs of B as the peaks (of Theorem 2). Finding a schedule respecting these peaks with objective value less than f^* solves the 3-Partition problem. This proves Theorem 2.

References

Ahmadi RH, Bagchi U (1990) Lower bounds for single-machine scheduling problems. Nav Res Logist 37(6):967–979

Ahuja RK, Özlem E, Orlin JB, Punnen AP (2002) A survey of very large-scale neighborhood search techniques. Discrete Appl Math 123:75–102

Baker KR (1974) Introduction to sequencing and scheduling. Wiley, New York

Chu C (1992) A branch-and-bound algorithm to minimize total flow time with unequal release dates. Nav Res Logist 39:859–875

Congram RK, Potts CN, van de Velde SL (2002) An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. INFORMS J Comput 14(1):52–67

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

Hurink J (1999) An exponential neighborhood for a one machine batching problem. OR Spectrum 21:461–476

Lenstra JK, Rinnooy Kan AHG, Brucker P (1977) Complexity of machine scheduling problems. Ann Discrete Math 1:343–362

Potts CN, van de Velde SL (1995) Dynasearch-iterative local improvement by dynamic programming. Part 1. The traveling salesman problem. Technical Report, University of Twente, The Netherlands

Rinnooy Kan AHG (1976) Machine scheduling problems: classification, complexity and computations. Martinus Nijhoff, The Hague, pp 79–88

Smith WE (1956) Various optimizers for single-stage production. Nav Res Logist Q 3:59–66

Yanai S, Fujie T (2004) On a dominance test for the single machine scheduling problem with release dates to minimize total flow time. J Oper Res Soc Jpn 47(2):96–111