algorithms

MDPI

Article

On Neighborhood Structures and Repair Techniques for Blocking Job Shop Scheduling Problems †

Julia Lange [1,*] and Frank Werner [2]

[1] Division of Information Process Engineering, FZI Forschungszentrum Informatik Karlsruhe, 76131 Karlsruhe, Germany
[2] Fakultät für Mathematik, Otto-von-Guericke-Universität Magdeburg, D-39016 Magdeburg, Germany; frank.werner@ovgu.de
* Correspondence: lange@fzi.de
† This paper is an extended version of our paper published in the proceedings of the 9th IFAC Conference on Manufacturing Modeling, Management, and Control.

Received: 23 October 2019; Accepted: 11 November 2019; Published: 12 November 2019

Abstract: The job shop scheduling problem with blocking constraints and total tardiness minimization represents a challenging combinatorial optimization problem of high relevance in production planning and logistics. Since general-purpose solution approaches struggle with finding even feasible solutions, a permutation-based heuristic method is proposed here, and the applicability of basic scheduling-tailored mechanisms is discussed. The problem is tackled by a local search framework, which relies on interchange- and shift-based operators. Redundancy and feasibility issues require advanced transformation and repairing schemes. An analysis of the embedded neighborhoods shows beneficial modes of implementation on the one hand and structural difficulties caused by the blocking constraints on the other hand. The applied simulated annealing algorithm generates good solutions for a wide set of benchmark instances. The computational results especially highlight the capability of the permutation-based method in constructing feasible schedules of valuable quality for instances of critical size and support future research on hybrid solution techniques.

Keywords: job shop scheduling; blocking; total tardiness; permutations; repairing scheme; simulated annealing

1. Introduction

Complex scheduling problems appear in customer-oriented production environments, automated logistics systems, and railbound transportation as an everyday challenge. The corresponding job shop setting, where a set of jobs is to be processed by a set of machines according to individual technological routes, constitutes one of the non-trivial standard models in scheduling research. Even ·simple variants of this discrete optimization problem are proven to be NP-hard, see [1]. While the classical job shop scheduling problem with infinite buffers and makespan minimization has been a subject of extensive studies for many decades, see, for instance [2–4], solving instances with additional real-world conditions, such as the absence of intermediate buffers, given release and due dates of the jobs and recirculation, newly receives increasing attention.

The blocking job shop problem with total tardiness minimization (BJSPT) is regarded as an exemplary complex planning situation in this paper. Blocking constraints refer to a lack of intermediate buffers in the considered system. A job needs to wait on its current machine after processing, and thus blocks it, until the next required machine is idle. Such situations occur, for instance, in the manufacturing of huge items, in railbound or pipeline-based production and logistics as well as in train scheduling environments; see, for instance [5–8]. The consideration of a tardiness-based objective

implements efficient economic goals like customer satisfaction and schedule reliability as they appear in most enterprises.

On the one hand, existing computational experiments indicate that exact general-purpose solution methods have significant difficulties in finding optimal and even feasible solutions for non-classical job shop instances of practically relevant size; see, for instance [9–11]. On the other hand, the application of special-purpose heuristics shows a necessity of complicated construction, repairing and guiding schemes to obtain good solutions; see [12–14]. This work is intended to analyze the capability of well-known scheduling-tailored heuristic search methods in determining high quality solutions for complex job shop scheduling problems. Structural reasons for the appearing complexity are detected and algorithms, which assure the applicability of basic strategies, are proposed.

As a natural foundation, permutations shall be used to represent feasible schedules of BJSPT instances. Widely applied interchange- and shift-based operators are chosen as transition schemes to set up neighboring solutions. Combining these ideas causes considerable redundancy and feasibility issues. A *Basic Repair Technique* (BRT) is proposed to construct a feasible schedule from any given permutation, cf. [15,16]. To fit the requirements in generating neighboring solutions, it is extended to an *Advanced Repair Technique* (ART), which defines a feasible neighboring schedule from an initial permutation and a desired interchange, see [15,16].

The resulting distances of solutions in a neighborhood are discussed to shed light onto the nature of the search space. In addition, different shifting strategies are analyzed with regard to their advantageousness in the search process. The presented neighborhood structures are embedded in a simulated annealing (SA) metaheuristic, which is applied to solve a diverse set of benchmark instances. Beneficial and critical aspects regarding the quality of the schedules found and the search guidance are pointed out by the computational results.

The remainder of the article is organized as follows. Section 2 summarizes existing work on complex job shop scheduling problems related to the BJSPT. A theoretical description of the problem and its notation are given in Section 3. Two variants of permutation-based encodings of schedules are discussed with regard to redundancy and feasibility in Section 4. Therein, the BRT is introduced and the distance of two schedules is defined. Section 5 incorporates explanations on the applied transition schemes, their implementation, and the operating principles of the ART. Furthermore, the neighborhoods are described and characteristics such as connectivity and solution distance are analyzed. Computational experiments on solving the BJSPT by an SA algorithm are reported in Section 6. Finally, Section 7 concludes with the main findings and further research perspectives.

2. Literature Review

A variety of exact and heuristic solution approaches to complex scheduling problems reported in the literature exist. This section will focus on observations and findings on job shop problems featuring constraints and optimization criteria similar to the BJSPT.

Exact solution methods are only sparsely applied to job shop problems with blocking constraints or tardiness-based objective functions. In 2002, Mascis and Pacciarelli [17] present a Branch & Bound procedure that is enhanced by scheduling-tailored bounds and a special branching technique. The approach is tested on complex instances with ten machines and ten jobs involving blocking constraints and makespan minimization. Obtaining proven optimal solutions for the benchmark problems takes between 20 min and four hours of computation time. Even if technical enhancements have been achieved and general-purpose mixed-integer programming solvers became more powerful, the job shop scheduling problem remains one of the hardest combinatorial optimization models. It is recently shown in [9,11,15,18] that even sophisticated mixed-integer programming solvers, such as IBM ILOG CPLEX and Gurobi, struggle with finding optimal and even feasible solutions to BJSPT instances with up to 15 machines processing up to 20 jobs in reasonable computation time.

Table 1 summarizes the heuristic approaches presented for job shop problems involving blocking constraints and tardiness-based objectives. A reference is stated in the first column, while the second

column specifies whether a job shop problem with blocking constraints (BJSP) or without such restrictions (JSP) is examined. Column three contains the objective function regarded and the fourth column displays the maximal size (m, n) of the considered instances, where m denotes the number of machines and n defines the number of jobs. The last column presents the applied heuristic technique.

For reasons of comparison, the first two works [19,20] mentioned in Table 1 deal with a classical variant of the problem, namely the job shop problem with makespan minimization. The applied solution approaches constitute fundamental heuristic methods and the best-known algorithms to solve instances of the standard job shop setting until today. With the popular $(10, 10)$ instance of Fisher and Thompson having been open for decades, the size of standard job shop problems, for which good solutions can be obtained, has grown. However, most of the large instances have never been solved to optimality, which highlights the significant intricacy of the combinatorial optimization problems under study.

The following set of studies on JSPs with tardiness-based optimization criteria is intended to show the variety and the evolution of heuristic solution approaches applied together with the limitations in solvable problem size. A more comprehensive literature review including details on the types of instances solved can be found in [21].

In [22], a shifting bottleneck procedure is presented to generate schedules with minimal total tardiness for JSPs with release dates. The method is tested on a set of benchmark instances of size $(10, 10)$. A well-known critical path-oriented neighborhood, cf. [2], is discussed with regard to its applicability to pursue tardiness-based objectives in [23]. The authors tackle JSPs with total tardiness minimization by Simulated Annealing (SA) and show that a general neighborhood based on interchanges of adjacent operations on machines leads to better results. A hybrid genetic algorithm (GA) is proposed for JSPs with recirculation, release dates and various tardiness-based objective functions in [14]. Even if the procedure is enhanced by a flexible encoding scheme and a decomposition approach, the results do not significantly support GAs as a favorable solution method. The computational experiments are conducted on a set of twelve instances with maximum size $(8, 50)$. In [24], a generalized JSP consisting of a set of operations with general precedence constraints and required time lags is optimized with regard to total weighted tardiness. The authors apply a tabu search (TS) approach where neighboring solutions are also constructed by interchanges of adjacent operations, and the starting times of the operations are calculated based on a network flow. Here, the instance size of $(10, 10)$ is still critical.

A classical job shop setting involving release dates and minimizing the total weighted tardiness is considered in [25]. The authors combine a GA with an iterative improvement scheme and discuss the effect of various search parameters. Computational experiments are conducted on the set of benchmark instances with up to 15 machines and 30 jobs. The iterative improvement scheme seems to be a highly beneficial part of the solution approach, since it counteracts the occurring quality variance of consecutively constructed solutions. The same type of problems has also been tackled by a hybrid shifting bottleneck procedure including a preemption-allowing relaxation and a TS step in [26]. A more general type of optimization criteria, namely regular objective functions, are considered for JSPs in [27]. An enhanced local search heuristic with modified interchange-based neighborhoods is applied. The computational experiments are based on a large set of instances, where the most promising results are obtained for problems with up to 10 machines and 15 jobs. In [28], the critical path-based neighborhood introduced in [2] is used together with a block reversion operator in SA. Numerical results indicate that the combination of transition schemes is beneficial for finding job shop schedules with minimal total weighted tardiness. This work involves the largest test instances featuring 10 machines and 50 jobs as well as five machines and 100 jobs. However, these instances do not constitute a commonly used benchmark set so that no optimal solutions or lower bounds are known or presented to evaluate the capability of the method.

Table 1. Overview of existing heuristic solution approaches related to the BJSPT.

Reference	Problem	Objective *	Max. Size (m, n)	Solution Approach
Nowicki and Smutnicki 2005 [19]	JSP	C_{max}	$(10,50), (20,100)$	Tabu Search
Balas et al. 2008 [20]	JSP	C_{max}	$(22,75)$	Shifting Bottleneck Procedure
Singer and Pinedo 1999 [22]	JSP	$\sum w_i T_i$	$(10,10)$	Shifting Bottleneck Algorithm
Wang and Wu 2000 [23]	JSP	$\sum T_i$	$(30,90)$	Simulated Annealing
Mattfeld and Bierwirth 2004 [14]	JSP	tardiness-based	$(8,50)$	Genetic Algorithm
De Bontridder 2005 [24]	JSP	$\sum w_i T_i$	$(10,10)$	Tabu Search
Essafi et al. 2008 [25]	JSP	$\sum w_i T_i$	$(10,30), (15,15)$	Hybrid Genetic Algorithm with Iterated Local Search
Bülbül 2011 [26]	JSP	$\sum w_i T_i$	$(10,30), (15,15)$	Hybrid Shifting Bottleneck Procedure with Tabu Search
Mati et al. 2011 [27]	JSP	regular	$(20,30), (8,50)$	Local Search Heuristic
Zhang and Wu 2011 [28]	JSP	$\sum w_i T_i$	$(15,20), (10,50), (5,100)$	Simulated Annealing
Gonzalez et al. 2012 [29]	JSP	$\sum w_i T_i$	$(10,30), (15,20)$	Hybrid Genetic Algorithm with Tabu Search
Kuhpfahl and Bierwirth 2016 [30]	JSP	$\sum w_i T_i$	$(10,30), (15,15)$	Local Descent Scheme, Simulated Annealing
Bierwirth and Kuhpfahl 2017 [21]	JSP	$\sum w_i T_i$	$(10,30), (15,15)$	Greedy Randomized Adaptive Search Procedure
Brizuela et al. 2001 [12]	BJSP	C_{max}	$(20,20)$	Genetic Algorithm
Mati et al. 2001 [8]	BJSP	C_{max}	$(10,30)$	Tabu Search
Mascis and Pacciarelli 2002 [17]	BJSP	C_{max}	$(10,30), (15,20)$	Greedy Heuristics
Meloni et al. 2004 [31]	BJSP	C_{max}	$(10,10)$	Rollout Metaheuristic
Gröflin and Klinkert 2009 [13]	BJSP	C_{max}	$(10,50), (15,20), (20,20)$	Tabu Search
Oddi et al. 2012 [32]	BJSP	C_{max}	$(10,30), (15,15)$	Iterative Improvement Scheme
AitZai and Boudhar 2013 [33]	BJSP	C_{max}	$(10,30), (15,15)$	Particle Swarm Optimization
Pranzo and Pacciarelli 2016 [34]	BJSP	C_{max}	$(10,30), (15,20)$	Iterative Greedy Algorithm
Bürgy 2017 [9]	BJSP	regular	$(10,30), (15,20), (20,30)$	Tabu Search
Dabah et al. 2019 [35]	BJSP	C_{max}	$(10,30), (15,15)$	Parallel Tabu Search

* Objective Functions: makespan C_{max}, total tardiness $\sum T_i$, total weighted tardiness $\sum w_i T_i$, various *regular* or *tardiness-based* objectives.

A JSP with setup times and total weighted tardiness minimization is tackled by a hybrid heuristic technique in [29]. A TS method is integrated into a GA to balance intensification and diversification in the search process. Furthermore, an improvement potential evaluation is applied to guide the selection of neighboring solutions in the TS. Promising results are found on a widely used set of benchmark instances. Different neighborhood structures are discussed and analyzed according to their capability of constructing schedules for a JSP with release dates and total weighted tardiness minimization in [30]. The experimental results show that the choice of the main metaheuristic method and the initial solution influence the performance significantly. Complex neighborhood structures involving several partially critical path-based components yield convincing results for instances with up to 15 machines and 30 jobs. In [21], an enhanced Greedy Randomized Adaptive Search Procedure (GRASP) is proposed and tested on the same set of benchmark instances. The applied method involves a neighborhood structure based on a critical tree, a move evaluation scheme as well as an amplifying and a path relinking strategy. The comprehensive computational study of Bierwirth and Kuhpfahl [21] shows that the presented GRASP is able to compete with the most powerful heuristic techniques tackling JSP instances with total tardiness minimization, namely the GA-based schemes proposed by Essafi et al. [25] and Gonzalez et al. [29]. Overall, the complexity of the applied methods, which is required to obtain satisfactory results for instances of still limited size, highlights the occurring difficulties in guiding a heuristic search scheme based on tardiness-related objective functions.

Considering the second set of studies on BJSPs given in Table 1, an additional feasibility issue arises and repairing or rescheduling schemes become necessary. The inclusion or exclusion of swaps of jobs on machines constitutes a significant structural difference with regard to real-world applications and the applied solution approach, see Section 4.2 for further explanations. Note that almost all existing solution approaches are dedicated to makespan minimization, even if this does not constitute the most practically driven objective.

In [12], a BJSP involving up to 20 machines and 20 jobs with swaps is tackled by a GA based on a permutation encoding. The well-known critical path-oriented transition scheme, cf. [2], is applied together with a job insertion-based rescheduling method in a TS algorithm in [8]. The authors consider a real-world application where swaps are not allowed and test their approach on instances with up to 10 machines and 30 jobs. Different greedy construction heuristics are compared in solving BJSP instances with and without swaps in [17]. Even for small instances, the considered methods have significant difficulties in constructing feasible schedules, since the completion of an arbitrary partial schedule is not always possible. The same issue occurs in [31], where a rollout metaheuristic involving a scoring function for potential components is applied to BJSP instances of rather small size with and without swaps.

A connected neighborhood relying on interchanges of adjacent operations and job reinsertion is presented for the BJSP in [13]. Instances involving setup and transfer times, and thus excluding swaps, are solved by a TS algorithm with elite solutions storage. Computational experiments are conducted on a large set of benchmark instances with up to 20 machines and 50 jobs. In [32], an iterative improvement algorithm incorporating a constraint-based search procedure with relaxation and reconstruction steps is proposed for the BJSP with swaps. A parallel particle swarm optimization is tested on instances of the BJSP without swaps in [33] but turns out not to be competitive with the method proposed in [13] and the following one. In [34], an iterated greedy algorithm, which loops deconstruction and construction phases, is applied to problems with and without swaps. Computational experiments on well-known benchmark instances with up to 15 machines and 30 jobs imply that forced diversification of considered solutions is favorable to solve the BJSP. A study tackling instances with up to 20 machines and 30 jobs and approaching a wider range of regular objective functions including total tardiness is reported in [9]. The authors embed a job reinsertion technique initially proposed in [36] in a TS and test their method on the BJSP with swaps. A parallel TS including the critical path-oriented neighborhood, cf. [2], and construction heuristics to recover feasibility is presented in [35]. Parallel search trajectories without communication are set up to increase the number of considered solutions.

Overall, the most promising approaches to solve BJSP instances proposed by Bürgy [9], Dabah et al. [35], and Pranzo and Pacciarelli [34] give evidence for focusing on the application of sophisticated neighborhood and rescheduling structures instead of increasing the complexity of the search procedure itself. This motivates the following work on evaluating the capability of basic scheduling-tailored techniques. Furthermore, a study on the interaction of blocking constraints and tardiness-based optimization criteria will be provided.

3. Problem Description and Benchmark Instances

The BJSPT is defined by a set of machines $\mathcal{M} = \{M_k \mid k = 1, \ldots, m\}$ which are required to process a set of jobs $\mathcal{J} = \{J_i \mid i = 1, \ldots, n\}$ with individual technological routes. Each job consists of a set of operations $\mathcal{O}^i = \{O_{i,j} \mid j = 1, \ldots, n_i\}$, where operation $O_{i,j}$ describes the j-th non-preemptive processing step of job J_i. The overall set of operations is defined by $\mathcal{O} = \bigcup_{J_i \in \mathcal{J}} \mathcal{O}^i$ containing n_{op} elements. Each operation $O_{i,j}$ requires a specific machine $Ma(O_{i,j})$ for a fixed processing time $p_{i,j} \in \mathbb{Z}_{>0}$. The recirculation of jobs is allowed. Furthermore, a release date $r_i \in \mathbb{Z}_{\geq 0}$ and a due date $d_i \in \mathbb{Z}_{>0}$ are given for every job $J_i \in \mathcal{J}$.

Blocking constraints are introduced for every pair of operations $O_{i,j}$ and $O_{i',j'}$ of different jobs requiring the same machine. Given that $O_{i,j} \rightarrow O_{i',j'}$ determines the operation sequence on the corresponding machine M_k and $j \neq n_i$ holds, the processing of operation $O_{i',j'}$ cannot start before job J_i has left machine M_k, in other words, the processing of operation $O_{i,j+1}$ has started. To account for the optimization criterion, a tardiness value is determined for every job with $T_i = max\{0,\ C_i - d_i\}$, where C_i describes the completion time of the job.

There exist different mathematical formulations of the described problem as a mixed-integer optimization program. For an overview of applicable sequence-defining variables and comprehensive studies on advantages and disadvantages of the corresponding models, the reader is referred to [11,15]. According to the well-known three-field notation, cf. for instance [37,38], the BJSPT can be described by

$$Jm \mid block, recrc, r_i \mid \sum T_i.$$

A feasible schedule is defined by the starting times $s_{i,j}$ of all operations $O_{i,j} \in \mathcal{O}$, which fulfill the processing sequences, the technological routes and the release dates of all jobs as well as the blocking constraints. Since the minimization of total tardiness constitutes a regular optimization criterion, it is sufficient to consider semi-active schedules where no operation can be finished earlier without modifying the order of processing of the operations on the machines, see e.g., [38,39]. Thus, the starting times of the operations and the operation sequences on the machines constitute uniquely transformable descriptions of a schedule. If a minimal value of the total tardiness of all jobs $\sum_{J_i \in \mathcal{J}} T_i$ is realized, a feasible schedule is denoted as optimal. Note that, regarding the complexity hierarchies of shop scheduling problems, see for instance [38,40] for detailed explanations, the BJSPT is harder than the BJSP with the minimization of the makespan C_{max}.

To discuss the characteristics of neighborhood structures and to evaluate their performance, a diverse set of benchmark instances is used. It is intended to involve instances of different sizes (m, n) featuring different degrees of inner structure. The set of problems contains train scheduling-inspired (ts) instances that are generated based on a railway network, distinct train types, routes and speeds, cf. [11,15], as well as the Lawrence (la) instances which are set up entirely random with $n_i = m$ for $J_i \in \mathcal{J}$, cf. [41]. The problems include 5 to 15 machines and 10 to 30 jobs. The precise instance sizes can be found in Tables 2–4.

For all instances, job release dates and due dates are generated according to the following terms in order to create computationally challenging problems. The release dates are restricted to a time

interval which forces jobs to overlap and the due dates are determined with a tight due date factor, see [9,11,30]:

$$r_i \in \left[0\,,\ 2\cdot\min_{J_i\in\mathcal{J}}\left\{\sum_{j=1}^{n_i} p_{ij}\right\}\right] \quad \text{and} \quad d_i = \left[r_i + \left(1.2\cdot\sum_{j=1}^{n_i} p_{ij}\right)\right] \qquad \text{for all } J_i \in \mathcal{J}. \tag{1}$$

4. Representations of a Schedule

The encoding of a schedule is basic to every heuristic solution approach. In contrast to most of the existing work on BJSPs, the well-known concept of permutation-based representations is used here. In the following, redundancy and feasibility issues will be discussed and overcome, and a distance measure for two permutation-based schedules is presented.

4.1. Permutation-Based Encodings

An *operation-based representation* s^{op} of a schedule, also called permutation, is given as a single list of all operations. Consider exemplarily

$$s^{op} = [O_{i,1}, O_{i',1}, O_{i,2}, O_{i'',1}, O_{i,3}, O_{i',2}, \ldots].$$

The permutation defines the operation sequences on the machines, and the corresponding starting times of all operation can be determined by a list scheduling algorithm. Note that the processing sequences of the jobs are easily satisfiable with every operation $O_{i,j}$ having a higher list index than its job predecessor $O_{i,j-1}$. Furthermore, blocking restrictions can be implemented by list index relations so that the feasibility of a schedule can be assured with the operation-based representation. However, when applying the permutation encoding in a heuristic search procedure, redundancy issues need to be taken into account. Regarding the list s^{op} shown above and assuming that the first two operations $O_{i,1}$ and $O_{i',1}$ require different machines, the given ordering $O_{i,1} \rightarrow O_{i',1}$ and the reverse ordering $O_{i',1} \rightarrow O_{i,1}$ imply exactly the same schedule. Generally, the following conditions can be identified for two adjacent operations in the permutation being interchangeable without any effects on the schedule encoded, cf. [15]:

- The operations belong to different jobs.
- The operations require different machines.
- The operations are not connected by a blocking constraint.
- None of the operations is involved in a swap.

Details on the relation of two operations due to a blocking constraint and the implementation of swaps are given in the subsequent Sections 4.2 and 4.3. To avoid unnecessary computational effort caused by treating redundant permutations as different schedules, the application of a unique representation is desirable.

A second permutation-based encoding of a schedule, namely the *machine-based representation* s^{ma}, describes the operation sequences on the machines as a nested list of all operations. Consider

$$s^{ma} = [[O_{i,1}, O_{i'',1}, \ldots], [O_{i',1}, O_{i,2}], \ldots], [O_{i,3}, \ldots], [O_{i',2}, \ldots], \ldots]$$

as a general example, where the k-th sublist indicates the operation sequence on machine M_k. It can be observed that the machine-based representation uniquely encodes these operation sequences and any modification leads to the creation of a different schedule. However, since the machine-based encoding does not incorporate any ordering of operations requiring different machines, the given schedule may be infeasible with regard to blocking constraints. Preliminary computational experiments have shown that this blocking-related feasibility issue frequently appears when constructing BJSP schedules in heuristic search methods. Therefore, both representations are simultaneously used here to assure the uniqueness and the feasibility of the considered schedules.

As a consequence, the applied permutation-based encodings need to be transformed efficiently into one another. Taking the general representations given above as examples, the operation-based encoding features list indices $lidx(O_{i,j})$ and required machines $Ma(O_{i,j})$ as follows:

$$\begin{array}{rccccccc} s^{op} = [& O_{i,1}, & O_{i',1}, & O_{i,2}, & O_{i'',1}, & O_{i,3}, & O_{i',2}, & \ldots] \\ lidx(O_{i,j}): & 1 & 2 & 3 & 4 & 5 & 6 & \ldots \\ Ma(O_{i,j}): & M_1 & M_2 & M_2 & M_1 & M_3 & M_4 & \ldots \end{array}$$

The transformation $s^{op} \rightarrow s^{ma}$ can be performed by considering the operations one by one with increasing list indices in s^{op} and assigning them to the next idle position in the operation sequence of the required machine in s^{ma}, see [15,16]. As an example, after transferring the first two operations $O_{i,1}$ and $O_{i',1}$ from s^{op} to s^{ma}, the machine-based representation turns out as $s^{ma} = [[O_{i,1}], [O_{i',1}], [], [], \ldots]$. After transferring all operations given in the permutation s^{op}, the machine-based encoding exactly corresponds to the nested list shown above.

While performing the transformation $s^{ma} \rightarrow s^{op}$, the redundancy of operation-based encodings needs to be taken into account. If the machine-based representation s^{ma} is constructed from an operation-based representation s^{op} of a specific schedule, it will be desirable that the reverse transformation yields exactly the initially given permutation s^{op} instead of a redundant equivalent. To assure that the resulting list of operations is equivalent or closest possible to an initially given operation-based representation, the following method is proposed.

Priority-Guided Transformation Scheme $s^{ma} \rightarrow s^{op}$, cf. [15]: In transferring a machine-based representation s^{ma} to a permutation $s^{op'}$, the set of candidate operations to be added to the permutation $s^{op'}$ next consists of all operations $O_{i,j}$ in s^{ma}, for which the job predecessor $O_{i,j-1}$ and the machine predecessor given in s^{ma} either do not exist or are already present in $s^{op'}$. Considering the machine-based representation s^{ma} given above and an empty permutation $s^{op'}$, the set of candidate operations to be assigned to the first list index in $s^{op'}$ contains the operations $O_{i,1}$ and $O_{i',1}$. To guarantee the recreation of the initially given permutation s^{op}, the operation $O_{i,j}$ with the maximum priority $prio(O_{i,j})$ is chosen among all candidate solutions, whereby

$$prio(O_{i,j}) = \begin{cases} \frac{1}{lidx(O_{i,j}) - lidx'(*)} & \text{if } lidx'(*) < lidx(O_{i,j}), \\ 2 & \text{if } lidx'(*) = lidx(O_{i,j}), \\ lidx'(*) - lidx(O_{i,j}) + 2 & \text{if } lidx(O_{i,j}) < lidx'(*), \end{cases} \tag{2}$$

with $lidx(O_{i,j})$ being the list index of the operation in the initially given permutation s^{op} and $lidx'(*)$ being the currently considered, idle list index in the newly created list $s^{op'}$. Recalling the example described above, the currently considered index features $lidx'(*) = 1$, while $prio(O_{i,1}) = 2$ due to $lidx'(*) = lidx(O_{i,1}) = 1$ and $prio(O_{i',1}) = \frac{1}{2-1} = 1$ due to $lidx'(*) = 1 < 2 = lidx(O_{i',1})$. Thus, $s^{op'} = [O_{i,1}]$ holds after the first iteration, and the set of candidate operations to be assigned to the next idle list index $lidx'(*) = 2$ consists of the operations $O_{i'',1}$ and $O_{i',1}$. Following this method iteratively, the newly constructed permutation $s^{op'}$ will be equivalent to the initially given permutation s^{op} from which s^{ma} has been derived.

Given that the considered machine-based representation is feasible with regard to the processing sequences and the technological routes of the jobs, the priority-guided transformation scheme will assign exactly one operation to the next idle list index of the new permutation in every iteration and can never treat an operation more than once. Thus, the method constructs a unique operation-based representation from a given machine-based representation in a finite number of $O(n_{op} \cdot m)$ steps, see [15] for detailed explanations.

4.2. Involving Swaps

When considering a lack of intermediate buffers in the job shop setting, the moments in which jobs are transferred from one machine to the next require special attention. The situation where two or more jobs interchange their occupied machines at the same point in time is called a swap of jobs. Figure 1 shows an excerpt of a general BJSP instance involving operations of three jobs with unit processing times on two machines. The Gantt chart in part (a) of the figure illustrates a swap of the jobs J_i and $J_{i'}$ on the machines M_k and $M_{k'}$ at time point $\bar{t}$.

(**a**) A swap of jobs as a feasible cycle in the BSJP schedule

(**b**) A blocking-infeasible cycle in the BJSP schedule

Figure 1. Illustration of feasible and infeasible cycles in BJSP schedules.

Dependent on the type of real-world application, swaps are treated as feasible or infeasible in a BJSP schedule; see, for instance [8,17]. In particular, the implementation of rail-bound systems and the existence of setup times require their exclusion, cf. [5,6,13,34]. In this work, it is assumed that, even if jobs cannot be stored between consecutive processing steps, it is possible to move several jobs simultaneously on the production site. Thus, swaps are treated as feasible here.

Considering the alternative graph representation of a BJSP, initially proposed in [42], reveals an upcoming issue related to swaps and feasibility. In Figure 1, the corresponding excerpt of the alternative graph implementing the given operations as nodes and the implied ordering constraints as arcs is shown on the right next to the Gantt chart. The gray arcs represent the processing sequences of the jobs, while the black arcs indicate the existing operation sequence and blocking constraints. Taking the swap situation in part (a) of Figure 1 as an example again and assuming that the operation $O_{i',2}$ is the last processing step of job $J_{i'}$, the operation sequence $O_{i,1} \rightarrow O_{i',2} \rightarrow O_{i'',1}$ on machine M_k implies the solid arcs $(O_{i,2}, O_{i',2})$ and $(O_{i',2}, O_{i'',1})$ in the alternative graph. Equivalently, the operation sequence $O_{i',1} \rightarrow O_{i,2}$ on machine $M_{k'}$ causes a blocking constraint, which is represented by the dashed arc $(O_{i',2}, O_{i,2})$. The resulting structure of arcs shows that swaps appear as cycles in the alternative graph representation of the schedule. These cycles refer to feasible situations, since the underlying blocking inequalities can simultaneously be fulfilled by an equivalent starting time of all involved operations.

In part (b) of Figure 1, a Gantt chart and the corresponding graph-based representation of infeasible operation sequences are shown as a contrasting example. When trying to determine the starting times of the operations according to the operation sequences on the machines, an infeasible cyclic dependency of ordering and blocking constraints occurs at point $\bar{t}$ as follows:

$$s_{i'',1} + p_{i'',1} \leq s_{i',2} \leq s_{i,2} \leq s_{i'',1}.$$

It can be observed that such infeasible operation sequences similarly appear as cycles in the alternative graph representation. Thus, treating swaps as feasible results in a need to differentiate feasible and infeasible cycles when encoding BJSP schedules by an alternative graph. Following findings presented in [39] on a weighted graph representation, a simple structural property to contrast feasible swap cycles from infeasible sequencing cycles can be proposed. An alternative graph represents a feasible schedule for the BJSP, if all cycles involved do only consist of operations of different jobs requiring different machines, cf. [15]. The operations forming the cycle and featuring their start of processing at the same point in time are called a swap group. The given property facilitates the interchange of two or more jobs on a subset of machines, since it assures that every machine required by an operation of the swap group is currently occupied by the job predecessor of another operation of the group. Comparing the cycles in Figure 1, it can be seen that the arcs involved in the feasible swap cycle in part (a) feature different patterns, since the operations at their heads require different machines. On the contrary, two of the arcs forming the infeasible cycle in part (b) are solid arcs indicating that the two involved operations $O_{i'',1}$ and $O_{i',2}$ require the same machine.

Since this work relies on permutation-based encodings of schedules and corresponding feasibility checking procedures, the concept of swap groups is used to handle feasible cyclic dependencies. In the previous section, it is already mentioned that relations between operations on different machines can only be included in the operation-based representation of a schedule. Thus, the appearance of a swap is implemented in a single list by forming a swap group of operations which is assigned to one single list index. This list index fulfills the existing processing sequence and blocking constraints of all involved operations, and indicates that these operations will also feature a common starting time in the schedule. Considering the small general example given in part (a) of Figure 1, an operation-based representation of this partial schedule may result in $s^{op} = [\ldots, O_{i,1}, O_{i',1}, (O_{i,2}, O_{i',2}), O_{i'',1}, \ldots]$.

4.3. Feasibility Guarantee

In the following, the feasibility of a schedule given by its operation-based representation shall be examined more closely. As mentioned before, the processing sequences of the jobs and the blocking constraints can be translated to required list index relations of pairs of operations in the permutation.

For two consecutive operations $O_{1,j}$ and $O_{1,j'}$ of a job J_1 with $j < j'$, the starting time constraint $s_{1,j} + p_{1,j} \leq s_{1,j'}$ has to be fulfilled by a feasible schedule. Since these starting times are derived from the ordering of the operations in the permutation-based encoding, the required processing sequence can easily be implemented by assuring $lidx(O_{1,j}) + 1 \leq lidx(O_{1,j'})$. Blocking constraints can be described using list indices following the same pattern. Assume that, besides the operations $O_{1,j}$ and $O_{1,j'}$ requiring two machines M_k and $M_{k'}$, respectively, there is another operation $O_{2,j''}$ requiring machine M_k. If $O_{1,j} \rightarrow O_{2,j''}$ is determined as the operation sequence on this machine, the absence of intermediate buffers causes the starting time constraint $s_{1,j+1} \leq s_{2,j''}$. Translating this blocking restriction to a list index constraint implies that the list index of the job successor of the machine predecessor of an operation needs to be smaller than the list index of the operation. Formally, for two operations $O_{i,j}$ and $O_{i',j'}$ of different jobs requiring the same machine and a given operation sequence $O_{i,j} \rightarrow O_{i',j'}$, the following list index relation has to be fulfilled by a feasible permutation:

$$lidx(O_{i,j+1}) + 1 \leq lidx(O_{i',j'}), \tag{3}$$

provided that the operation $O_{i,j+1}$ exists.

This type of list index constraints constitutes the basis of checking and retrieving the feasibility of a BJSP schedule given by a single list of operations. The proposed method, called the *Basic Repair Technique (BRT)*, takes any permutation *perm*, which is feasible with regard to the processing sequences of the jobs, as input and constructs the operation-based representation s^{op} of a feasible schedule for the BJSP, cf. [15,16]. Note that the different terms *perm* and s^{op} both describing an operation-based encoding of the schedule are only used for reasons of clarification here. Figure 2 outlines the algorithm.

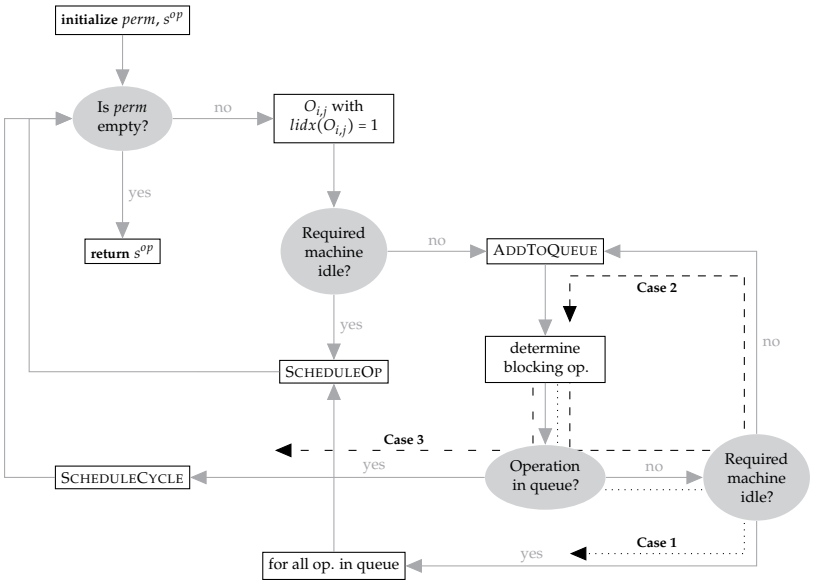


Figure 2. Schematic outline of the Basic Repair Technique (BRT), cf. [15].

A permutation *perm*, from which a feasible schedule is to be constructed, is initially given. The return permutation s^{op} is initialized with an empty list. The basic strategy of the BRT is to iterate over the list *perm*, take at least one operation from this list in each iteration and place it in the list s^{op}, so that all BJSP constraints are satisfied. As long as *perm* is not empty, the operation $O_{i,j}$ at the first list index is considered in the current iteration. If the required machine $Ma(O_{i,j})$ is idle, meaning that there is no other operation blocking it, the function ScheduleOp is called on operation $O_{i,j}$. This function

- determines and stores the earliest possible starting time of the considered operation,
- removes the operation from *perm*,
- adds the operation to the next idle list index in s^{op}, and
- sets the status of $Ma(O_{i,j})$ to blocked provided that a job successor $O_{i,j+1}$ exists.

With this, operation $O_{i,j}$ is said to be scheduled and the algorithm continues with the next iteration.

In case the required machine $Ma(O_{i,j})$ is not idle, meaning that it is blocked by another operation, the currently considered operation is involved on the right-hand side of a blocking constraint as given in (3). The operation $O_{i,j}$ is added to a queue and the operation required at a smaller list index to satisfy the blocking constraint is determined. The required operation is denoted as $O_{a,b}$ in the following. At this point, the BRT proceeds according to one of three different paths indicated in Figure 2.

Case 1: If the operation $O_{a,b}$ is not involved in the queue and its required machine $Ma(O_{a,b})$ is idle, operation $O_{a,b}$ and all operations in the queue are scheduled following a last in-first out strategy. Note that, when Case 1 is singly executed, exactly two operations are transferred from *perm* to the new operation-based representation s^{op}.

Case 2 → Case 1: If operation $O_{a,b}$ is not involved in the queue but its required machine $Ma(O_{a,b})$ is not idle, operation $O_{a,b}$ is added to the queue and the next required operation to fulfill the occurring blocking constraint is determined. Operations are added to the queue according to Case 2 until a required operation with an idle machine is found. Then, Case 1 is executed and all queuing operations are scheduled. Note that following this path, at least three operations are transferred from *perm* to s^{op}.

Case 2 → Case 3: Equivalent to the previous path, the operation $O_{a,b}$ is added to the queue and the next required operation is determined. Case 2 is executed until an operation already present in the queue is found. This refers to the situation where a cyclic dependency of blocking constraints exists and a swap needs to be performed in the schedule. The swap group is defined by all operations in the queue added in between the two occurrences of the currently considered operation. Following case 3, all operations of the swap group are scheduled with equivalent starting times and potentially

remaining operations in the queue are scheduled correspondingly after. Since the smallest possible swap cycle is formed by two operations, at least two operations are transferred from $perm$ to s^{op} when this path is executed.

With this, the BRT captures all occurring dependencies in arbitrary permutation-based encodings of BJSP schedules. The method assures that all blocking constraints are fulfilled by shifting required operations to positions with smaller list indices and by forming and modifying swap groups. Given that the number of operations in the problem is finite and the initially given permutation is feasible with respect to the processing sequences of the jobs, the following proposition holds, cf. [15].

Proposition 1. *The Basic Repair Technique (BRT) terminates and constructs an operation-based representation s^{op} of a feasible schedule for the BJSP.*

Proof. It has to be shown that

(1) the resulting permutation s^{op} is feasible with regard to the processing sequences of all jobs $J_i \in \mathcal{J}$,
(2) the resulting permutation s^{op} is feasible with regard to blocking constraints and
(3) every operation $O_{i,j} \in \mathcal{O}$ is assigned to a position in the feasible permutation s^{op} exactly once.

An unsatisfied blocking constraint $s_{i',j'} \leq s_{i,j}$ is detected in the BRT while an operation $O_{i',j'-1}$ is already scheduled in the feasible partial permutation s^{op} and operation $O_{i,j}$ is the currently considered operation, for which $Ma(O_{i,j})$ is not idle. The BRT shifts required operation(s), here only operation $O_{i',j'}$, to the next idle position $lidx'(*) > lidx'(O_{i',j'-1})$ and will never affect list indices prior to or equal to $lidx'(O_{i',j'-1})$. Hence, a given feasible ordering accounting for processing sequences and technological routes, such as $lidx(O_{i',j'-1}) < lidx(O_{i',j'})$, can never be violated by changes in the operation sequences made to fulfill blocking constraints. (1) is true.

The initially empty permutation s^{op} is expanded iteratively in the BRT. Every time an operation $O_{i,j}$ is considered to be assigned to the next idle list index $lidx'(*)$, unsatisfied blocking constraints are detected and fulfilled. Accordingly assigning an operation $O_{i',j'}$ to the list index $lidx'(*)$ in s^{op} prior to its initially given index $lidx(O_{i',j'})$ in $perm$ may implement a change in the operation sequence on the concerned machine. This may only cause new blocking constraints referring to the positions of the job successor $O_{i',j'+1}$ and the machine successor of operation $O_{i',j'}$. Due to given feasible processing sequences, affected operations cannot be part of the current partial permutation s^{op} and unsatisfied blocking constraints do only arise in the remainder of the permutation $perm$. Thus, it is assured that the existing partial permutation s^{op} is feasible with regard to blocking constraints in every iteration. Since this remains true until the BRT terminates, (2) is shown.

The consideration of operations in the BRT follows the ordering given in the initial list $perm$ starting from the first position. Since the assignment of an operation $O_{i,j}$ to the next idle list index $lidx'(*)$ in s^{op} may only affect constraints that are related to succeeding operations in the initial list $perm$, the necessity of a repeated consideration of an operation can never occur, once it is added to the feasible ordering s^{op}. Therefore, (3) is true. □

Considering the remarks on the numbers of operations scheduled in every iteration of the BRT, it can already be expected that a feasible schedule is determined by the BRT in polynomial time. In [15], it is shown in detail that the schedule construction takes $O(n_{op} \cdot m)$ steps. Thus, the BRT is an appropriate basic tool to be applied in heuristic search schemes.

4.4. Distance of Schedules

The distance of feasible solutions is an important measure in analyzing search spaces and neighborhood structures of discrete optimization problems, cf. [43,44]. When a heuristic search method is applied, the distance of two consecutively visited solutions refers to the size of the search step. In such a procedure, the step size may act as a control parameter or observed key measure

to guide the search. Intensification and diversification are strategically implemented by conducting smaller or bigger steps to avoid an early entrapment in locally optimal solutions.

In scheduling research, the distance $\delta(s, s')$ of two feasible schedules s and s' is commonly defined by the minimum number of basic operators required to construct one schedule from the other, cf. [44,45]. Here, the adjacent pairwise interchange (API) of two neighboring operations in the machine-based representation of a schedule is used as the basic operator. Formally, it can be described by the introduction of an indicator variable for all pairs of operations $O_{i,j}$ and $O_{i',j'}$ with $i < i'$ requiring the same machine as follows:

$$h_{i,j,i',j'} = \begin{cases} 1, & \text{if an ordering } O_{i,j} \to O_{i',j'} \text{ or } O_{i',j'} \to O_{i,j} \text{ in } s \text{ is reversed in } s', \\ 0, & \text{else.} \end{cases} \tag{4}$$

Consequently, the distance of two schedules is determined by

$$\delta(s, s') = \sum_{\substack{O_{i,j}, O_{i',j'} \in \mathcal{O} \text{ with} \\ Ma(O_{i,j}) = Ma(O_{i',j'}),\ i < i'}} h_{i,j,i',j'} \ . \tag{5}$$

Note that, when describing the BJSP with a mixed-integer program and implementing the pairwise ordering of operations with binary variables, the given distance measure is highly related to the well-known Hamming distance of binary strings, see [15] for further explanations.

5. Neighborhood Structures

In the following, neighborhood structures, which apply interchanges and shifts to the permutation-based representations of a schedule, are defined. The generation of feasible neighbors receives special attention, and the connectivity of the neighborhoods when dealing with complex BJSPT instances is discussed. A statistical analysis of a large set of generated neighboring solutions is reported to detect critical characteristics of the repairing scheme and the search space in general.

5.1. Introducing Interchange- and Shift-Based Neighborhoods

5.1.1. Transition Schemes and Their Implementation

In line with the findings presented in the literature, intensification and diversification shall both be realized in a heuristic search procedure by appropriate moves. When solving general sequencing problems, interchanges and shifts of elements in permutations constitute generic operators which are widely used, cf. [45,46]. The interchange-based moves applied here to the BJSPT and their implementation in the permutation-based encodings are defined as follows, see [15,16].

Definition 1. *An **API move** denotes the interchange of two adjacent operations $O_{i,j}$ and $O_{i',j'}$ of different jobs requiring the same machine $M_k \in \mathcal{M}$ in the machine-based representation of the schedule. Adjacency is defined in a strict sense. A pair of operations $O_{i,j}$ and $O_{i',j'}$ is called **adjacent** if there is no idle time on machine M_k between the preceding operation leaving the machine and the start of the processing of the succeeding operation.*

Definition 2. *A **TAPI move** denotes an interchange of two adjacent operations $O_{i,j}$ and $O_{i',j'}$ of different jobs requiring the same machine $M_k \in \mathcal{M}$ with $O_{i,j} \to O_{i',j'}$ in the machine-based representation of the schedule, where strict adjacency is given and the job $J_{i'}$ is currently tardy.*

The limitation to pairs of operations, which are strictly adjacent in a schedule, can be made without loss of search capability, cf. [15]. An idle time between two consecutively processed operations of different jobs on a machine may only occur due to

1. the technological routes of the jobs and the corresponding processing sequences on other machines or
2. the release date of the job of the succeeding operation.

In the first case, there always exists a sequence of applicable API moves that eliminates the idle time and enables an interchange of the considered pair of operations. In the second case, an interchange of the considered pair of operations will only result in postponing the starting time of the initially preceding operation, since the succeeding operation cannot be processed earlier. If such a postponement is beneficial, this will also be indicated by an applicable API at another point in the schedule. Otherwise, postponing the preceding operation can never be advantageous with regard to total tardiness.

These operators are intended to construct close neighboring solutions with a desired distance $\delta(s, s') = 1$. Small steps are supposed to intensify the search and make a heuristic search procedure nicely tractable towards locally optimal schedules. The advantageousness of restricting the set of potential API moves based on the objective function value, namely considering only TAPI moves, shall be closely investigated in the computational experiments. Figure 3 shows all applicable API moves (solid arrows) and TAPI moves (dashed arrows) for a small BJSPT instance with three machines and three jobs. It can be observed that, referring to the same schedule, the set of TAPI moves is a subset of the set of API moves. Note that there is an idle time occurring between three pairs of consecutively processed operations on the machines M_2 and M_3, while there is no blocking time on any machine in the given schedule.

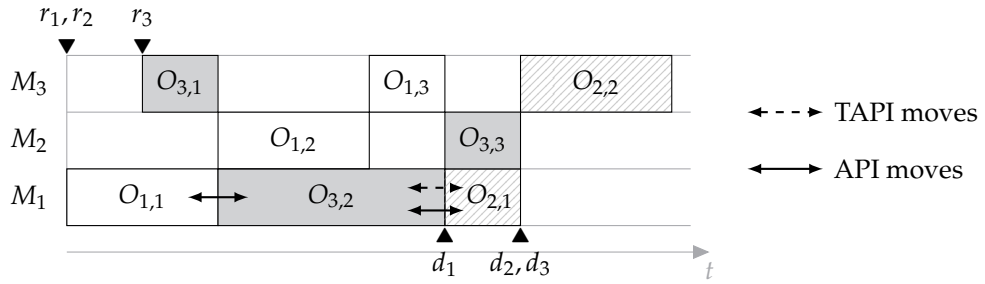


Figure 3. Illustration of applicable API and TAPI moves in a given schedule for the BJSPT

To avoid redundancy in the neighborhood structure on the one hand, API moves are applied to the machine-based representation of a schedule. To check the feasibility of the constructed neighboring schedule on the other hand, the applied API move needs to transferred to the operation-based representation of the schedule, cf. [16]. Since the interchanged operations are not necessarily directly adjacent in the permutation, this can be done by a left shift or a right shift transformation, respectively. First, consider the API move $O_{1,1} \leftrightarrow O_{3,2}$ in the schedule given in Figure 3 and the following permutation encoding this schedule:

$$s^{op} = [\underline{O_{1,1}}, O_{3,1}, O_{1,2}, \underline{O_{3,2}}, O_{1,3}, O_{3,3}, O_{2,1}, O_{2,2}].$$

The API move can be implemented either by shifting operation $O_{1,1}$ to the right together with its job successor $O_{1,2}$ to preserve the processing sequence or by shifting operation $O_{3,2}$ to the left together with its job predecessor $O_{3,1}$. In both cases, the following permutation *perm* is generated:

$$perm = [O_{3,1}, \underline{O_{3,2}}, \underline{O_{1,1}}, O_{1,2}, O_{1,3}, O_{3,3}, O_{2,1}, O_{2,2}].$$

It appears that the permutations generated by implementing API moves are infeasible with regard to blocking constraints. Here, operation $O_{1,1}$ cannot be scheduled on machine M_2, since this machine is blocked by operation $O_{3,2}$. Thus, the given list needs to be repaired. After applying the BRT to

perm, the following feasible neighboring schedule $s^{op'}$, which incidentally turns out as a permutation schedule, is constructed:

$$s^{op'} = [O_{3,1}, \underline{O_{3,2}}, O_{3,3}, \underline{O_{1,1}}, O_{1,2}, O_{1,3}, O_{2,1}, O_{2,2}].$$

Considering the second applicable API move $O_{3,2} \leftrightarrow O_{2,1}$ in the schedule of the $(3,3)$-instance in Figure 3, the left shift of operation $O_{2,1}$ and the right shift of operation $O_{3,2}$ in s^{op} generate two different permutations $perm_1$ and $perm_2$, respectively:

$$perm_1 = [O_{1,1}, O_{3,1}, O_{1,2}, \underline{O_{2,1}}, \underline{O_{3,2}}, O_{1,3}, O_{3,3}, O_{2,2}],$$
$$perm_2 = [O_{1,1}, O_{3,1}, O_{1,2}, O_{1,3}, \underline{O_{2,1}}, \underline{O_{3,2}}, O_{3,3}, O_{2,2}].$$

Applying the BRT to $perm_1$ constructs a feasible neighboring schedule

$$s_1^{op'} = [O_{1,1}, O_{3,1}, O_{1,2}, \underline{O_{2,1}}, (O_{2,2}, \underline{O_{3,2}}), O_{1,3}, O_{3,3}],$$

which is displayed in Figure 4. This schedule features a swap of the jobs J_2 and J_3 on the machine M_1 and M_3 and two periods of blocking time on the machines M_2 and M_3 indicated by the curved lines.

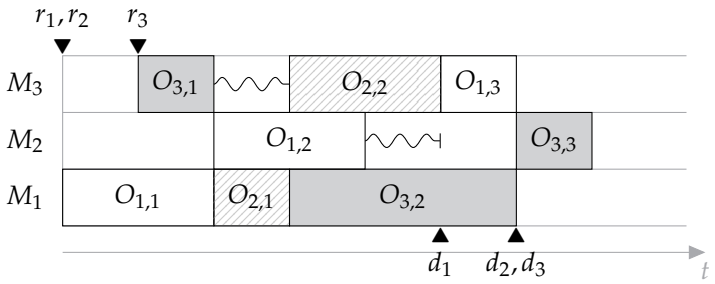


Figure 4. Illustration of the feasible neighboring schedule $s_1^{op'}$ resulting from an API move.

Applying the BRT to $perm_2$ reveals a major difficulty of using permutations, interchange-based operators, and repairing schemes in solution approaches for BJSPs. After the first three operations have been added to the partial permutation $s_2^{op'} = [O_{1,1}, O_{3,1}, O_{1,2}]$, the operation $O_{1,3}$ is considered. It requires machine M_3, which is blocked by operation $O_{3,1}$. Thus, the job successor $O_{3,2}$ must be scheduled prior to operation $O_{1,3}$, and the BRT reverts the given API to regain the feasibility of the schedule. A graphical representation of the critical step is given in Figure 5.

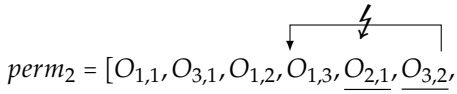


Figure 5. Schematic presentation of an API reverted by applying the BRT.

It can easily be seen that the operation sequence given in the first part of the permutation $perm_2$ and the operation sequence resulting from the API cannot be implemented together. Additional changes in the schedule are necessary to construct a feasible solution involving the desired ordering $O_{2,1} \rightarrow O_{3,2}$. Preliminary experiments have shown that a reversion occurs in 80 to 90% of all generated and repaired neighboring schedules. Therefore, an enhanced repairing scheme is required to find feasible solutions that contain given orderings while featuring as few changes as possible compared to the initially given ones.

The desired operation sequence can be interpreted as a partial schedule with exactly two elements. While the decision problem on the existence of a completion of an arbitrarily large partial schedule is NP-complete for the BJSP, cf. [17], the generation of a feasible schedule with a given ordering of

exactly two operations is always possible. Nonetheless, the challenging task is to find a structured and commonly applicable procedure, which returns a feasible neighbor from an initially given schedule and a desired operation sequence resulting from an API. The subsequent section deals with this issue in detail.

It is indicated by previous studies on BJSPs that a diversification strategy is beneficial to reach promising regions of the search space, see [32,34]. Therefore, a randomized and objective function-oriented transition scheme is defined. It is applied to the operation-based representation of a schedule and relies on shifts of operations in the permutation as generic operators, see [15,16].

Definition 3. *A **TJ move** is defined by applying random leftward shifts to all operations of a tardy job J_i in the permutation-based representation of a schedule, while preserving the processing sequence $O_{i,1} \rightarrow O_{i,2} \rightarrow \cdots \rightarrow O_{i,n_i}$ of the job.*

The resulting permutation might be infeasible with regard to blocking constraints, and the BRT is used to construct a feasible neighboring schedule. Since a TJ move creates desired partial sequences for every shifted operation, it is not guaranteed that a solution involving all of these orderings simultaneously exists. Thus, no fixation can be applied and the BRT is potentially able to revert all shifts. To avoid neighboring schedules which are equivalent to the initially given ones, sufficiently large shifts are executed.

5.1.2. Generating Feasible API-Based Neighbors

As mentioned in the previous section, the generation of feasible neighboring schedules for BJSPs involving a given API-based ordering is a critical issue. Since potentially required additional changes in the schedule are not contained in the BRT, an *Advanced Repair Technique (ART)* is proposed, cf. [15,16]. This method takes the operation-based representation of a schedule s, named *perm*, and a desired sequence of two operations $O_{a,b} \rightarrow O_{i',j'}$ and returns the operation-based representation s^{op} of a neighboring schedule s', which involves the given ordering. All additional APIs necessary to transform the schedule s into s' follow a basic rule. Instead of reverting a given pairwise sequence $O_{a,b} \rightarrow O_{i',j'}$, the initial permutation is adapted by leftward interchanges of an operation of the job J_a and the repairing scheme is restarted. Figure 6 gives a schematic illustration of the ART in total.

It can be observed in the left part of the chart that the BRT constitutes the foundation of the ART. The operations are iteratively taken from the list *perm*, requiring machines are checked for idleness, and blocking operations are determined, if necessary. The first important difference is indicated by the ellipsoid node printed in bold face. An operation is defined to be fixed, if it acts as the successor operation in a given pairwise sequence. The corresponding predecessor is denoted as the associated operation. In the general example stated above, operation $O_{i',j'}$ is fixed with the associated operation $O_{a,b}$. In case a fixed operation shall be added to the queue and its associated operation is already scheduled in the feasible permutation s^{op}, no reversion occurs and the procedure continues in the basic version. In case the associated operation $O_{a,b}$ is not yet scheduled, the given ordering $O_{a,b} \rightarrow O_{i',j'}$ would be reverted by adding operation $O_{i',j'}$ to the queue. To avoid this, the ART follows one of four modification paths in the gray box, the initial list *perm* is adapted, and the whole procedure is restarted.

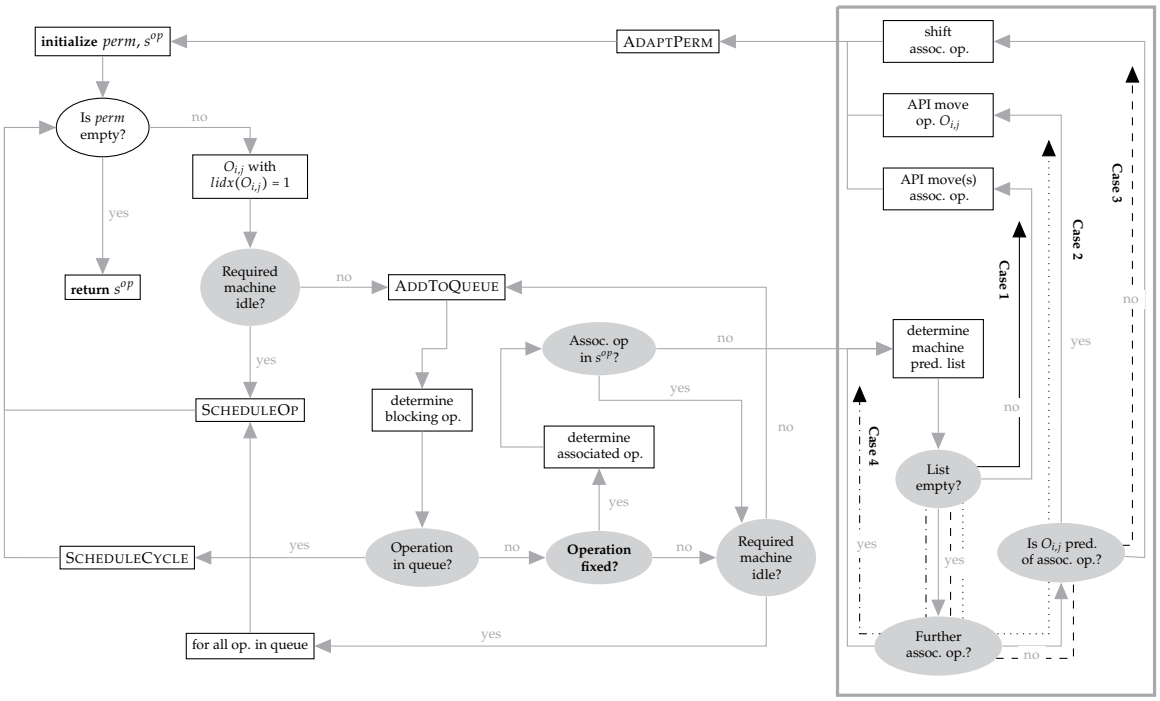


Figure 6. Schematic outline of the Advanced Repair Technique (ART), cf. [15].

Figure 7 illustrates the four possible cases of adaptation by means of Gantt charts. Assume that $O_{a,b} \rightarrow O_{i',j'}$ is a given ordering and operation $O_{i,j}$, shown in striped pattern, is the operation currently considered by the ART. Irreversible pairwise sequences are given by bold rightward arrows connecting adjacent operations on a machine, and the required additional APIs are indicated by leftward arrows with case-corresponding line patterns, see Figure 6. Let the feasible partial schedule s^{op} already contain operation $O_{i',j'-1}$ in the cases 1, 2, and 3, and operation $O_{i',j'-2}$ in case 4. The consideration of operation $O_{i,j}$ to be scheduled next requires the following blocking constraint to be fulfilled: $lidx(O_{i',j'}) < lidx(O_{i,j})$. Since the associated operation $O_{a,b}$ is not yet included in the list s^{op}, the fixed operation $O_{i',j'}$ cannot be positioned at the next idle list index prior to operation $O_{i,j}$. To resolve the situation, the basic strategy is to shift or interchange the associated operation $O_{a,b}$ further to the left on its machine, so that it will be scheduled before the required repairing shift of operation $O_{i',j'}$ occurs in the next run of the procedure. Therefore, the machine predecessor list of $O_{a,b}$ excluding operations of job J_a is determined, see Figure 6, and the adaptation of the initial permutation is conducted according to one the following cases:

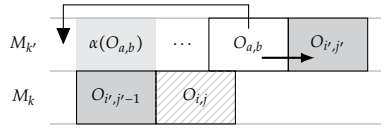
Case 1: If there exists a machine predecessor $\alpha(O_{a,b})$, an additional API move is performed and the ordering $O_{a,b} \rightarrow \alpha(O_{a,b})$ is defined to be fixed additionally, see part (a) of Figure 7. The API is implemented in the list *perm* by a left shift of operation $O_{a,b}$.

Case 2: If there exists no machine predecessor of operation $O_{a,b}$ and there exists no other operation associated with operation $O_{i',j'}$, the currently considered operation $O_{i,j}$ might itself be a job predecessor $O_{a,b'}$ of the associated operation $O_{a,b}$. If this is true, an API move is performed with its machine predecessor $O_{i',j'-1}$, see part (b) of Figure 7. Note that, for this situation to occur, the machine predecessor necessarily needs to exist and belong to job $J_{i'}$.

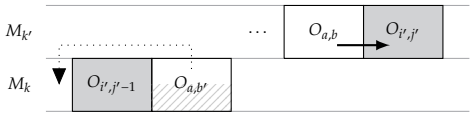
Case 3: Assume that there exists no machine predecessor of operation $O_{a,b}$ and no other operation associated with operation $O_{i',j'}$, and, furthermore, the currently considered operation is not a job predecessor of operation $O_{a,b}$. Then, the associated operation is shifted leftward in the permutation *perm* to the position prior to the currently considered operation $O_{i,j}$, see part (c) of Figure 7. This shift does not implement an API move but is sufficient to satisfy the given blocking constraint.

Case 4: This situation differs structurally from the other three cases. It involves at least three machines, and it can only appear with operations of recirculating jobs after one or more additional APIs

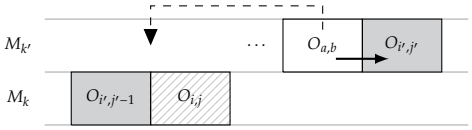
have already been performed. In part (d) of Figure 7, besides the initially given ordering, the pairwise sequences $O_{a,b'} \rightarrow O_{i',j'-1}$ and $O_{a,b''} \rightarrow O_{i',j'-1}$ are exemplarily fixed. After the machine predecessor list of the associated operation $O_{a,b'}$ has been determined as empty, a second operation associated with the fixed operation $O_{i',j'-1}$ can be found, namely operation $O_{a,b''}$. Dependent on the existence of a machine predecessor $\alpha(O_{a,b''})$, the ART proceeds according to Cases 1, 2, or 3 with an adaptation of the list *perm*. In the depicted Gantt chart, a shift following Case 1 is shown as an example.



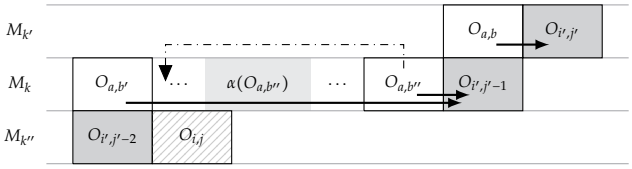
(**a**) **Case 1:** Shifting the associated operation $O_{a,b}$ in the operation sequence on $Ma(O_{a,b})$



(**b**) **Case 2:** Shifting the currently considered operation $O_{i,j} = O_{a,b'}$, $(b' < b)$ in the operation sequence on $Ma(O_{a,b'})$



(**c**) **Case 3:** Shifting the associated operation $O_{a,b}$ in *perm*



(**d**) **Case 4:** Shifting further associated operations like operation $O_{a,b''}$, $(b' < b'' < b)$ of the fixed blocking operation $O_{i',j'-1}$ in the operation sequence on $Ma(O_{a,b''})$

Figure 7. Adapting the permutation *perm* in the ART, cf. [15].

After the permutation is adapted according to one of the four cases, the ART is restarted involving one additionally fixed pairwise sequence. The following observation can be made regarding the operations moved during adaptation.

Observation 1. *While executing the ART, the operation interchanged or shifted leftwards when adapting the permutation is always the associated operation $O_{a,b}$ defined by the initially given fixed sequence $O_{a,b} \rightarrow O_{i',j'}$ or one of its job predecessors.*

Based on this, arguments indicating the correctness of the ART can be derived as follows, cf. [15]:

Proposition 2. *The ART terminates and returns a permutation s^{op} encoding a feasible schedule for the BJSP involving a predefined ordering $O_{a,b} \rightarrow O_{i',j'}$ of two operations of different jobs requiring the same machine.*

Proof. It is equivalently assumed here that the initially given list *perm* is feasible with regard to the processing sequences of all jobs $J_i \in \mathcal{J}$. The ART proceeds like the BRT until there is a fixed operation $O_{i',j'}$ to be scheduled prior to its associated operation $O_{a,b}$. According to Proposition 1, the BRT terminates and returns a feasible encoding s^{op} of a BJSP schedule from a given permutation. Consequently, the adaptation of the permutation and the restart of the ART are the only critical aspects to regard **here** in detail.

It needs to be shown that

(1) an adaptation does not violate the processing sequences of the jobs,
(2) an adaptation can never be reverted,
(3) the number of possible adaptations is finite and
(4) there exists a sequence of adaptations leading to a feasible schedule for the BJSP including the predefined pairwise sequence $O_{a,b} \rightarrow O_{i',j'}$.

When an adaptation is performed, an operation of job J_a is shifted leftwards in the permutation. The processing sequence of this job is the only one potentially affected and it may only get violated,

if the operation is moved prior to one or more of its job predecessors. This situation is checked during the adaptation and job predecessors are additionally shifted, if necessary. Thus, (1) is always true.

The set of irreversible orderings is extended by one pairwise sequence in every execution of the adaptation procedure. Thus, the incorporated BRT mechanisms can never reverse an adaptation. A consecutively required API or shift can only result from a fixed ordering $O_{a,b'} \rightarrow O_{c,d}$ with $b' \in \{1, \ldots, b\}$, where the currently regarded operation $O_{i,j}$ features a list index prior to $lidx(O_{a,b'})$ in $perm$. This means that a consecutively required adaptation does always appear at a position prior to the previous adaptation causing the fixed sequence $O_{a,b'} \rightarrow O_{c,d}$. As a consequence, the operation $O_{a,b'}$ or one of its job predecessors is moved to a list index smaller than $lidx(O_{i,j})$, for which $lidx(O_{i,j}) < lidx(O_{a,b'}) < lidx(O_{c,d})$ holds. Thus, an implemented adaptation can never be reverted by an ART mechanism. (2) is true.

Since the list $perm$ contains a finite number of elements, and the set of shifted operations is restricted to all operations of a job $J_a \in \mathcal{J}$, see Observation 1, and (2) is true, the number of possible adaptations is finite. (3) holds.

The strategy of the ART can be summarized as shifting the operations of a job J_a iteratively leftwards in the operation sequences on the required machines. This is repeatedly applied until the given pairwise sequence is realized in a feasible schedule for the BJSP constructed by BRT mechanisms only. Following from Observation 1 and statement (2), all moved operations may end up at the first positions in the operation sequences on their machines in the extreme case. Thus, the job J_a involving operation $O_{a,b}$ is scheduled prior to all other jobs involved in the problem and $O_{a,b} \rightarrow O_{i',j'}$ is guaranteed. (4) is shown. □

Considering that the total number of operations is given by n_{op} and, furthermore, taking this measure as a worst case estimate for the number of operations requiring a certain machine, the ART determines a feasible schedule involving a given pairwise sequence in $O\left((n_{op})^4\right)$ steps, cf. [15]. This method enables the usage of APIs as generic operators in neighborhood structures for BJSPs.

5.1.3. Definition of the Neighborhoods

In line with the transition schemes described in the previous section, the examined neighborhoods are defined as follows:

Definition 4. *The **API neighborhood** of a schedule s is defined as the set of schedules s', where s' is a feasible schedule involving a given API move implemented by a left shift or a right shift.*

Definition 5. *The **TAPI neighborhood** of a schedule s is defined as the set of schedules s', where s' is a feasible schedule involving a given TAPI move implemented by left shift or right shift.*

Definition 6. *The **TJ neighborhood** of a schedule s is defined as the set of schedules s', where s' is a feasible schedule resulting from a TJ move.*

In the following, all neighboring solutions constructible through a given API or TAPI are generally denoted as API-based neighbors of a schedule. Note that, due to the required repairing schemes, the actual distances of neighboring schedules are not precisely determined, cf. [15]. The minimum distance of a schedule and its API-based neighbor is given by 1, and the maximum distance is theoretically bounded by $\sum_{M_k \in \mathcal{M}} \binom{|\Omega^k|}{2}$, where Ω^k defines the set of operations requiring machine M_k. As mentioned in the previous section, schedules in the TJ neighborhood might have a minimum distance of 0 to the initially given one, while the maximum distance is equivalently restricted by the structural upper bound. While the leftward shifting strategy applied by the ART in the adaptation of the permutation is required for the termination of the method, it is not guaranteed that the smallest number of necessary changes is implemented. To the best of the authors' knowledge, there does not yet exist a general neighborhood structure or repairing scheme for BJSP schedules capable of certainly

constructing the closest possible neighbor for an initial solution and a given change. An empirical study on the distances resulting for the proposed neighborhoods together with the ART is reported in the next section.

5.2. Characteristics and Evaluation

5.2.1. Connectivity of the Neighborhoods

A neighborhood is said to be connected, if every existing feasible schedule can be transformed into every other existing feasible schedule by (repeatedly) applying a given neighbor-defining operator, see [13,39,45]. Here, the neighbor-defining operators consist of a move and a repairing scheme. The connectivity of the neighborhood is of significant importance in the application of search procedures, since it guarantees that the methods are capable of finding optimal solutions. However, such a structural result can only be interpreted as an indication for the actual performance of a neighborhood-based heuristic solution approach on practically relevant instances.

Proposition 3. *Given general release dates $r_i \in \mathbb{Z}_{\geq 0}$ for $J_i \in \mathcal{J}$ and the minimization of total tardiness as the optimization criterion, the proposed neighborhoods, namely the API neighborhood, the TAPI neighborhood and the TJ neighborhood, are* ***not connected****.*

Proof. Consider API and TAPI moves first. As described in Section 5.1, in a feasible schedule, there may exist two subsequent operations $O_{i,j}$ and $O_{i',j'}$ on a machine in the schedule which are considered as non-adjacent due to an idle time caused by the release date of the succeeding job $J_{i'}$. Such a pair of operations can never be chosen for an API or a TAPI move in constructing neighboring schedules. Thus, a schedule involving the ordering $O_{i',j'} \rightarrow O_{i,j}$ cannot be reached by applying the neighbor-defining operators even if it is feasible for the BJSP. Thus, the API and the TAPI neighborhoods are not connected.

Furthermore, regarding the TJ move which shifts all operations of a currently tardy job in the permutation, the limitation to choosing a job with a strictly positive tardiness value implies that the neighborhood of feasible schedules with a total tardiness of 0 is empty. Even if optimal solutions for the BJSPT are found in this case, these schedules are isolated by definition and no other feasible schedule can be constructed subsequently. Thus, the TJ neighborhood is not connected. □

Despite these negative findings on the connectivity of the neighborhoods, the proposed structures are still supposed to be successfully applicable in a metaheuristic search method for the BJSPT. It can be expected that extraordinarily widespread release dates, which cause a disjoint partitioning of the search space, do not occur in practically relevant problems. Furthermore, in most of the cases, it is not necessary to continue the search once an optimal solution is found.

However, the questions on whether the described API neighborhood is connected for the special case $r_i = 0$, $J_i \in \mathcal{J}$ or for a specific combination of release date and processing time ranges remain open. It is conjectured that the API neighborhood together with the ART feature the connectivity property for the BJSP without release dates.

5.2.2. Observations on the Interchange-Based Transition Scheme

Besides the general problem solving capability of a metaheuristic involving the proposed neighborhood structures, the API-based transition schemes shall be evaluated with regard to their ability to generate small distance and high quality neighbors. Special attention is given to the differences appearing in using a left shift or a right shift transformation to implement an API in the operation-based encoding of a schedule. For all API and TAPI neighbors constructed during the computational experiments, a specific interchange is chosen, left shift and right shift transformation are performed, and both resulting solutions are evaluated with regard to their distances from the initially given schedule and their total tardiness values.

Figure 8 displays the distributions of the distances of API-based neighbors with left shift (LS) and right shift (RS) transformation for the benchmark instances by boxplots. The range in which 50% of the distance measures of the neighbors can be found, the so-called interquartile range, is represented by the box. The black horizontal line indicates the median of the sample. The whiskers plot the minimal and the maximal distance value which are not more than one and a half interquartile ranges away from the box.

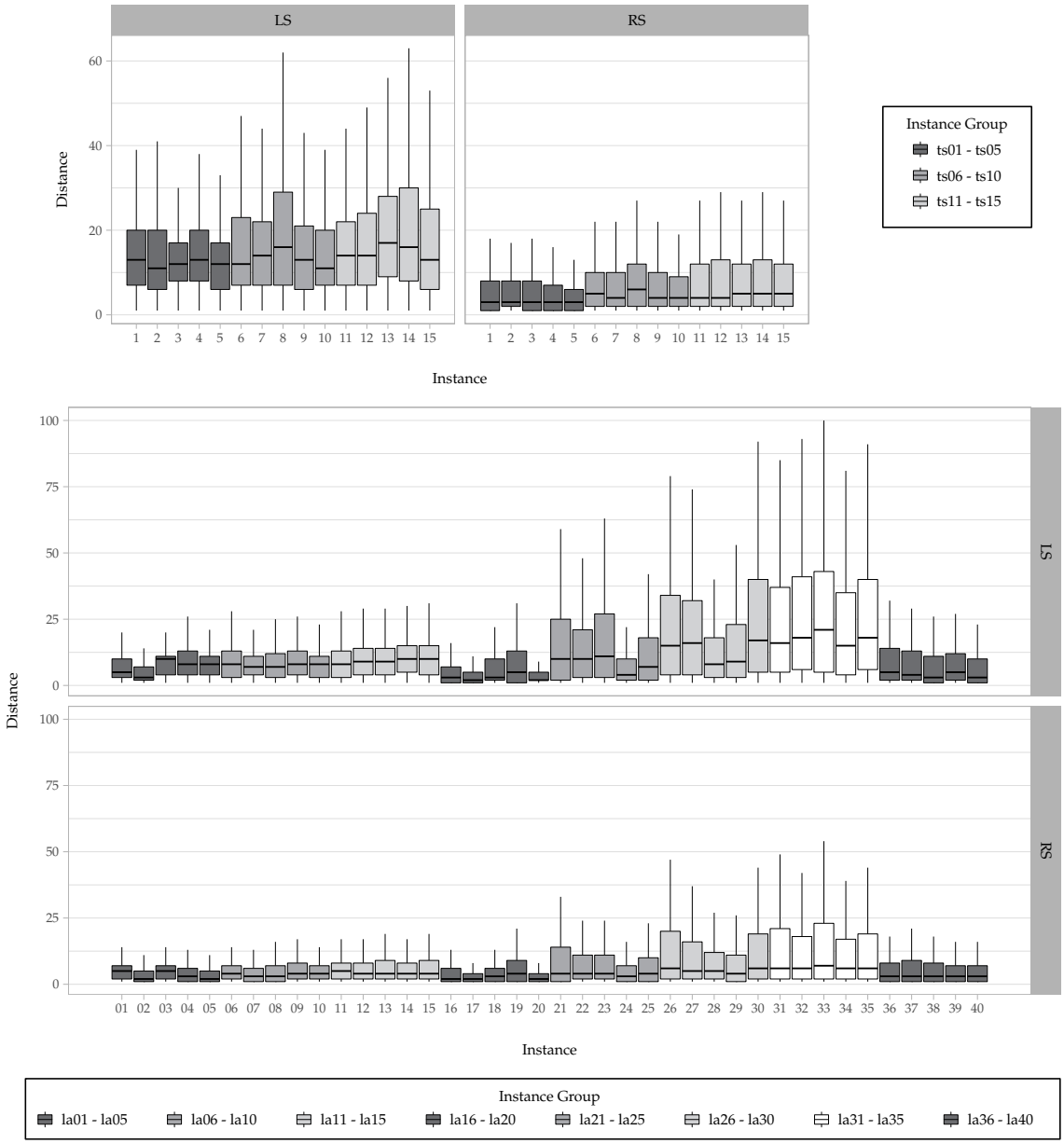


Figure 8. Boxplots representing the distribution of the distance measure among neighbors based on API-moves for the benchmark instances of the BJSPT, cf. [15].

Considering all transitions independent of the direction of implementation, it can be stated that the distance of neighboring schedules based on a single API is remarkably large for the BJSP. Evidently, a significant amount of additional adaptations is required to fit a given pairwise sequence to a feasible schedule. Even if it is not guaranteed that the closest neighbor is generated by the ART, these results highlight the complexity of the search space caused by blocking constraints. This may lead to difficulties in the effectiveness and the control of a heuristic search method, since an iterative execution

of small changes is desirable to systematically explore the set of feasible schedules. Comparing the directed implementations of APIs, the distances of neighbors constructed using a right shift transformation are significantly smaller than the measures of neighboring schedules generated by a left shift transformation. Thus, it is recommendable to implement APIs by a right shift in the operation-based encoding to support the execution of smaller search steps, cf. [15].

The chart in Figure 9 shows the proportion of APIs for which the neighbors resulting from a left shift and a right shift transformation are equivalent (EQ). Given that two different schedules arise, it is displayed to which extent the schedule with a smaller total tardiness value is generated by a left shift or by a right shift implementation of the API in the operation-based encoding. It can be observed that the majority of neighboring schedules based on the same API end up to be equivalent after applying the ART. Regarding the cases where different schedules are constructed, the right shift transformation clearly outperforms the left shift transformation by means of total tardiness. Since heuristic methods are intended to require limited computational effort, it is reasonable to implement APIs by right shift transformation only, cf. [15]. The analysis indicates that the proportion of cases in which the best possible neighboring schedule is not generated is less than 13%.

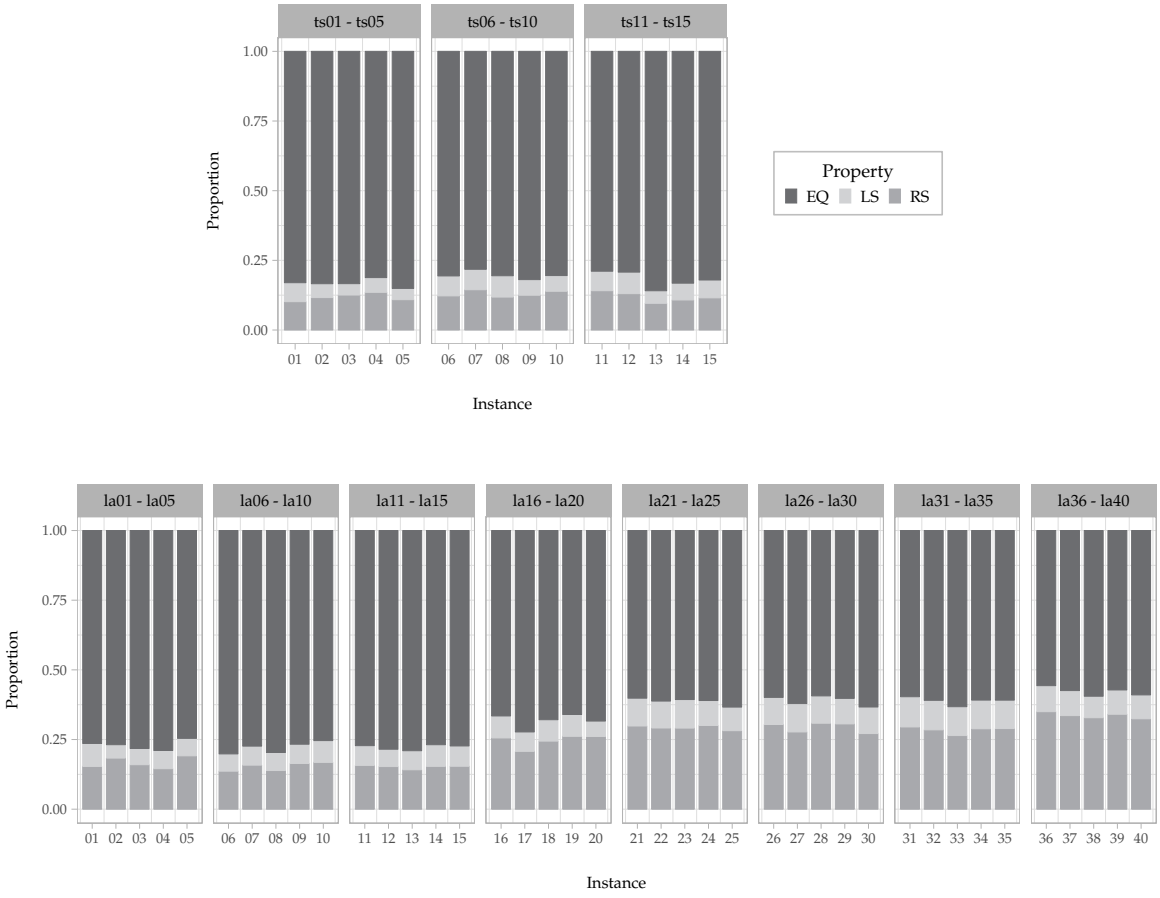


Figure 9. Proportion of equivalent solutions and performance of right shift and left shift transformation among all API-based neighbors for the benchmark instances of the BJSPT, cf. [15].

6. Computational Experiments and Results

Finally, the proposed neighborhood structures and repairing schemes are used to solve the benchmark instances of the BJSPT introduced in Section 3. In line with the findings in the literature, SA is chosen as a simple and generic local search scheme. The neighborhoods and repairing schemes can easily be embedded and the method facilitates moves to inferior neighboring solutions. The latter aspect seems especially promising with regard to the observed ruggedness of the search space of the

BJSPT. The following computational results give insight to the general capability of permutation-based procedures in solving the problem under study. Furthermore, the potential change in the performance when guiding the search scheme by objective function values is observed.

6.1. A Simulated Annealing Algorithm

The metaheuristic framework is implemented in the standard variant, see, for instance [2,38,39], with a geometric cooling scheme $t_{\tau+1} = c \cdot t_\tau$. Correspondingly, the initial temperature t_0, the terminal temperature T, and the cooling factor c act as the control parameters of the procedure. With regard to the asymptotic convergence of SA, $n_{op} - m$ neighboring solutions are evaluated for every temperature level.

Since the probability for a generated neighbor to be accepted as the new current solution depends on the objective function values of the considered schedules next to the temperature level, the parameter setting needs to be adjusted according to the magnitude of the total tardiness. Preliminary experiments indicated the following settings (t_0, T, c) as beneficial for the benchmark problems: (20, 0.5, 0.9925) and (20, 10, 0.999) for the train scheduling-inspired instances and (200, 50, 0.995) for the Lawrence instances, cf. [15,47]. Dependent on the size of the instances, 11,000 to 84,000 iterations are performed, cf. [15].

Furthermore, the extent to which the API-based and the randomized shift-based neighborhood structures are used to include intensification and diversification advantageously has been part of an initial study. The proposed algorithm applies either the API or the TAPI neighborhood combined with the TJ neighborhood, respectively. This implies that the effectiveness of an objective function-oriented guidance can be analyzed, while a random component is always involved. For every generation of a neighbor, the API-based neighborhood is chosen with a probability of 0.9 and a TJ neighbor is constructed with a probability of 0.1, cf. [15,47]. If an API is performed, both schedules resulting from a left shift and a right shift transformation are evaluated, and the superior one becomes the candidate to represent the next incumbent solution. The created algorithm is called permutation-based simulated annealing (PSA).

6.2. Numerical Results

The computational experiments are conducted on a notebook featuring an Intel Dual Core i5 processor (2.20 GHz) with 8 GB RAM. Algorithm PSA is implemented in Python 3. Tables 2–4 summarize the numerical results of five independent runs operated for each instance, parameter setting, and neighborhood structure. The first two columns of each table display the instance and the corresponding size (m, n). For reasons of comparison, the third column contains the best total tardiness value obtained by solving the considered problem with the help of IBM ILOG CPLEX 12.8 using a mixed-integer programming (MIP) formulation with pairwise precedence variables, see [15] for detailed explanations on the model. Objective function values with proven optimality are denoted by an asterisk. The next pairs of columns show the average total tardiness $\overline{\sum T_i}$ and the minimal total tardiness $\min(\sum T_i)$ obtained for each instance by Algorithm PSA based on the API and the TAPI neighborhood, respectively. Contrasting the mean total tardiness values reached, the smaller measure is highlighted by boldface printing.

Generally, it can be stated that Algorithm PSA yields satisfactory results for instances with and without inner structure especially when being compared to the MIP approach. For instances of small size and an equivalent number of jobs and machines, such as ts01 to ts05, la02 to la05, la07 and la08, la17, la19 and la20, the method is capable of finding an optimal solution. Even more important, the algorithm is able to generate medium quality solutions for large problems like la31 to la35, for which a general-purpose method might even struggle in generating feasible schedules. This gives evidence for the advantageousness of the proposed heuristic approach in solving real-world production planning instances of critical size.

Nonetheless, the complex repairing schemes constitute a drawback of the heuristic algorithm with regard to computation time. PSA requires 2 to 70 min of runtime dependent on the size of the instances, while the MIP technique is able to solve small instances to optimality in a few seconds, cf. [15]. As indicated by the statistical analysis of the neighborhoods in the previous section, the runtime of PSA can be improved by implementing APIs by a right shift transformation only. To further overcome these difficulties, a hybrid method combining heuristic and MIP mechanisms seems promising. Heuristic methods can be used to generate feasible schedules for large instances quickly, while solving smaller subproblems by MIP may be a superior improvement strategy towards locally optimal solutions. First, results following this research direction are presented in [15,18].

Table 2. Computational results of Algorithm PSA with $(20, 0.5, 0.9925)$ applied to the train scheduling-inspired instances, cf. [15].

Inst.	(m,n)	MIP	API		TAPI	
			$\overline{\sum T_i}$	$\min(\sum T_i)$	$\overline{\sum T_i}$	$\min(\sum T_i)$
ts01	(11,10)	138 *	**140.0**	138 *	142.6	138 *
ts02	(11,10)	90 *	**95.0**	91	96.6	90 *
ts03	(11,10)	72 *	**78.8**	72 *	84.8	76
ts04	(11,10)	41 *	41.4	41 *	**41.2**	41 *
ts05	(11,10)	71 *	**71.2**	71 *	71.6	71 *
ts06	(11,15)	88 *	125.0	108	**119.4**	109
ts07	(11,15)	172 *	**196.0**	184	201.0	192
ts08	(11,15)	163 *	185.6	163 *	185.6	181
ts09	(11,15)	153	**174.0**	160	175.2	161
ts10	(11,15)	97 *	116.6	107	**112.6**	108
ts11	(11,20)	366	**409.4**	387	411.8	392
ts12	(11,20)	419	**429.2**	412	442.4	419
ts13	(11,20)	452	492.2	472	**478.2**	445
ts14	(11,20)	459	**500.6**	473	508.8	492
ts15	(11,20)	418	433.2	413	**428.2**	387

Table 3. Computational results of Algorithm PSA with $(20, 10, 0.999)$ applied to the train scheduling-inspired instances, cf. [15].

Inst.	(m,n)	MIP	API		TAPI	
			$\overline{\sum T_i}$	$\min(\sum T_i)$	$\overline{\sum T_i}$	$\min(\sum T_i)$
ts01	(11,10)	138 *	140.2	138 *	**140.0**	138 *
ts02	(11,10)	90 *	**94.6**	91	95.2	91
ts03	(11,10)	72 *	**74.2**	72 *	74.4	72 *
ts04	(11,10)	41 *	41.8	41 *	**41.0** *	41 *
ts05	(11,10)	71 *	71.4	71 *	**71.0** *	71 *
ts06	(11,15)	88 *	121.6	107	**119.8**	111
ts07	(11,15)	172 *	195.4	189	**192.8**	185
ts08	(11,15)	163 *	**184.2**	179	185.0	181
ts09	(11,15)	153	178.8	168	**177.4**	174
ts10	(11,15)	97 *	114.8	97 *	**112.0**	105
ts11	(11,20)	366	406.4	390	**401.6**	387
ts12	(11,20)	419	428.2	412	**424.6**	405
ts13	(11,20)	452	462.6	448	**460.6**	447
ts14	(11,20)	459	**462.8**	418	495.0	466
ts15	(11,20)	418	**419.4**	401	435.0	414

Comparing the API and TAPI neighborhood with regard to solution quality over all instances, no transition scheme clearly dominates. An advantageousness of guiding the search by current

total tardiness values cannot be observed. A preliminary performance testing might be beneficial for every individual application of the API-based neighborhood structures to other BJSPT instances, since the solution quality reached may dependent on the problems size and structure as well as on the setting of the metaheuristic framework. It can be remarked that, based on the experiments on the ts instances with two different parameter settings, the API neighborhood performs better with lower temperature levels between 20 and 0.5, while the TAPI neighborhood is favorable combined with higher temperature levels between 20 and 10. This implies that simultaneously applying a strict limitation of the acceptance of inferior schedules in the search procedure and a restriction of the possible interchanges based on the objective function value is not reasonable.

Table 4. Computational results of Algorithm PSA with $(200, 50, 0.995)$ applied to the Lawrence instances, cf. [15].

Inst.	(m, n)	MIP	API		TAPI	
			$\overline{\sum T_i}$	$\min(\sum T_i)$	$\overline{\sum T_i}$	$\min(\sum T_i)$
la01	(5,10)	762 *	787.4	773	**783.8**	773
la02	(5,10)	266 *	283.4	266 *	**277.6**	266 *
la03	(5,10)	357 *	357.0 *	357 *	357.0 *	357 *
la04	(5,10)	1165 *	**1217.2**	1165 *	1284.2	1165 *
la05	(5,10)	557 *	557.0 *	557 *	557.0 *	557 *
la06	(5,15)	2516	**2790.0**	2616	2912.4	2847
la07	(5,15)	1677 *	1942.2	1869	**1904.2**	1677 *
la08	(5,15)	1829 *	2335.0	1905	**2129.6**	1829 *
la09	(5,15)	2851	3275.2	3161	**3226.6**	3131
la10	(5,15)	1841 *	2178.2	2069	**2119.4**	2046
la11	(5,20)	6534	6186.2	5704	**5846.4**	5253
la12	(5,20)	5286	5070.0	4859	**4997.8**	4809
la13	(5,20)	7737	7850.6	7614	**7611.8**	7342
la14	(5,20)	6038	**6616.8**	5714	6872.4	6459
la15	(5,20)	7082	**7088.6**	5626	7153.6	6330
la16	(10,10)	330 *	395.8	335	**360.8**	335
la17	(10,10)	118 *	144.2	120	**118.8**	118 *
la18	(10,10)	159 *	**229.4**	159 *	264.0	235
la19	(10,10)	243 *	306.6	243 *	**301.0**	243 *
la20	(10,10)	42 *	55.6	42 *	**42.0** *	42 *
la21	(10,15)	1956	**2847.2**	2101	2961.8	2680
la22	(10,15)	1455	**2052.8**	1773	2123.0	1988
la23	(10,15)	3436	**3692.6**	3506	3746.8	3424
la24	(10,15)	560 *	966.8	761	**724.0**	644
la25	(10,15)	1002	**1557.4**	1289	1583.0	1390
la26	(10,20)	7961	9275.8	8475	**8600.8**	7858
la27	(10,20)	8915	**7588.0**	6596	7641.8	6457
la28	(10,20)	2226	3430.8	2876	**3367.6**	2849
la29	(10,20)	2018	**2948.0**	2432	3099.0	2626
la30	(10,20)	6655	7621.6	6775	**7372.8**	6395
la31	(10,30)	20,957	18,921.8	17,984	**18,409.6**	17,751
la32	(10,30)	23150	21,991.4	20,401	**21,632.2**	20,546
la33	(10,30)	none	**22,494.2**	19,750	22,913.2	20,553
la34	(10,30)	none	**20,282.8**	18,633	21,911.8	19,577
la35	(10,30)	none	21,895.0	18,778	**21,384.4**	20,537
la36	(15,15)	675	1856.0	1711	**1839.0**	1599
la37	(15,15)	1070	**1774.2**	1621	1835.8	1594
la38	(15,15)	489 *	760.4	645	**745.4**	676
la39	(15,15)	754	**1573.0**	1391	1850.2	1551
la40	(15,15)	407 *	**1008.6**	613	1187.6	912

Moreover, it can be observed that the mean and the minimal total tardiness values differ significantly for most of the instances. Especially for problems of larger size, the mean objective function value often exceeds the minimal one by more than 10%. This aspect numerically emphasizes the ruggedness of the search space of the BJSPT, which leads to difficulties in the guidance of any heuristic search method. There seems to be a necessity of developing tailored neighborhood structures to more efficiently solve job shop problems with practically relevant constraints and objective functions. Based on these results, the involvement of random and diversifying components in a solution approach is recommendable together with the performance of several independent runs when using standard scheduling-tailored mechanisms.

7. Conclusions

In this paper, instances of a complex job shop scheduling problem are solved by a permutation-based heuristic search method. Two repairing schemes are proposed to facilitate the usage of well-known list encodings and generic operators for job shop problems with blocking constraints. In applying interchange- and shifts-based transition schemes, three neighborhoods are defined and analyzed with regard to structural issues and performance in an SA algorithm.

The computational experiments indicate that the proposed heuristic method using basic scheduling-tailored operators is capable of finding optimal and near-optimal schedules for small and medium size instances. Furthermore, it outperforms general-purpose techniques in generating feasible schedules for problems of large size. This gives evidence to its applicability in decision support systems for solving problems of practical relevance in production planning and logistics.

It turns out that the implementation of APIs by right shifts in the operation-based representation of a schedule is favorable compared to other mechanisms with respect to small search steps and solution quality. This narrows the required computational effort for heuristic search schemes using these types of operators. The complexity of the problem under study becomes clearly visible in the necessary enhancements of neighbor-defining moves and the resulting large distances of feasible schedules in the API-based neighborhoods. This work shows that existing generic scheduling-tailored operators have limits in their applicability to job shop problems with blocking constraints and tardiness-based objectives. The development of dedicated heuristic solution approaches, which allow more controllable search patterns, can be named as an important aspect of future research.

An advantage of guiding the choice of the executed interchanges by the objective function value is not substantiated by the numerical results. Furthermore, considering the total tardiness values obtained in several independent runs of the metaheuristic on the same instances, a high variance in quality of the best schedules found is observed. Thus, the ruggedness of the search space of the BJSPT and remarkable feasibility issues in the generation of neighboring schedules can be named as reasons for the ongoing difficulties in solving instances of practically relevant size. However, the computational results give evidence for hybrid solution approaches as a promising future research direction to overcome such issues. The combination of a heuristic technique to find feasible schedules for large instances and a general-purpose MIP method to quickly generate superior neighboring solutions is expected to be beneficial.

Overall, the proposed permutation-based heuristic can enhance solving capability of complex job shop scheduling problems. Important insights are gained into advantages and limits of applying generic operators to BJSPT instances, and future research directions are highlighted.

Author Contributions: Conceptualization, J.L. and F.W.; Software, J.L.; Investigation, J.L. and F.W.; Writing–original draft preparation, J.L. and F.W.

Funding: This research received no external funding.

Acknowledgments: The authors would like to thank Felix Müller for his remarkable commitment in the preparation of the statistical figures.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Garey, M.R.; Johnson, D.S.; Sethi, R. The Complexity of Flowshop and Jobshop Scheduling. *Math. Oper. Res.* **1976**, *1*, 117–129. [CrossRef]
2. Van Laarhoven, P.J.; Aarts, E.H.; Lenstra, J.K. Job shop scheduling by simulated annealing. *Oper. Res.* **1992**, *40*, 113–125. [CrossRef]
3. Adams, J.; Balas, E.; Zawack, D. The shifting bottleneck procedure for job shop scheduling. *Manag. Sci.* **1988**, *34*, 391–401. [CrossRef]
4. Nowicki, E.; Smutnicki, C. A fast taboo search algorithm for the job shop problem. *Manag. Sci.* **1996**, *42*, 797–813. [CrossRef]
5. Bürgy, R.; Gröflin, H. The blocking job shop with rail-bound transportation. *J. Comb. Optim.* **2016**, *31*, 152–181. [CrossRef]
6. D'Ariano, A.; Pacciarelli, D.; Pranzo, M. A branch and bound algorithm for scheduling trains in a railway network. *Eur. J. Oper. Res.* **2007**, *183*, 643–657. [CrossRef]
7. Liu, S.Q.; Kozan, E. Scheduling trains as a blocking parallel-machine job shop scheduling problem. *Comput. Oper. Res.* **2009**, *36*, 2840–2852. [CrossRef]
8. Mati, Y.; Rezg, N.; Xie, X. A taboo search approach for deadlock-free scheduling of automated manufacturing systems. *J. Intell. Manuf.* **2001**, *12*, 535–552. [CrossRef]
9. Bürgy, R. A neighborhood for complex job shop scheduling problems with regular objectives. *J. Sched.* **2017**, *20*, 391–422. [CrossRef]
10. Heger, J.; Voss, T. Optimal Scheduling of AGVs in a Reentrant Blocking Job-shop. *Procedia CIRP* **2018**, *67*, 41–45. [CrossRef]
11. Lange, J.; Werner, F. Approaches to modeling train scheduling problems as job-shop problems with blocking constraints. *J. Sched.* **2018**, *21*, 191–207. [CrossRef]
12. Brizuela, C.A.; Zhao, Y.; Sannomiya, N. No-wait and blocking job-shops: Challenging problems for GA's. *Int. Conf. Syst. Man Cybern.* **2001**, *4*, 2349–2354.
13. Groeflin, H.; Klinkert, A. A new neighborhood and tabu search for the blocking job shop. *Discret. Appl. Math.* **2009**, *157*, 3643–3655. [CrossRef]
14. Mattfeld, D.C.; Bierwirth, C. An efficient genetic algorithm for job shop scheduling with tardiness objectives. *Eur. J. Oper. Res.* **2004**, *155*, 616–630. [CrossRef]
15. Lange, J. Solution Techniques for the Blocking Job Shop Scheduling Problem with Total Tardiness Minimization. Ph.D. Thesis, Otto-von-Guericke-Universität Magdeburg, Magdeburg, Germany, 2019. [CrossRef]
16. Lange, J.; Werner, F. A Permutation-Based Neighborhood for the Blocking Job-Shop Problem with Total Tardiness Minimization. In *Operations Research Proceedings 2017*; Springer International Publishing: Cham, Switzerland, 2018; pp. 581–586.
17. Mascis, A.; Pacciarelli, D. Job-shop scheduling with blocking and no-wait constraints. *Eur. J. Oper. Res.* **2002**, *143*, 498–517. [CrossRef]
18. Lange, J.; Bürgy, R. Mixed-Integer Programming Heuristics for the Blocking Job Shop Scheduling Problem. In Proceedings of the 14th Workshop on Models and Algorithms for Planning and Scheduling Problems, MAPSP 2019, Renesse, The Netherlands, 3–7 June 2019; pp. 58–60.
19. Nowicki, E.; Smutnicki, C. An advanced tabu search algorithm for the job shop problem. *J. Sched.* **2005**, *8*, 145–159. [CrossRef]
20. Balas, E.; Simonetti, N.; Vazacopoulos, A. Job shop scheduling with setup times, deadlines and precedence constraints. *J. Sched.* **2008**, *11*, 253–262. [CrossRef]
21. Bierwirth, C.; Kuhpfahl, J. Extended GRASP for the job shop scheduling problem with total weighted tardiness objective. *Eur. J. Oper. Res.* **2017**, *261*, 835–848. [CrossRef]
22. Pinedo, M.; Singer, M. A shifting bottleneck heuristic for minimizing the total weighted tardiness in a job shop. *Nav. Res. Logist. (NRL)* **1999**, *46*, 1–17. [CrossRef]
23. Wang, T.Y.; Wu, K.B. A revised simulated annealing algorithm for obtaining the minimum total tardiness in job shop scheduling problems. *Int. J. Syst. Sci.* **2000**, *31*, 537–542. [CrossRef]
24. De Bontridder, K.M.J. Minimizing total teighted tardiness in a generalized job shop. *J. Sched.* **2005**, *8*, 479–496. [CrossRef]

25. Essafi, I.; Mati, Y.; Dauzère-Pérès, S. A genetic local search algorithm for minimizing total weighted tardiness in the job-shop scheduling problem. *Comput. Oper. Res.* **2008**, *35*, 2599–2616. [CrossRef]
26. Bülbül, K. A hybrid shifting bottleneck-tabu search heuristic for the job shop total weighted tardiness problem. *Comput. Oper. Res.* **2011**, *38*, 967–983. [CrossRef]
27. Mati, Y.; Dauzère-Pérès, S.; Lahlou, C. A general approach for optimizing regular criteria in the job-shop scheduling problem. *Eur. J. Oper. Res.* **2011**, *212*, 33–42. [CrossRef]
28. Zhang, R.; Wu, C. A simulated annealing algorithm based on block properties for the job shop scheduling problem with total weighted tardiness objective. *Comput. Oper. Res.* **2011**, *38*, 854–867. [CrossRef]
29. González, M.Á.; González-Rodríguez, I.; Vela, C.R.; Varela, R. An efficient hybrid evolutionary algorithm for scheduling with setup times and weighted tardiness minimization. *Soft Comput.* **2012**, *16*, 2097–2113. [CrossRef]
30. Kuhpfahl, J.; Bierwirth, C. A study on local search neighborhoods for the job shop scheduling problem with total weighted tardiness objective. *Comput. Oper. Res.* **2016**, *66*, 44–57. [CrossRef]
31. Meloni, C.; Pacciarelli, D.; Pranzo, M. A rollout metaheuristic for job shop scheduling problems. *Ann. Oper. Res.* **2004**, *131*, 215–235.[CrossRef]
32. Oddi, A.; Rasconi, R.; Cesta, A.; Smith, S.F. Iterative Improvement Algorithms for the Blocking Job Shop. In Proceedings of the ICAPS, Atibaia, Brazil, 25–29 June 2012.
33. AitZai, A.; Boudhar, M. Parallel branch-and-bound and parallel PSO algorithms for job shop scheduling problem with blocking. *Int. J. Oper. Res.* **2013**, *16*, 14–37. [CrossRef]
34. Pranzo, M.; Pacciarelli, D. An iterated greedy metaheuristic for the blocking job shop scheduling problem. *J. Heuristics* **2016**, *22*, 587–611. [CrossRef]
35. Dabah, A.; Bendjoudi, A.; AitZai, A.; Taboudjemat, N.N. Efficient parallel tabu search for the blocking job shop scheduling problem. *Soft Comput.* **2019**. [CrossRef]
36. Gröflin, H.; Klinkert, A. Feasible insertions in job shop scheduling, short cycles and stable sets. *Eur. J. Oper. Res.* **2007**, *177*, 763–785. [CrossRef]
37. Graham, R.L.; Lawler, E.L.; Lenstra, J.K.; Rinnooy Kan, A. Optimization and approximation in deterministic sequencing and scheduling: A survey. In *Annals of Discrete Mathematics*; Elsevier: Amsterdam, The Netherlands, 1979; Volume 5, pp. 287–326.
38. Pinedo, M. *Scheduling: Theory, Algorithms, and Systems*; Springer: Berlin/Heidelberg, Germany, 2016.
39. Brucker, P.; Knust, S. *Complex Scheduling*; Springer: Berlin/Heidelberg, Germany, 2011.
40. Błażewicz, J.; Ecker, K.H.; Pesch, E.; Schmidt, G.; Weglarz, J. *Handbook on Scheduling: From Theory to Applications*; International Handbook on Information Systems; Springer: Berlin/Heidelberg, Germany, 2007. [CrossRef]
41. Lawrence, S. *Supplement to Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques*; GSIA, Carnegie Mellon University: Pittsburgh, PA, USA, 1984.
42. Mascis, A.; Pacciarelli, D. *Machine Scheduling via Alternative Graphs*; Technical Report; Universita degli Studi Roma Tre, DIA: Rome, Italy, 2000.
43. Bierwirth, C.; Mattfeld, D.C.; Watson, J.P. Landscape regularity and random walks for the job-shop scheduling problem. In *European Conference on Evolutionary Computation in Combinatorial Optimization*; Springer: Berlin/Heidelberg, Germany, 2004, pp. 21–30.
44. Schiavinotto, T.; Stützle, T. A review of metrics on permutations for search landscape analysis. *Comput. Oper. Res.* **2007**, *34*, 3143–3153. [CrossRef]
45. Werner, F. Some relations between neighbourhood graphs for a permutation problem. *Optimization* **1991**, *22*, 297–306. [CrossRef]
46. Anderson, E.J.; Glass, C.A.; Potts, C.N., Machine Scheduling. In *Local Search in Combinatorial Optimization*; Aarts, E.H.L.; Lenstra, J.K., Eds.; Wiley: Chichester, UK, 1997; Chapter 11, pp. 361–414.
47. Lange, J. A comparison of neighborhoods for the blocking job-shop problem with total tardiness minimization. In Proceedings of the 16th International Conference of Project Management and Scheduling 208, Rome, Italy, 17–20 April 2018; pp. 132–135.