European Journal of Operational Research 311 (2023) 455–471

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Discrete Optimization

A general efficient neighborhood structure framework for the job-shop and flexible job-shop scheduling problems

Karim Tamssaouet*, Stéphane Dauzère-Pérès

Department of Accounting and Operations Management, BI Norwegian Business School, Nydalsveien 37, Oslo N-0484, Norway

A R T I C L E   I N F O

Article history:
Received 30 September 2021
Accepted 10 May 2023
Available online 13 May 2023

Keywords:
Scheduling
Job-shop scheduling
Flexible job-shop scheduling
Heuristics

A B S T R A C T

This article introduces a framework that unifies and generalizes well-known literature results related to local search for the job-shop and flexible job-shop scheduling problems. In addition to the choice of the metaheuristic and the neighborhood structure, the success of most of the influential local search approaches relies on the ability to quickly and efficiently rule out infeasible moves and evaluate the quality of the feasible neighbors. Hence, the proposed framework focuses on the feasibility and quality evaluation of a general move when solving the job-shop and flexible job-shop scheduling problems for any regular objective function. The proposed framework is valid for any scheduling problem where the defined neighborhood structure is appropriate, and each solution to the problem can be modeled with a directed acyclic graph with non-negative weights on nodes and arcs. The feasibility conditions and quality estimation procedures proposed in the literature rely heavily on information on the existence of a path between two nodes. Thus, based on an original parameterized algorithm that asserts the existence of a path between two nodes, novel generic procedures to evaluate the feasibility of a move and estimate the value of any regular objective function of a neighbor solution are proposed. We show that many well-known literature results are special cases of our results, which can be applied to a wide range of shop scheduling problems.

1. Introduction

Scheduling deals with the allocation of resources to tasks over time to optimize one or more objectives (Pinedo, 2016). The job-shop scheduling problem is one of the most well-known (Bowman, 1959) and difficult (Gonzalez & Sahni, 1978) scheduling problems in the literature. In a job-shop scheduling problem, a set of jobs must be processed on a set of machines, and each job requires a sequence of operations (a route) before being completed. A machine can only perform one operation at a time, and preemption is not allowed. Each operation can only be performed on a specified machine while, in the flexible job-shop scheduling problem, operations can be assigned to multiple machines. Such problems and their extensions can model a wide range of real-world applications, for example in the contexts of healthcare (Pham & Klinkert, 2008), manufacturing (Tamssaouet et al., 2022) and transport (Lamorgese & Mannino, 2019). This explains why so much research has been conducted on designing exact and heuristic algorithms to solve these problems.

Job-shop scheduling problems are computationally intractable when dealing with large instances. Therefore, heuristic algorithms are considered in practice to find good quality solutions in reasonable computational times. It is common to differentiate between constructive and improvement heuristics. A constructive heuristic builds a solution from scratch, often by using some greedy criteria. An improvement heuristic, starting from one or a set of initial solutions generated randomly or by some constructive heuristic, iteratively explores the search space to obtain better solutions. Local search algorithms are a broad class of improvement algorithms where, at each iteration, an improving solution is found by searching the *neighborhood* of the current solution, that is, a set of solutions that are, in some sense, “close” to that solution. Therefore, to derive a local search algorithm for an optimization problem, one must carefully define the *neighborhood structure* specifying the neighborhood to explore for each solution. Typically, a neighborhood structure is not defined by explicitly enumerating the set of possible neighbors, but rather implicitly by defining the possible local changes, called *moves*, to apply to the current solution. For example, a well-known move for the job-shop scheduling problem consists in swapping two consecutive critical operations on a machine (Van Laarhoven et al., 1992).

* Corresponding author.
E-mail addresses: karim.tamssaouet@bi.no (K. Tamssaouet), stephane.dauzere-peres@bi.no (S. Dauzère-Pérès).

https://doi.org/10.1016/j.ejor.2023.05.018
0377-2217/© 2023 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/)

Local search approaches, such as Tabu Search or Simulated Annealing, have been developed since the early 1990s to solve the job-shop and flexible job-shop scheduling problems. In addition to local search algorithms, another broad class of improvement heuristics can be viewed as an iterative improvement of a population of solutions. Algorithms such as Genetic Algorithms, Particle Swarm Optimization, and Scatter Search belong to this class of improvement heuristics. Over the last years, a new class of hybrid improvement heuristics has been getting increasing interest (Chaudhry & Khan, 2016). These heuristics may combine a population-based heuristic with a local search heuristic to exploit their strengths. This paper describes a general framework, which combines two parameterized procedures, that provides an efficient neighborhood structure that can be used within a pure local search heuristic or a hybrid heuristic, combining population-based and local search heuristics. We show that the proposed framework encompasses well-known and different results from the literature, including Dauzère-Pérès & Paulli (1997); Mastrolilli & Gambardella (2000); Shen et al. (2018) and Kasapidis et al. (2021) (see Tables 2 and 3), thus validating the computational effectiveness of our procedures.

The paper is organized as follows. Section 2, by highlighting the weaknesses of the available local search heuristics to solve job-shop scheduling problems, motivates this work and summarizes our contributions. Section 3 states the notation and concepts used throughout the paper and defines the validity scope of the proposed framework. Section 4 presents a parameterized algorithm making it possible to assert the existence of a path between two nodes. Section 5 introduces a parameterized procedure based on the algorithm of Section 4 to evaluate the feasibility of a move. This section also demonstrates the generality of the proposed novel procedure by considering several results in the literature and showing that they can be obtained using specific values for the procedure parameters. Section 6 is devoted to the development of a generic procedure to be used in the computation of valid lower bounds on any regular objective function of the neighbors of a solution. As for the feasibility evaluation procedure, several results from the literature are used to show the generality of the proposed procedure. To avoid overloading the paper, the Supplementary Material accompanying this article provides all the propositions showing how a selected list of known results can be reproduced by choosing specific values for the parameters of the proposed procedures. Finally, this work is concluded in Section 7 with some perspectives on the possible extension of the scope of the proposed framework.

2. Motivations and contributions

At each iteration of a local search heuristic and using some quality measures and a selection strategy, a solution from the neighborhood of the current solution is selected to become the new current solution.

To structure the discussion and position our contributions, Algorithm 1 provides a high-level template of a local search heuristic. The representation of the initial solution, the incumbent solution, and the best solution are respectively denoted G_0, G and G^*. The neighborhood structure defining the neighborhood of the solution represented by G is denoted $\mathcal{N}(G)$, and the neighbor of G is denoted $\tilde{G}$. Below, let us discuss some of the critical and inter-related questions that need to be answered when implementing a local search heuristic:

- *Solution representation:* Choosing the appropriate representation or the encoding of a solution is a fundamental design question in developing a solution approach. When defining a representation, one has to bear in mind how the solution is evaluated and how the search operators work (Talbi, 2009). One common representation for job-shop scheduling problems is a list of strings, each representing a permutation of operations on a specific machine. Another popular representation relies on a graph, and provides a convenient data structure that can benefit from the diversity of algorithms available in graph theory. Graph modeling is adopted in this work, and a detailed discussion is provided in Section 3.1.

Algorithm 1 High-level template of a local search heuristic.

```
1:  procedure LocalSearch($G_0$)
2:      $G \leftarrow G^* \leftarrow G_0$
3:      while Stopping criteria not satisfied do
4:          ComputeSchedule($G$)
5:          for $\tilde{G} \in \mathcal{N}(G)$ do
6:              if EvaluateFeasibility($\tilde{G}$) = true then
7:                  EvaluateQuality($\tilde{G}$)
8:          $G \leftarrow$ Select($G, \mathcal{N}(G)$)
9:          if $G$ is better than $G^*$ then
10:             $G^* \leftarrow G$
11:     return $G^*$
```

- *Solution evaluation (Algorithm 1 - Line 4):* As the most useful representations do not fully characterize a solution, the solution must be decoded. For example, it is necessary to compute the start times of operations given that the two representations discussed above “only” provide the assignment and sequencing decisions. The quality evaluation of a solution is usually straightforward after computing the schedule, given that most objective functions rely on the timing decisions of operations. It is often desirable to deduce additional information that can be useful in practice or to make the search more effective and efficient. For example, identifying the critical operations is a pre-requisite for some of the successful neighborhood structures (e.g., Dauzère-Pérès & Paulli, 1997; Van Laarhoven et al., 1992). The results and procedures proposed in this work depend on the information calculated during the evaluation of the incumbent solution. Therefore, Section 3.1 also elaborates on this question.
- *Neighborhood structure (Algorithm 1 - Line 5):* By defining the neighborhood to explore, and therefore its size, choosing an appropriate neighborhood structure is a crucial design decision for the performance of the local search heuristic. One must consider the trade-off between the solution quality and the search efficiency. The neighborhood structure adopted in this work is described in Section 3.2.
- *Neighborhood evaluation (Algorithm 1 - Lines 6 and 7):* In general, it is crucial for the success of a local search heuristic to be able to *quickly evaluate the quality* of the neighbors of a solution. Moreover, as some neighbors might not be feasible, it is also critical to *quickly evaluate the feasibility* of the neighbors of a solution. The primary objective of this work is to propose procedures that can efficiently evaluate the feasibility (see Section 5) and quality (see Section 6) of the neighbors of a solution.
- *Neighbor Selection (Algorithm 1 - Line 8):* Usually, the *quality evaluation* of a neighbor is performed with the optimized objective function as a quality measure. The most common selection strategies are “best improvement”, which selects the best move in the neighborhood, and “first improvement”, which selects the first strictly improving move encountered when exploring the neighborhood.

This work aims to contribute to the design of efficient local search heuristics. More specifically, we propose a general framework that encompasses efficient procedures for the feasibility and quality evaluation of the neighbors of a solution. Indeed, when the number of neighbors is large, the neighborhood evaluation is computationally the most expensive operation in each local search iteration. This explains why the most well-known approaches innovate by reducing the size of the neighborhood (e.g., Nowicki & Smutnicki, 1996) or by designing efficient procedures that quickly evaluate the neighbors of a solution (e.g., Dauzère-Pérès & Paulli, 1997; Dell'Amico & Trubian, 1993). Despite the originality of most of these approaches, the properties on which the neighborhood evaluation procedures are based strongly depend on scheduling problem to solve.

Consequently, these properties must be adapted or generalized each time the problem being solved is modified to include additional constraints or optimize different objective functions. For example, the work of Balas & Vazacopoulos (1998) is adapted in Braune et al. (2013) to consider the optimization of min-sum objectives instead of the classical makespan criterion, which corresponds to minimizing the completion time of all jobs. Beyond the research effort spent to make these non-trivial adaptations, this strong dependency on the solved problems of the advanced approaches may explain their scarce implementation in practice as real problems include features not considered in the pure research problems.

Our objective is to contribute to the design of a general neighborhood structure framework that can support the resolution of a large variety of scheduling problems. The proposed framework encompasses results from some of the most influential works that solve job-shop and flexible job-shop scheduling problems with local search heuristics. Among those works, we can cite Balas & Vazacopoulos (1998); Dauzère-Pérès & Paulli (1997); Dauzère-Pérès et al. (1998); Dell'Amico & Trubian (1993); Mastrolilli & Gambardella (2000); Mati et al. (2011); Taillard (1994); Zhang et al. (2007) and Shen et al. (2018). Establishing such a framework should allow researchers to avoid spending time on the difficult and time-consuming tasks of adapting existing results and focus on questions that generate new knowledge. The applicability scope of the framework is first defined by the relevance of the proposed neighborhood structure to the scheduling problem to be solved. The neighborhood structure specifies the moves that consist of deleting an operation from a sequence and inserting it in a new sequence. As shown in Section 3.2, the set of moves specified by the most used neighborhood structures in the literature for the job-shop scheduling problems is a subset of the moves specified by the neighborhood structure defined in this work.

Despite the widespread use of graphs to model scheduling problem solutions, the problem-dependent components of local search heuristics are strongly related to the specific constraints of the problem to be solved, and are only partly based on the knowledge that can be extracted from a graph. For example, it is sufficient to modify the arc weights of a graph representing a solution for the flexible job-shop scheduling problem to account for sequence-dependent setup times. However, the feasibility and quality evaluation procedures proposed in Dauzère-Pérès & Paulli (1997) are adapted in Shen et al. (2018) to take sequence-dependent setup times into account in the flexible job-shop scheduling problem. The insight that allows the proposed framework to be less dependent on the problem constraints is to define its applicability scope through general specifications on the *solution graph*. In addition to the considered neighborhood structure, the applicability of the proposed framework is possible whenever a problem induces a solution graph that is a directed acyclic graph (*DAG*) with non-negative weights on nodes and arcs. Therefore, our framework is applicable as long as the considered constraint does not induce a graph not complying with the specifications. Such "valid" constraints include release dates, minimum time lags, sequence-dependent setup times, multiple resources, and nonlinear routing. Some constraints our proposed framework cannot handle are maximum time lags and blocking constraints. When the defined neighborhood structure is appropriate for a scheduling problem and its solution graphs comply with the specifications described above, the framework provides efficient procedures for feasibility evaluation and quality evaluation.

Like in the existing successful local search approaches, the efficiency of the proposed procedures is achieved by evaluating a neighbor's feasibility and quality without actually performing its related move. Instead, a neighbor is evaluated by only relying on the already available information provided by the current solution. For example, the feasibility of a neighbor is not checked after performing its move, but by checking some sufficient conditions exploiting the start times in the current solution of the involved operations. However, the efficiency is obtained at the expense of the *accuracy* of the evaluation. The feasibility evaluation procedure can be viewed as a classifier separating feasible and infeasible neighbors. The feasibility evaluation accuracy can be measured, for instance, as a ratio of the total number of true positives and the number of true negatives divided by the total number of the neighbors. For example, Kasapidis et al. (2021) report that the feasibility conditions adapted from Dauzère-Pérès & Paulli (1997) to solve a flexible job-shop scheduling problem with an arbitrary precedence graph, similarly to Dauzère-Pérès et al. (1998), have an accuracy of 91%. By increasing the accuracy, the risk of missing improving moves is decreased. Regarding the quality evaluation, the objective function of a neighbor is estimated rather than computed. Therefore, the quality evaluation accuracy can for instance be measured as the mean absolute relative deviation of the estimated objective functions from the actual objective functions of the neighbors. For instance, Mati et al. (2011) report an accuracy ranging from 0.1% to 13% depending on the problem instance and the optimized objective function. Increasing this accuracy decreases the risk of selecting a non-improving move while the neighborhood contains at least an improving move.

Contrary to the existing approaches in the literature, the proposed framework, beyond its generality, allows the trade-off between evaluation accuracy and computational cost to be controlled by setting some key parameters. The proposed *parameterized* procedure for feasibility evaluation can reach 100% accuracy if the resulting computational cost is considered less important than the risk of discarding feasible and promising moves. The proposed *parameterized* procedure for quality evaluation does not guarantee 100% accuracy, even if it offers some control over the trade-off between evaluation accuracy and computational cost. However, the procedure can be used for any regular objective function, while most existing approaches are specifically designed for the case of makespan minimization. The two procedures evaluating the neighbors' feasibility and quality rely on a novel parameterized procedure that asserts the existence of a path between two nodes in a DAG with non-negative weights on nodes and arcs. While the existence of a path between two nodes is an explicit argument used in the proofs of the different feasibility conditions of moves proposed in the literature, this knowledge is more implicit regarding the available quality evaluation procedures. For example, as discussed in Section 6.1, the distinction between *forward* and *backward* moves makes it possible to assert the absence of paths between some nodes implicitly.

In summary, the proposed framework encompasses and generalizes results already published in the literature. First, contrary to what can be found in the literature and as discussed above, the proposed feasibility and quality evaluation procedures allow the trade-off between evaluation accuracy and computational cost to be controlled. Second, the generality of the framework is possible by basing its validity on the specifications of the solution representation instead of the problem constraints. As a consequence, more general problems can be handled. For example, the proposed procedures are valid when optimizing any regular objective function (Mati et al., 2011) when solving a shop problem where an operation may require several resources, have several route predecessors and successors (Dauzère-Pérès et al., 1998) while considering sequence-dependent setup times (Shen et al., 2018). This is possible as the solution for such a problem can be represented by a DAG with non-negative weights on nodes and arcs.

3. Modeling and notations

This section formally introduces the framework within which the propositions in this article are valid. The notations are also defined and summarized in Table 1. They are chosen to hopefully clearly illustrate the generality of the proposed results by showing that results well known in the literature are special cases.

3.1. Solution representation and evaluation

The propositions of this work are valid for any scheduling problem for which schedules can be represented by a DAG with non-negative weights on nodes and arcs. As different graph models can be found in the literature, we use a general graph modeling to encompass the different cases. For example, in Balas & Vazacopoulos (1998), the duration of an operation is associated to an arc weight. On the contrary, Mastrolilli & Gambardella (2000) and Shen et al. (2018) associate the processing times of operations with the node weights. Hereafter, weights are assigned to both nodes and arcs. Therefore, let $G = (V, A)$ be a DAG with V the set of nodes and A the set of arcs. The set of nodes represents the job operations to schedule, while the arcs model precedence constraints to be satisfied by the schedule. For a node $v \in V$, let us denote the set of incoming arcs by $in(v) \subset A$ and the set of predecessor nodes by $\mathcal{B}(v) \subset V$. Also, the set of outgoing arcs of node v is denoted by $out(v) \subset A$ and the set of successor nodes by $\mathcal{F}(v) \subset V$. A path P_{v_1,v_k} from $v_1 \in V$ to $v_k \in V$ is defined as a sequence of nodes and arcs $P_{v_1,v_k} = (v_1, v_2, \dots, v_k)$ with $(v_i, v_{i+1}) \in A$ for all $1 \le i < k$. Let the set of all paths from $v_1 \in V$ to $v_k \in V$ be denoted by $\mathcal{P}_{v_1,v_k}$.

Let l_v and $l_{u,v}$ denote the weights associated to node $v \in V$ and arc $(u, v) \in A$, respectively. As already stressed, the properties and procedures developed in this work assume that all weights are non-negative, i.e., $l_v \ge 0$ for each $v \in V$ and $l_{u,v} \ge 0$ for each $(u, v) \in A$. Also, let $l_{out(v)} = \min\{l_{v,w} | (v, w) \in out(v)\}$ and $l_{in(v)} = \min\{l_{u,v} | (u, v) \in in(v)\}$. The length of a path P_{v_1,v_k} is defined as the sum of all nodes weights belonging to the path with the exception of the two extreme nodes plus the weights of all path arcs, i.e., $l(P_{v_1,v_k}) = \sum_{i=2}^{i=k-1} l_{v_i} + \sum_{i=1}^{i=k-1} l_{v_i,v_{i+1}}$. Let us denote the length of a longest path from a node $u \in V$ to a node $v \in V$ by $L(u, v) = \max\{l(P_{u,v}) | P_{u,v} \in \mathcal{P}_{u,v}\}$. In this work, we assume that the realistic triangle inequality is satisfied, i.e., given three nodes u, v and w, $l_{u,w} \le l_{u,v} + l_v + l_{v,w}$.

It has already been mentioned in Section 2 that the graph representation does not fully characterize a schedule. Given a solution graph, defined as a DAG whose node and arc weights are initialized with the relevant problem data, it is possible to compute the corresponding semi-active schedule by assigning to each operation start time the length of the longest path from the graph source node to the node associated to the operation. The term *reference* node is used to qualify the node α from which the length of the longest path is computed. As discussed later in the paper, it is also useful to compute the longest path length to a node from a reference node that is not the source node. Given a reference node α, the vector of the lengths of the longest path from α to all nodes is denoted $\mathcal{L}_\alpha$, i.e., $\mathcal{L}_\alpha = (L(\alpha, v) | v \in V)$.

Algorithm 2 presents a well-known procedure (Katriel et al., 2005) that computes, for each node $v \in V$, the length $L(\alpha, v)$ of

Table 1
List of notations.

Notations	Descriptions
G	DAG with non-negative weights on the nodes N and arcs A
l_v	Weight associated to node $v \in V$
$l_{u,v}$	Weight associated to arc $(u, v) \in A$
$out(v)$	Set of outgoing arcs of node v
$in(v)$	Set of incoming arcs of node v
$l_{out(v)}$	Minimum weight among those of the outgoing arcs, i.e., $l_{out(v)} = \min\{l_{v,w} \| (v, w) \in out(v)\}$
$l_{in(v)}$	Minimum weight among those of the incoming arcs, i.e., $l_{in(v)} = \min\{l_{u,v} \| (u, v) \in in(v)\}$
$\mathcal{B}(v)$	Set of predecessors of node v
$\mathcal{F}(v)$	Set of successors of node v
$P_{u,v}$	Path from u to v
$l(P_{u,v})$	Length of path $P_{u,v}$
$\mathcal{P}_{u,v}$	Set of all paths from u to v
$L(u, v)$	Length of the longest path from u to v, i.e., $L(u, v) = \max\{l(P_{u,v}) \| P_{u,v} \in \mathcal{P}_{u,v}\}$
α	Reference node in G, e.g., the dummy start node
$\mathcal{L}_\alpha$	Vector of the length of the longest path from α to all nodes of G, i.e., $\mathcal{L}_\alpha = (L(\alpha, v) \| v \in V)$
G'	Reverse graph of G with nodes N and arcs A'
$\square'$	All the notations with a prime are related to G'
ω	Reference node in G', e.g., the dummy end node in G
$\mathcal{L}'_\omega$	Vector of the length of the longest path from ω to all nodes of G', i.e., $\mathcal{L}'_\omega = (L'(\omega, v) \| v \in V)$
$\mathcal{M}$	A move specified by the neighborhood structure defined as $\mathcal{M} = \{u, O, D\}$ Initially sequenced between and ordered pair O (e.g., $O = (s, t)$) in G, a node u is inserted between a new ordered pair D (e.g., $D = (v, w)$)
$\mathcal{D}$	Set of arcs to be deleted when applying a move $\mathcal{M} = \{u, O, D\}$, i.e., $\mathcal{D} = \{(O_1, u), (u, O_2), (D_1, D_2)\}$
$\mathcal{I}$	Set of arcs to be inserted when applying a move $\mathcal{M} = \{u, O, D\}$, i.e., $\mathcal{I} = \{(D_1, u), (u, D_2), (O_1, O_2)\}$
$\tilde{G}$	Directed graph representing a neighbor obtained by applying move $\mathcal{M}$ on G
$\tilde{\square}$	All the notations with a tilde are related to the neighbor graph $\tilde{G}$
$\tilde{L}(u, v)$	Length of the longest path from u to v in $\tilde{G}$
$\hat{L}(u, v)$	Estimated length of the longest path from u to v in $\tilde{G}$
δ_v	Decrease in the longest path from a reference node α due to move $\mathcal{M}$, i.e., $\delta_v = L(\alpha, v) - \tilde{L}(\alpha, v)$
$\hat{\delta}_v$	Upper bound on δ_v, i.e., $\hat{\delta}_v = L(\alpha, v) - \hat{L}(\alpha, v)$
$\mathcal{C}_v$	Layer associated to node v, i.e., set of nodes ensuring that, if a path traverses v in G, it traverses at least one of the layer nodes
k	Forward search cutoff from a node u in G
k'	Forward search cutoff from a node v in G', i.e., backward search from v in G

Algorithm 2 Evaluation of length of longest path from reference node α to any node of G.

```
1:  procedure COMPUTELONGESTPATHLENGTH($G$, $\alpha$)
2:      for $v \in V$ do
3:          $indegree(v) \leftarrow |in(v)|$
4:          $L(\alpha, v) \leftarrow -\infty$
5:      $Q \leftarrow \{v | indegree(v) = 0\}$
6:      $L(\alpha, \alpha) \leftarrow 0$
7:      while $Q \neq \emptyset$ do
8:          $v \leftarrow dequeue(Q)$
9:          for $(u, v) \in (v)$ do
10:             $L(\alpha, v) \leftarrow \max\{L(\alpha, v), L(\alpha, u) + l_u + l_{u,v}\}$
11:         for $(v, w) \in out(v)$ do
12:             $indegree(w) \leftarrow indegree(w) - 1$
13:             if $indegree(w) = 0$ then
14:                 $enqueue(Q, w)$
15:     $\mathcal{L}_\alpha \leftarrow (L(\alpha, v) | v \in V)$
16:     return $\mathcal{L}_\alpha$
```

the longest path from reference node α to v. The running time of the algorithm is $O(|V|+|A|)$, and its key idea is to consider the nodes in topological order, which guarantees that, when a node is considered, its predecessors have the correct longest paths. Lines 2–4 compute the initial indegree of the nodes and initialize the length of their longest path from reference node α. Line 5 inserts all the source nodes into the queue Q, and Line 6 initializes the distance from α to itself. The queue Q is a basic data structure that is maintained in a sequence, and which can be updated by adding elements at one end of the sequence, called the rear of the queue, and by removing elements from the other end of the sequence, called the front of the queue. Adding an element to the rear of the queue is known as *enqueue*, and removing an element from the front is known as *dequeue*. Line 8 dequeues a node v for which the length of its longest path from α is computed in Lines 9 and 10. Lines 11–14 decrement the indegrees of the successors of v and enqueue those for which all the predecessors have been visited. At the end of the procedure, the length of the longest path from α to $v \in V$ can be either $L(\alpha, v) = -\infty$ or $L(\alpha, v) \geq 0$. Having $L(\alpha, v) = -\infty$ for node $v \in V$ means that there is no path from α to v in G. If there is a path from α to v in G, then $L(\alpha, v) \geq 0$ due to the non-negative weights on nodes and arcs. Another trivial property that supports our findings, formally stated in Property 1, is the non-decreasing lengths along any path in the graph when the node and arc weights are non-negative.

Property 1. *Let $G = (V, A)$ be a DAG with non-negative weights on nodes and arcs and $\alpha \in V$ a reference node. If there is path from u to v, then $L(\alpha, u) \leq L(\alpha, v)$.*

As mentioned above, when the source node of the graph, that models the start of the schedule, is used as a reference node in Algorithm 2, the procedure assigns the earliest start times (also called operation heads) to the graph nodes. In addition to the earliest start times, it is often necessary to derive other information, such as the tail, the slack, or the criticality of each operation. Again, Algorithm 2 can provide such information when computing the longest path length from some sink nodes. When minimizing the makespan, the DAG has one sink node that models the end of the schedule. When other min-sum objective functions (e.g., the total weighted completion time or the number of tardy jobs) are optimized, the DAG has as many sink nodes as the number of jobs, each sink modeling the completion time of a job. As is most often in the literature, we assume in this work that Line 4 of Algorithm 1 consists of running Algorithm 2 from the source node and the different sink nodes. The tails, slacks, and criticality of operations that can be computed by running Algorithm 1 from the sink nodes are rarely useful in practice. The benefit this information can bring during the search when exploited is, however, significant and proven, making it worth paying the computational cost of using Algorithm 2 many times. For example, several well-known neighborhood structures restrict the moves to the set of operations that are critical (Nowicki & Smutnicki, 1996; Van Laarhoven et al., 1992) and (Dauzère-Pérès & Paulli, 1997). Instead of the criticality of operations, Mastrolilli & Gambardella (2000) rely on the heads and tails of operations to design an effective tabu search heuristic.

Although having the same objective, the properties proposed in the literature to design efficient procedures for feasibility and quality evaluation are heterogeneous. One of the differences is whether only the heads or the tails of operations are used, or both. For example, the sufficient conditions on the feasibility of a move in Dauzère-Pérès & Paulli (1997) only rely on the heads, while Mastrolilli & Gambardella (2000) use both the heads and the tails. By recognizing the symmetry between the heads and the tails of operations, it is possible to state a set of properties that are valid for both the heads and the tails. To achieve this, we need to associate to each DAG $G = (V, A)$ a reverse graph $G' = (V, A')$ obtained by reversing all the arcs A of G. Note that no change is made to the node and arc weights. All the notations presented above for the initial graph are modified by adding a prime $'$ when applied to the reverse graph G'. Due to the transformation, we have, for example, $\mathcal{B}(v) = \mathcal{F}'(v)$ and $\mathcal{F}(v) = \mathcal{B}'(v)$. For clarity, an exception for the use of prime $'$ is made for the reference node that is denoted in the reverse graph G' by ω. Thanks to the definition of a path length, $L(v, \omega) = L'(\omega, v)$. Using this transformation, answering the question of the existence of a path from u to v in G is equivalent to answering the question of the existence of a path from v to u in G'. It becomes then possible to check whether a path exists between u to v in G by using the lengths of the longest paths in G'. Note that there is no need to construct the reverse graph G', as it is sufficient to use the reverse topological ordering in Algorithm 2 to compute the longest path from a reference node ω to any node of the reverse graph G'.

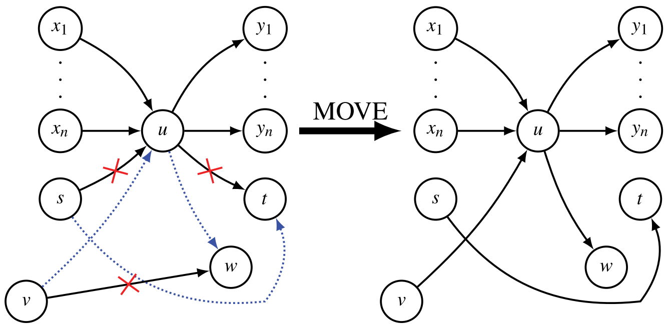

Fig. 1. Moving node u from between nodes s and t to between nodes v and w.

In addition to the heads and tails, some of the feasibility and quality evaluation procedures available in the literature are based on the notion of “node level”, introduced in Mati et al. (2011) and defined as the maximum number of arcs from the dummy start node to the given node. Following the comment on tails, the same observation supports the conditions proposed in the literature that use node levels. Therefore, it is sufficient to apply the same conditions on an unweighted graph, i.e., $l_v = 0 \ \forall v \in V$ and $l_{u,v} = 1 \ \forall (u, v) \in A$.

3.2. Neighborhood structure definition

In addition to the specifications of the solution graph, the applicability scope of the framework is defined by the relevance of the proposed neighborhood structure for the scheduling problem. The neighborhood structure used in this work specifies the moves that consist of deleting an operation from a sequence and inserting it in a new sequence. Suppose that a feasible solution is given with operation u sequenced between operations s and t on resource l. Let $O = (s, t)$ denote the original ordered pair of nodes s and t between which u is sequenced in G and let $O_1 = s$ and $O_2 = t$. The neighborhood structure specifies the moves, each consisting of moving operation u between operations v and w on resource m, where m can be different from l. Similarly, let $D = (v, w)$ denote the destination ordered pair of nodes v and w between which u will be sequenced in $\tilde{G}$ and let $D_1 = v$ and $D_2 = w$. Let $\mathcal{M} = \{u, O = (s, t), D = (v, w)\}$ denote a move specified by the neighborhood structure. As shown in Fig. 1, a move $\mathcal{M}$ induces the deletion of a set of arcs, denoted $\mathcal{D} = \{(s, u), (u, t), (v, w)\}$, and the insertion of a set of new arcs, denoted $\mathcal{I} = \{(s, t), (v, u), (u, w)\}$.

The directed graph of the neighbor obtained by applying move $\mathcal{M}$ to the solution associated to G is denoted $\tilde{G}$. The notations using $\tilde{}$ refer to neighbor graph $\tilde{G}$. For example, the length of the longest path from u to v in neighbor graph $\tilde{G}$ is denoted $\tilde{L}(u, v)$. As

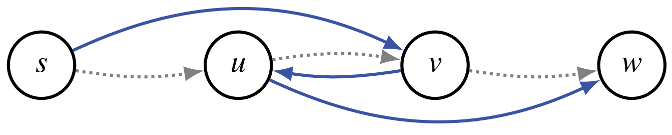

Fig. 2. Swap move on (u, v).

the quality of the neighbors is estimated using the information of the current solution, the estimated length of the longest path from u to v in $\tilde{G}$ is denoted $\hat{L}(u, v)$. Let $\delta_v = L(\alpha, v) - \tilde{L}(\alpha, v)$ denote the decrease in the longest path from a reference node to node v. As $\tilde{L}(\alpha, v)$ is the value we aim to estimate, let $\hat{\delta}_v = L(\alpha, v) - \hat{L}(\alpha, v)$ denote an upper bound on δ_v. The equivalent notations in the reverse graph G', where ω is the reference node, are δ'_v and $\hat{\delta}'_v$.

If move $\mathcal{M} = \{u, O, D\}$ is applied on graph G, the move to be applied on the reverse graph G' is $\mathcal{M}' = \{u, O', D'\}$ where $O' = (t, s)$ if $O = (s, t)$ and $D' = (w, v)$ if $D = (v, w)$. In other words, we have the following identities $O_1 = O'_2$, $O_2 = O'_1$, $D_1 = D'_2$ and $D_2 = D'_1$. A move $\mathcal{M}'$ induces the deletion of a set of arcs, denoted $\mathcal{D}' = \{(t, u), (u, s), (w, v)\}$, and the insertion of a set of new arcs, denoted $\mathcal{I}' = \{(t, s), (w, u), (v, w)\}$. These notations are helpful to avoid establishing separate results for G and G'. For example, instead of defining a lower bound on $\tilde{L}(\alpha, s)$ and $\tilde{L}'(\omega, t)$, one general result regarding a lower bound on $\alpha, \tilde{O}_1$ can be formulated because of the symmetry between $\tilde{L}(\alpha, O_1 = s)$ in G and $\tilde{L}'(\omega, O'_1 = t)$ in G'.

The neighborhood structure defined above is large enough that its associated move set includes the move set specified by the most commonly used neighborhood structures for solving the job-shop scheduling problem and its extensions. Below is a shortlist of references using a neighborhood structure specifying a set of moves that is included in the neighborhood structure defined in this work:

- Job-shop scheduling problem: Balas & Vazacopoulos (1998); Dell'Amico & Trubian (1993); Nowicki & Smutnicki (1996); Van Laarhoven et al. (1992); Zhang et al. (2007) and Mati et al. (2011).
- Flexible job-shop scheduling problems: Dauzère-Pérès & Paulli (1997) and Mastrolilli & Gambardella (2000).
- Flexible job-shop scheduling problem with additional constraints: Dauzère-Pérès et al. (1998); Kasapidis et al. (2021); Knopp et al. (2017); Shen et al. (2018) and Tamssaouet et al. (2022).

Note that, as a DAG can represent the schedules of problems tackled in the works listed above with non-negative weights on nodes and arcs, the proposed framework can be used to solve different problems. However, nothing prevents making the framework tailored to the specific problem to be solved. For example, Van Laarhoven et al. (1992) introduce the first successful neighborhood structure for the job-shop scheduling problem, often denoted N1 (Błażewicz et al., 1996). The N1 neighborhood is generated by swapping any adjacent pair of critical operations on the same machine. The swap of an arc (u, v) is illustrated in Fig. 2, and it is clear that it is part of the moves specified by the neighborhood structure defined in this work. By considering only the swapping of any adjacent pair of critical operations, it is shown that all N1 moves lead to feasible solutions when solving the classical job-shop scheduling problem. Therefore, using the N1 neighborhood structure does not require the use of the proposed feasibility evaluation procedure. However, this property is no longer valid when, for example, including the sequence-dependent setup times (Zoghby et al., 2005). Also, there is a need for an efficient quality evaluation even when solving the pure job-shop scheduling problem. The proposed procedure in this work can perform such evaluation on only the moves defined by the N1 structure. For example, it is shown in Section 6.3 that the proposed quality evaluation procedure reproduces the makespan estimation proposed by Taillard (1994).

The remainder of this article is devoted to developing parameterized procedures for feasibility and quality evaluation of a move without actually making the move. The proposed procedures can be used as long as the solutions to a problem can be modeled through a DAG with non-negative weights on nodes and arcs, and the relevant neighborhood structure specifies a set of moves that is equal to or a subset of the move set determined by the neighborhood structure defined above. To avoid redundancy when presenting the results, we assume the non-negativity of the weights in the DAG in the remainder of the paper.

For convenience, the main contributions and structure of the paper are highlighted in Fig. 3. Based on a BFS algorithm with cutoff, Section 4 presents a parameterized procedure in Algorithm 4 making it possible to assert the existence of a path between two nodes. Section 5 introduces a parameterized procedure in Algorithm 5 to evaluate the feasibility of a move by direct application of Algorithm 4. Section 6 is devoted to the development of a generic procedure for the computation of valid lower bounds for a move and any regular objective function.

4. Parameterized procedure for path detection

To ensure the efficiency of a local search heuristic, it is important to identify unfeasible moves and evaluate their quality quickly. Several important results in the scheduling literature addressing these challenges rely mainly on the information on the existence of a path between two nodes in the solution graph. For example, let us consider move $\mathcal{M} = \{u, O = (s, t), D = (v, w)\}$ that sequences u between v and w. To show that the move is feasible (i.e., does not induce a cycle), it is sufficient to show that there is no path from u to v or from w to u. In a general directed graph, classical algorithms such as Depth-First Search (DFS) or Breadth-First Search (BFS) can be used to verify the existence of a path between two given nodes. However, such algorithms are computationally prohibitive and thus not practical if used intensively. Instead, the scheduling data calculated from the current solution DAG, such as heads or tails of nodes, can be used to formulate sufficient conditions inferring the existence of paths between nodes. The low computational cost to check these conditions comes with the drawback of potentially overlooking feasible and promising moves.

The fundamental result on which the other contributions of this article are based is a parameterized procedure that asserts the absence of a path between two nodes in a DAG. The procedure is qualified as parameterized because it allows the trade-off between evaluation accuracy and computational cost to be controlled by setting some key parameters. This section provides the necessary results that support the design of the proposed procedure.

As illustrated in Fig. 4, the basic question to answer can be stated as follows: *Is there a path from node u to node v given the length $(L(\alpha, u), L(\alpha, v))$ of the longest paths from a reference node α?* To answer this question, Lemma 1 provides a sufficient condition, which, if satisfied, asserts the absence of a path from u to v. As shown later in Section 5, several feasibility evaluation procedures in the literature are instantiations of the condition in Lemma 1. For example, by considering the start dummy node as the reference node α, the sufficient condition relies on the heads of operations. As highlighted in Section 3.1, it is also common that evaluation procedures require tails, in combination or not, with the heads of operations. However, the heads and tails are symmetric and differ only in how they are computed using the solution graph G or its associated graph G'. Therefore, Lemma 1 can be used to state another sufficient condition on the absence of a path between two nodes in G that is based on the lengths of the longest paths from

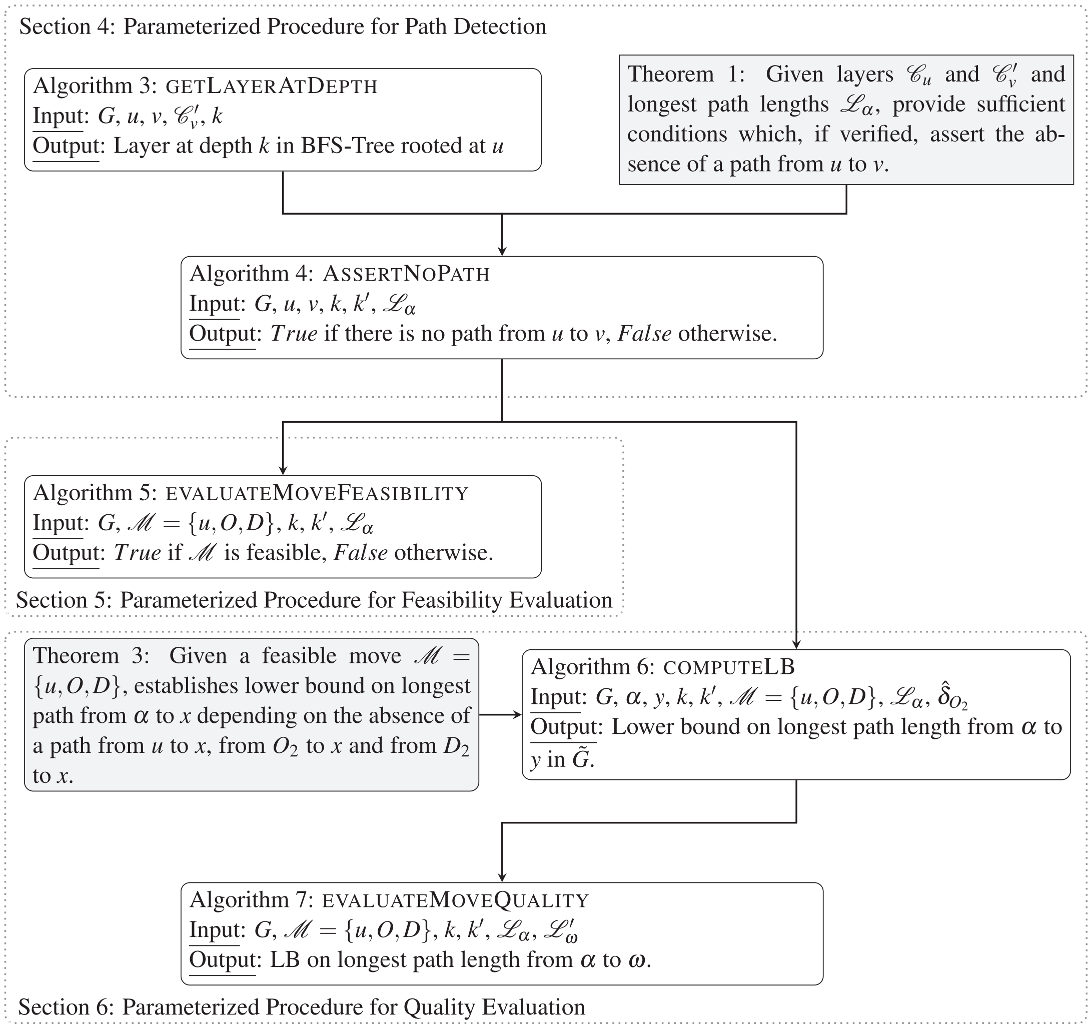
Section 4: Parameterized Procedure for Path Detection
Algorithm 3: GETLAYERATDEPTH
Input: G, u, v, $\mathscr{C}'_v$, k
Output: Layer at depth k in BFS-Tree rooted at u
Theorem 1: Given layers $\mathscr{C}_u$ and $\mathscr{C}'_v$ and longest path lengths $\mathscr{L}_\alpha$, provide sufficient conditions which, if verified, assert the absence of a path from u to v.
Algorithm 4: ASSERTNOPATH
Input: G, u, v, k, k', $\mathscr{L}_\alpha$
Output: $True$ if there is no path from u to v, $False$ otherwise.
Algorithm 5: EVALUATEMOVEFEASIBILITY
Input: G, $\mathscr{M} = \{u, O, D\}$, k, k', $\mathscr{L}_\alpha$
Output: $True$ if $\mathscr{M}$ is feasible, $False$ otherwise.
Section 5: Parameterized Procedure for Feasibility Evaluation
Theorem 3: Given a feasible move $\mathscr{M} = \{u, O, D\}$, establishes lower bound on longest path from α to x depending on the absence of a path from u to x, from O_2 to x and from D_2 to x.
Algorithm 6: COMPUTELB
Input: G, α, y, k, k', $\mathscr{M} = \{u, O, D\}$, $\mathscr{L}_\alpha$, $\hat{\delta}_{O_2}$
Output: Lower bound on longest path length from α to y in $\tilde{G}$.
Algorithm 7: EVALUATEMOVEQUALITY
Input: G, $\mathscr{M} = \{u, O, D\}$, k, k', $\mathscr{L}_\alpha$, $\mathscr{L}'_\omega$
Output: LB on longest path length from α to ω.
Section 6: Parameterized Procedure for Quality Evaluation

Fig. 3. Structure of paper.

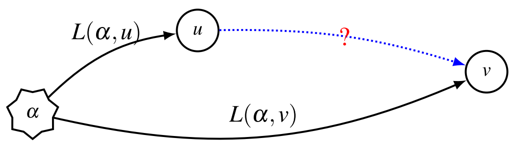

Fig. 4. Existence of a path between nodes u and v based on the lengths of the longest paths from reference node α to u and v.

reference node ω in the reverse graph G'. This new sufficient condition is given in Corollary 1. The sufficient condition relies on the operation tails when taking the last dummy node as the reference node ω.

Lemma 1. *If $L(\alpha, v) < L(\alpha, u) + l_u + l_{out(u)}$, then there is no path from u to v.*

Corollary 1. *If $L'(\omega, u) < L'(\omega, v) + l_v + l_{out'(v)}$, then there is no path from u to v in G.*

The satisfaction of the condition in Lemma 1 is sufficient to guarantee that there is no path from u to v in G. However, knowing that the inequality is false does not ensure that there is an actual path from u to v. For example, if there is no path from α to u, then the condition is never satisfied. Consequently, the low computation time to verify the sufficient condition (i.e., $O(1)$) has the disadvantage of potentially not recognizing the absence of a path from u to v. When used in a feasibility evaluation procedure, for example, this sufficient condition may lead to rejecting feasible moves. Given a set of node pairs for which the absence of a path must be asserted using the sufficient condition, let us define the *miss rate* as the proportion of node pairs for which the condition fails at guaranteeing the absence of a path. Suppose the miss rate of the sufficient condition is responsible for the poor performance of the local search heuristic. In that case, it may be necessary to formulate new conditions with a lower miss rate. Before providing the main intuition behind the parameterized procedure introduced in this section, it may be useful to rephrase the result in Lemma 1. Let us associate to node u the interval $]-\infty, L(\alpha, u) + l_u + l_{out(u)}[$. Lemma 1 ensures that there is no path from u to any node $v \in V$ such that $L(\alpha, v) \in]-\infty, L(\alpha, u) + l_u + l_{out(u)}[$. As soon as $L(\alpha, v) \geq L(\alpha, u) + l_u + l_{out(u)}$, Lemma 1 fails to differentiate between the existence and the absence of a path from u to v.

Assuming the need to reduce the miss rate of the condition $L(\alpha, v) \in]-\infty, L(\alpha, u) + l_u + l_{out(u)}[$, the intuition is to replace $L(\alpha, v)$ with an evaluation that gives a lower value, or increase the upper bound of the interval (i.e., $L(\alpha, u) + l_u + l_{out(u)}$) with an evaluation that gives a larger value. Recall that, in a graph with non-negative weights, the lengths along any path are non-decreasing (Property 1). As the length of the longest path from α to any predecessor of v is smaller than $L(\alpha, v)$, using the longest path from α to predecessors of v instead of v can yield conditions with lower miss rate. In the same way, better conditions can be formulated

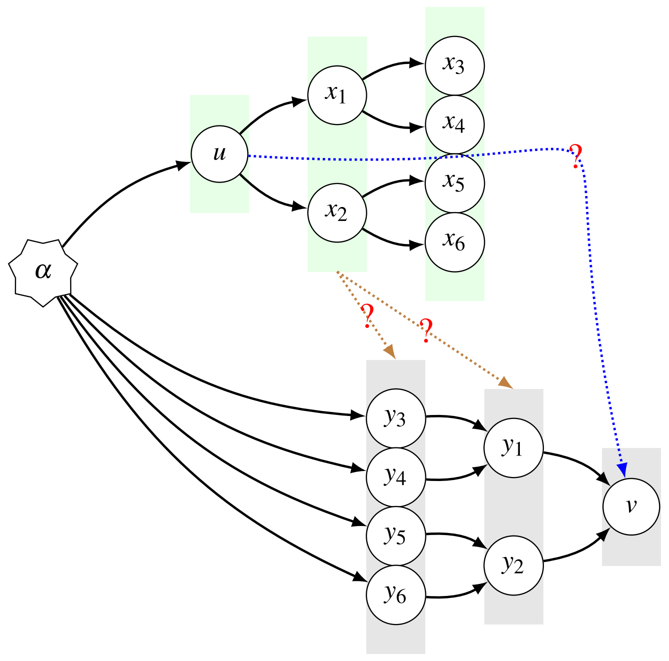

Fig. 5. Existence of a path from u to v based on the lengths of the longest paths from a reference node α to the successors and predecessors of u and v.

using the successors of u instead of u. This logic can be taken further to consider the successors of the successors and the predecessors of the predecessors. Figure 5 illustrates this idea of using the successors of u and the predecessors of v to characterize the existence of a path from u to v. The successors of u are the two nodes in the set $\{x_1, x_2\}$ which have as successors the nodes in the set $\{x_3, x_4, x_5, x_6\}$. The predecessors of v are the two nodes in the set $\{y_1, y_2\}$ which have as predecessors the nodes in the set $\{y_3, y_4, y_5, y_6\}$. For instance, showing that there is no path from any node in $\{x_1, x_2\}$ to any node in $\{y_1, y_2\}$ ensures that there is no path from u to v.

The idea of using, for instance, the set of direct successors of u relies on the fact that, if there is a path from u to v, then this path must go through at least one of the successors of u. Therefore, if there is no path from none of the direct successors of u to v, it is certain that there is no path from u to v. Similarly, if there is no path from u to any of the direct predecessors of v, there is no path from u to v. Instead of u or v, the condition in Lemma 1 can be based on a set of nodes succeeding u or preceding v respectively. However, the chosen set must satisfy the following requirement: Any path starting from u (reaching v) should pass through at least one of the nodes in the set associated with u (v).

As mentioned earlier, the BFS algorithm can be used to determine the existence or absence of a path between two nodes. Given a graph $G = (V, A)$ and a distinguished source node u, BFS systematically explores the arcs of G to discover each node reachable from u. The algorithm discovers all nodes at a distance d from u before discovering nodes at a distance $d + 1$; the distance is defined as the number of arcs in the shortest path between two nodes. Given a graph $G = (V, A)$ and a distinguished source node u, BFS produces the so-called *BFS-tree* rooted at u. For any node v reachable from u, the simple path in the BFS tree from u to v corresponds to a shortest path from u to v in G. The term *layer* is generally used to qualify the nodes of the BFS tree rooted at u at the same depth (Dasgupta et al., 2008) and is denoted in the rest of the article $\mathcal{C}_u$. Using such an algorithm to assert the existence of a path from u to v is not attractive in a local search heuristic because its running time is $O(|V| + |A|)$ (Dasgupta et al., 2008). However, as pointed out in Property 2, a layer in a BFS tree has the property that makes it a suitable candidate for formulating sufficient conditions with a lower miss rate.

Property 2. *Let G be a DAG and $\mathcal{C}_u$ a layer in the BFS-tree rooted at u and at depth k. Any path starting from u in G goes at least trough one of the nodes in $\mathcal{C}_u$.*

The basis of the proposed parameterized procedure that asserts the existence of a path between two nodes is the modified BFS algorithm in Algorithm 3. As this procedure is used to check if there is a path from u to a given node in G, the destination node is provided to the algorithm as v. The third parameter $\mathcal{C}'_v$ denotes a layer associated with v in the reverse graph G', i.e., a set of nodes preceding v in G. At first, we consider that $\mathcal{C}'_v = \{v\}$, i.e., the condition in line 13 is equivalent to $y = v$. Instead of exploring all the nodes that are reachable from u, k represents the search *cutoff*. In other words, the algorithm stops the search at the level of the nodes at depth k. The algorithm uses the same queue data structure as in Algorithm 2 and stores a set of nodes along with their distance d from u. Initially, the queue Q only consists of u, the one node at a distance 0. When node x with distance d is removed from Q, all of its successors that are not discovered yet are inserted in Q along with their distance $d + 1$.

Algorithm 3 Search for a layer related to a path from u to v at given depth k.

```
1:  procedure GETLAYERATDEPTH($G$, $u$, $v$, $\mathcal{C}'_v$, $k$)
2:      $\mathcal{C}_u \leftarrow \emptyset$
3:      $Q \leftarrow \emptyset$
4:      Label $u$ as discovered
5:      $enqueue(Q, (u, 0))$
6:      while $Q$ is not empty do
7:          $x, d \leftarrow dequeue(Q)$
8:          if $d == k$ then
9:              $\mathcal{C}_u \leftarrow \mathcal{C}_u \cup \{x\}$
10:             continue
11:         for $(x, y) \in out(x)$ do
12:             if $y$ is not discovered then
13:                 if $y \in \mathcal{C}'_v$ then
14:                     $\mathcal{C}_u \leftarrow \{v\}$
15:                     return $\mathcal{C}_u$
16:                 Label $y$ as discovered
17:                 $enqueue(Q, (y, d + 1))$
18:     return $\mathcal{C}_u$
```

The three following different results can be derived from Algorithm 3:

Case $\mathcal{C}_u = \emptyset$. If k is large enough, the algorithm explores all the nodes that can be reached from u, and the algorithm may or may not find v. If v is not found and all the nodes that can be reached from u are explored, we are sure that there is no path from u to v. Because $\mathcal{C}_u$ is only updated if the conditions in Lines 8 or 13 are satisfied, the algorithm returns an empty set when k is very large and v is not found. Hence, if an application requires the existence or absence of a path between two nodes to be guaranteed, k should be larger than the graph diameter, i.e., the length of the longest shortest path in G after disregarding the node and arc weights.

Case $\mathcal{C}_u = \{v\}$. If, during the search, v is found, the search is stopped and a set containing only v is returned as a layer $\mathcal{C}_u$ (Line 15), and we are sure that there is a path from u to v.

Case $\mathcal{C}_u \neq \emptyset$ **and** $\mathcal{C}_u \neq \{v\}$. In the two first cases, $\mathcal{C}_u$ is not a layer but is rather used to provide information with certainty on the absence ($\mathcal{C}_u = \emptyset$) or presence ($\mathcal{C}_u = \{v\}$) of a path from u to v. The last case occurs when k is not large enough to allow the algorithm to explore all the nodes that can be reached from u, and if v is reachable from u, the length of the shortest path from u is larger than k. In this case, the algorithm returns an actual layer $\mathcal{C}_u$ that includes all the nodes that are at depth k in the BFS-tree rooted at u (Line 9). Contrary to the two first cases, this third case does not provide any information on the existence or absence of a path from u to v. Therefore, Lemma 2 states a new condition regarding the absence of a path from u to v in G using the notion of a layer $\mathcal{C}_u$. When $k = 0$, Algorithm 3 returns $\mathcal{C}_u = \{u\}$, making the condition in Lemma 2 the same as the condition in Lemma 1. Lemma 3 shows that the sufficient condition Lemma 2 can have a lower miss rate condition in Lemma 1 provided that $k > 0$.

Lemma 2. *Let $\mathcal{C}_u = \text{GETLAYERATDEPTH}(G, u, v, \{v\}, k)$ be a layer, i.e., $\mathcal{C}_u \neq \emptyset$ and $v \notin \mathcal{C}_u$.*

If $L(\alpha, v) < \min_{x \in \mathcal{C}_u}(L(\alpha, x) + l_x + l_{out(x)})$, then there is no path from u to v.

Lemma 3. *If the condition in Lemma 1 is satisfied, then the condition in Lemma 2 is also satisfied.*

As a modified BFS, Algorithm 3 provides either information about the existence of a path between two nodes or a layer that can be used by sufficient conditions in Lemma 2. The procedure is parameterized because the search depth can be controlled via parameter k. Increasing the value of k increases the probability of falling in both cases $\mathcal{C}_u = \emptyset$ and $\mathcal{C}_u = \{v\}$, respectively providing certainty on the absence or presence of a path from u to v. If the algorithm falls into the third case, increasing the value of k increases the upper bound of the interval $]-\infty, \min_{x \in \mathcal{C}_u}(L(\ alpha, x) + l_x + l_{out(x)})[$, leading to the decrease of the miss rate of the sufficient condition. However, increasing the value of the search cutoff k also increases the computational cost. When k receives a large value, Algorithm 3 has the same execution time as the classic BFS, i.e. $O(|V| + |A|)$. However, this execution time does not consider the possibility of stopping the search at depth k. Instead of using the size of V and E, it is more convenient to describe the complexity of Algorithm 3 based on another feature of the G graph, the *branching factor*. The branching factor of the nodes of a graph G is defined as the number of direct successors of the node, i.e., the size of the set of successors. If the branching factors of the nodes are not uniform, we can consider the maximum branching factor, noted b. To find nodes that are at distance k from node u, Algorithm 3 takes $O(b^k)$ time. This exponential time complexity, usually used or graphs too large to store explicitly (Korf, 1985) must be carefully considered. First, remember that the running time of Algorithm 3 is bounded by $O(|V| + |A|)$. Moreover, as indicated in the following sections, most of the existing results can be obtained by the proposed procedures using $k \leq 2$ with $b = 2$ for the most commonly studied problems.

When studying the existence of a path from u to v, recall that we considered above that $\mathcal{C}'_v = \{v\}$ is given as an input to Algorithm 3 which returns $\mathcal{C}_u$. The search could be performed backward, i.e., started from v to find all the nodes that can reach v. To do this, it is sufficient to provide Algorithm 3 the following parameters: G', v, u, $\mathcal{C}_u = \{u\}$ and a depth k'. The algorithm determines a set of nodes such that, if there is a path starting from v in G', this path must pass at least one of the nodes in $\mathcal{C}'_v$. In other words, if there is a path from u to v, then this path must pass through at least one of the nodes in $\mathcal{C}'_v$. Theorem 1 formulates general sufficient conditions that use the layers determined by the forward search from u and the backward search from v. Using the same arguments as in Lemma 3, it can be shown that Theorem 1 generalizes Lemma 2.

Theorem 1. *There is no path from u to v if the two layers $\mathcal{C}_u = \text{GETLAYERATDEPTH}(G, u, v, \{v\}, k)$ and $\mathcal{C}'_v = \text{GETLAYERATDEPTH}(G', v, u, \mathcal{C}_u, k')$ satisfy the following conditions:*

1. *$\mathcal{C}_u \neq \emptyset$ and $\mathcal{C}_u \neq \{v\}$, and*
2. *$\mathcal{C}'_v \neq \emptyset$ and $\mathcal{C}'_v \neq \{u\}$, and*
3. *$\max_{y \in \mathcal{C}'_v} L(\alpha, y) < \min_{x \in \mathcal{C}_u}(L(\alpha, x) + l_x + l_{out(x)})$*

The results presented in this section are combined to design a parameterized procedure that determines if there is a path between two nodes u and v in a DAG G with non-negative weights. Algorithm 4 relies on Algorithm 3 in a forward search from u and a backward search from v. Parameters k and k' are the forward and backward search depths, respectively. Note that a single parameter can replace these two parameters. However, two parameters are necessary to show how the proposed algorithm can reproduce known results of the literature. The last procedure parameter $\mathcal{L}_\alpha$ represents the vector of the length of the longest path from α to all nodes of G, i.e., $\mathcal{L}_\alpha = (L(\alpha, v) | v \in V)$.

The layer $\mathcal{C}_u$ is obtained by initializing $\mathcal{C}'_v = \{v\}$. Algorithm 4 terminates by returning *True* (i.e., there is no path from u to v) or *False* (i.e., there is a path from u to v) when $\mathcal{C}_u = \emptyset$ or $\mathcal{C}_u = \{v\}$, respectively. If these two cases do not occur during the forward search, then it means that v is not found and that there are nodes that can be reached from u and that are not discovered yet. At this stage, the sufficient condition in Lemma 2 can be checked to assert the potential existence of a path from $\mathcal{C}_u$ to v. However, it is possible to run the backward search from v before using the sufficient condition. With the resulting set $\mathcal{C}_u$, the layer $\mathcal{C}'_v$ is recomputed according to the given depth k'. The difference with the computation of $\mathcal{C}_u$ is that $\mathcal{C}'_v$ is computed considering a layer $\mathcal{C}_u$ that does not contain u if $k > 0$. In this case, if $u \in \mathcal{C}'_v$, then it means that the backward search found one of the nodes in $\mathcal{C}_u$, which asserts with certainty that a path exists from u to v. If $\mathcal{C}'_v = \emptyset$, then u cannot be reached in G' from v, i.e., there is no path from u to v in G. If the resulting $\mathcal{C}'_v$ does not satisfy the two previous conditions, then none of the nodes in $\mathcal{C}_u$ is found by the backward search and not all nodes that can be reached from v in G' are found. In this case, the condition in Theorem 1 can be used. If the condition is satisfied, Theorem 1 ensures that there is no path from u to v. Otherwise, the *False* of Line 15, italicized to differentiate it from the others, does not guarantee the existence of a path but rather the inability of the sufficient condition to conclude. By increasing the value of k or k', the chances of executing Line 15 decrease, resulting in a lower miss rate.

Algorithm 4 Parametrized procedure for path detection from u to v.

```
1:  procedure AssertNoPath($G$, $u$, $v$, $k$, $k'$, $\mathcal{L}_\alpha$)
2:      $\mathcal{C}'_v \leftarrow \{v\}$
3:      $\mathcal{C}_u \leftarrow$ GETLAYERATDEPTH($G$, $u$, $v$, $\mathcal{C}'_v$, $k$)
4:      if $\mathcal{C}_u = \emptyset$ then
5:          return True
6:      if $\mathcal{C}_u = \{v\}$ then
7:          return False
8:      $\mathcal{C}'_v \leftarrow$ GETLAYERATDEPTH($G'$, $v$, $u$, $\mathcal{C}_u$, $k'$)
9:      if $\mathcal{C}'_v = \emptyset$ then
10:         return True
11:     if $\mathcal{C}'_v = \{u\}$ then
12:         return False
13:     if $\max_{y \in \mathcal{C}'_v} L(\alpha, y) < \min_{x \in \mathcal{C}_u}(L(\alpha, x) + l_x + l_{out(x)})$ then
14:         return True
15:     return *False*
```

Note that we assume that the sufficient conditions in Algorithm 4 use $\mathcal{L}_\alpha$, the lengths of the longest paths from a reference node α in G. It is also possible to use the lengths of the longest paths from a reference node ω in G'. Assuming that the lengths from ω are given in an the vector $\mathcal{L}'_\omega$, the existence of a path from u to v in G can be checked using $\mathcal{L}'_\omega$ by calling AssertNoPath$(G', v, u, k', k, \mathcal{L}'_\omega)$. In conclusion, Algorithm 4 provides a parameterized procedure that can be called to answer the question of whether there is a path u to v in a DAG with non-negative weights.

Such an algorithm makes it possible, through the search cutoff parameters k and k', to decide on the trade-off between computational efficiency and miss rate. By calling Algorithm 3 and given that the size of a layer is at most $O(b^k)$, the running time of Algorithm 4 is $O(b^{\max\{k,k'\}})$. This procedure is an essential block in the procedures evaluating move feasibility and move quality that are presented in the following sections.

5. Parameterized procedure for feasibility evaluation

This section focuses on the feasibility evaluation and presents a parameterized procedure that determines whether a move leads to a feasible neighbor. Such procedure can typically be called in Line 6 of Algorithm 1. Several important results in the scheduling literature addressing this question rely on the information on the existence of a path between two nodes in the solution graph. Similarly, the procedure for feasibility evaluation proposed in Section 5.1 heavily relies on the procedure for path detection presented in Section 4. Then, Section 5.2 shows that a selection of literature results are special cases of the procedure.

5.1. Procedure for move feasibility evaluation

Given a DAG G that models a feasible solution, the goal is to decide whether a move $\mathcal{M}$ as defined in Section 3.2 is feasible, i.e., whether the neighbor modeled by $\tilde{G}$ does not include a cycle. Algorithm 5 formalizes the proposed procedure that asserts whether a move $\mathcal{M}$ is feasible or not. After applying a move, let us assume there is a cycle in $\tilde{G}$ and the arc (x, y) is part of the cycle. Such assumption implies the existence of a path from y to x. Therefore, to ensure that the insertion of an arc (x, y) in G does not create a cycle, it is sufficient to ensure that there is no path from y to x, and this information can be obtained using Algorithm 4. The procedure for move feasibility is thus a "simple" application of the procedure for path detection of Section 4. To make the procedure applicable to both G and G', the general notation of a move $\mathcal{M} = \{u, O, D\}$ is used. In addition to the current solution graph G and a move $\mathcal{M}$, Algorithm 5 requires additional parameters for Algorithm 4. Parameters k, resp. k', are the depths of the forward, resp. backward, search. The last procedure parameter $\mathcal{L}_\alpha$ is the vector of the lengths of the longest paths from α to all nodes of G, i.e., $\mathcal{L}_\alpha = (L(\alpha, v)|v \in V)$.

Algorithm 5 General procedure for move feasibility.

```
1: procedure evaluateMoveFeasibility($G$, $\mathcal{M} = \{u, O, D\}$, $k$, $k'$, $\mathcal{L}_\alpha$)
2:     $A \leftarrow A \setminus \{(O_1, u), (u, O_2), (D_1, D_2)\}$ ▷ $A$ should be restored at end of procedure
3:     $\mathcal{I} \leftarrow \{(D_1, u), (u, D_2)\}$
4:     for $(x, y) \in \mathcal{I}$ do
5:         if AssertNoPath($G$, $y$, $x$, $k$, $k'$, $\mathcal{L}_\alpha$) then
6:             continue
7:         return False
8:     return True
```

For clarity, let us focus on the case where the original graph of schedule G is used. Suppose that operation u is sequenced in G between operations $O_1 = s$ and $O_2 = t$ on resource l, i.e., $O = (s, t)$. The move $\mathcal{M} = \{u, O, D\}$ consists in moving operation u between operations $D_1 = v$ and $D_2 = w$ on resource m, where m can be different from l. The move $\mathcal{M}$ induces the deletion of a set of arcs, denoted $\mathcal{D} = \{(s, u), (u, t), (v, w)\}$, and the insertion of a set of new arcs, denoted $\mathcal{I} = \{(s, t), (v, u), (u, w)\}$. If G is a DAG and $\tilde{G}$ contains a cycle, then the arcs added or deleted by applying the moves are responsible of introducing such a cycle. More specifically, as no cycle can be created by deleting an arc in G, it is sufficient to check whether a cycle is created as a consequence of inserting an arc in $\mathcal{I}$. Also, it can be shown that the insertion of $(D_1, D2) = (s, t) \in \mathcal{I}$ cannot lead to a cycle in the graph.

The procedure starts then by locally deleting the set of arcs $\mathcal{D}$ induced by move $\mathcal{M}$ from the set of arcs A of G. "Locally" means that the original set of arcs A should be recovered at the end of the procedure. Then, the set of arcs to be inserted to get the complete graph of the neighbor is initialized, i.e., $\mathcal{I} = \{(v, u), (u, w)\}$. Note that the arc $(O_1, O_2) = (s, t)$ is not included, as it is certain that no cycle can result from its insertion in G. To ensure that the insertion of an arc (x, y) does not lead to a cycle, Algorithm 4 checks if there is no path from y to x. Theorem 2 states that the moves characterized as feasible by Algorithm 5 lead to feasible solutions. By calling Algorithm 4, the running time of Algorithm 5 is also $O(b^{\max\{k,k'\}})$.

Theorem 2. *Let $\mathcal{M} = \{u, O, D\}$ be a move specified by the neighborhood structure. Let k and k' respectively denote the depths of the forward search and backward search, and let $\mathcal{L}_\alpha$ denote the vector of the lengths of the longest paths from a reference node α to all nodes of G.*

If evaluateMoveFeasibility$(G, \mathcal{M}, k, k', \mathcal{L}_\alpha)$ = **True** *(Algorithm 5), then $\mathcal{M}$ is a feasible move.*

Algorithm 5 is illustrated above when using the vector of the lengths of the longest paths from α to all nodes of the original graph G. The procedure can also be used when needed to use the vector of the lengths of the longest paths $\mathcal{L}'_\omega$ from ω to all nodes of the reverse graph G'. The procedure should get the following parameters: G', $\mathcal{M}' = \{u, O', D'\}$, k', k and $\mathcal{L}'_\omega$. If $O = (s, t)$ and $D = (v, w)$ as above, then $O' = (t, s)$ and $D' = (w, v)$. The procedure then starts by locally deleting the set of arcs $\mathcal{D}' = \{(t, u), (u, s), (w, v)\}$ induced by move $\mathcal{M}'$ from the set of arcs A' of G'. Next, the set of arcs to be inserted to get the complete graph of the neighbor is initialized, i.e., $\mathcal{I}' = \{(w, u), (u, v)\}$. The remainder of the procedure follows as above.

5.2. Relationship with results of the literature

This section shows the generality of our procedure by proving that a selection of classical and well-known results from the literature can be derived using specific parameter values. Table 2 provides the parameter setting for the results in ten references. The problem solved by each reference is denoted using the classical notation $\alpha|\beta|\gamma$ of Graham et al. (1979). In the α field, J, FJ and $FMRJ$ respectively denote a job-shop scheduling problem, a flexible job-shop scheduling problem and a flexible job-shop scheduling problem with operations requiring multiple resources. In the β field, release dates, sequence-dependent setup times, reentrance, parallel batching with incompatible families and non-linear routes are respectively denoted r_j, s, $recr$, $p-batch, incompatible$ and bom. In the γ field, $C_{\max}$, TWT and reg denotes, respectively, the makespan, the total weighted tardiness, and any regular objective function. The approach of Braune et al. (2013) optimizes any objective function in the form of *min-sum*, which is denoted here as $\sum_j f(C_j)$.

For each reference, Column *Result* reports the original results to be shown as a special case of the procedure in Algorithm 5, which is a "simple" application of Algorithm 4 for each of the arcs to be

Table 2
Some literature results dealing with move feasibility evaluation.

Reference	Problem	Result	Arc	AssertNoPath
Dauzère-Pérès & Paulli (1997)	$FJ\|\|C_{\max}$	Theorem 1	(v,u)	$(G,\ u,\ v,\ 1,\ 0,\ \mathcal{L}_\alpha)$
			(u,w)	$(G,\ w,\ u,\ 0,\ 1,\ \mathcal{L}_\alpha)$
Balas & Vazacopoulos (1998)*	$J\|\|C_{\max}$	Proposition 2.2	(v,u)	$(G',\ v,\ u,\ 0,\ 1,\ \mathcal{L}'_\omega)$
		Proposition 2.3	(u,w)	$(G,\ w,\ u,\ 0,\ 1,\ \mathcal{L}_\alpha)$
Dauzère-Pérès et al. (1998)	$FMRJ\|bom\|C_{\max}$	Theorem 1	(v,u)	$(G,\ u,\ v,\ 1,\ 0,\ \mathcal{L}_\alpha)$
			(u,w)	$(G,\ w,\ u,\ 0,\ 1,\ \mathcal{L}_\alpha)$
Mastrolilli & Gambardella (2000)*	$FJ\|\|C_{\max}$	Section 4.1	(v,u)	$(G,\ u,\ v,\ 0,\ 1,\ \mathcal{L}_\alpha)$
				$(G',\ v,\ u,\ 0,\ 1,\ \mathcal{L}'_\omega)$
			(u,w)	$(G,\ w,\ u,\ 0,\ 1,\ \mathcal{L}_\alpha)$
				$(G',\ u,\ w,\ 0,\ 1,\ \mathcal{L}'_\omega)$
Zhang et al. (2007)*	$J\|\|C_{\max}$	Theorem 1	(v,u)	$(G',\ v,\ u,\ 0,\ 1,\ \mathcal{L}'_\omega)$
		Theorem 2	(u,w)	$(G,\ w,\ u,\ 0,\ 1,\ \mathcal{L}_\alpha)$
Braune et al. (2013)*	$J\|\|\sum_j f(C_j)$	Proposition 2	(v,u)	$(G',\ v,\ u,\ 0,\ 2,\ \mathcal{L}'_\omega)$
		Proposition 3	(u,w)	$(G,\ w,\ u,\ 0,\ 1,\ \mathcal{L}_\alpha)$
Sobeyko & Mönch (2016)	$FJ\|r_j, bom\|TWT$	Theorem 1	(v,u)	$(G,\ u,\ v,\ 1,\ 0,\ \mathcal{L}_\alpha)$
			(u,w)	$(G,\ w,\ u,\ 0,\ 1,\ \mathcal{L}_\alpha)$
Knopp et al. (2017)	$FJ\|r_j, s, recr, p-batch, incompatible\|reg$	Theorem 2	(v,u)	$(G,\ u,\ v,\ 1,\ 0,\ \mathcal{L}_\alpha)$
			(u,w)	$(G,\ w,\ u,\ 0,\ 1,\ \mathcal{L}_\alpha)$
Shen et al. (2018)	$FJ\|s\|C_{\max}$	Proposition 4.1	(v,u)	$(G,\ u,\ v,\ 1,\ 0,\ \mathcal{L}_\alpha)$
			(u,w)	$(G,\ w,\ u,\ 0,\ 1,\ \mathcal{L}_\alpha)$
		Proposition 4.2	(v,u)	$(G,\ u,\ v,\ 1,\ 1,\ \mathcal{L}_\alpha)$
			(u,w)	$(G,\ w,\ u,\ 1,\ 1,\ \mathcal{L}_\alpha)$
Kasapidis et al. (2021)	$FJ\|bom\|C_{\max}$	Theorem 1	(v,u)	$(G,\ u,\ v,\ 1,\ 0,\ \mathcal{L}_\alpha)$
			(u,w)	$(G,\ w,\ u,\ 0,\ 1,\ \mathcal{L}_\alpha)$

inserted in G (i.e., $\mathcal{I} = \{(v,u), (u,w)\}$). As the setting are heterogeneous for two arcs within the same work, Table 2 reports the setting of Algorithm 4. All the proofs showing how the settings reported in the last column can reproduce the corresponding literature result are provided in the Supplementary Material accompanying this article to avoid overloading the paper. Note that a star follows some references in the table when the set of feasible moves defined by the original sufficient conditions is a subset of the set of feasible moves determined by our procedure with the settings in the last column. Below are a few remarks regarding the analysis of Table 2 and the referenced works:

- As highlighted previously, the literature results use heads or tails. The heads correspond to the longest path length $\mathcal{L}_\alpha$ in G with source node α considered as a reference node. The tails correspond to the longest path length $\mathcal{L}'_\omega$ in the reverse graph G' from sink node ω considered as a reference node. The parameter setting of a result published in Klemmt et al. (2017) using the node levels is provided in the Supplementary Material.
- The depth search cutoff necessary to reproduce the literature results is low. Note that most of the approaches proposed in the references in Table 2 were or are still state of the art. Therefore, it may be unnecessary to use high values for k and k', resulting in the high efficiency of the proposed procedure for the feasibility evaluation.
- Shen et al. (2018) report numerical results that can be used to understand the impact of increasing the search depth cutoff. Instead of using $k = 1$ and $k' = 0$ to assert the absence of a path from u to v in Proposition 4.1, Proposition 4.2 uses $k = 1$ and $k' = 1$. On average, the numerical results show that the quality of the final solution is improved with an additional computational time. The main reason for such an increase is the larger neighborhood to be explored. As the set of feasible neighbors determined by Proposition 4.2 includes the one determined by Proposition 4.1, the proposed tabu search spends more time evaluating the neighborhood of a solution. This indicates that, when using Algorithm 5, it might be necessary to experimentally choose the most appropriate search depth cutoff for the considered problem instances.
- By increasing the value of the search depth cutoff, it is possible to lower the miss rate of the sufficient conditions. The work of Mastrolilli & Gambardella (2000) shows another interesting alternative. For example, to show that the insertion of (v,u) does not create a cycle, Algorithm 4 can be called on G and G' using the same low search depth cutoff.

6. Parameterized procedure for quality evaluation

In addition to the move feasibility evaluation, the quality evaluation of a move is another critical element that is often discussed in the literature on heuristic approaches for the job-shop scheduling problem and its extensions. This section shows that several theoretical results published on this topic share the same insights and fit in the general framework proposed here. An important piece of information that implicitly supports the design of evaluation procedures is the existence of a path between two nodes. Therefore, the proposed procedure for move evaluation heavily relies on the results in Sections 4. A short literature review regarding quality evaluation procedures is first presented in Section 6.1. Then, Section 6.2 introduces the parameterized procedure for quality evaluation along with its underlying theoretical results. To show the broad applicability of the proposed parameterized procedure, a selection of literature results are shown to be special cases of the procedure in Section 6.3.

6.1. Literature review

In the scheduling literature, the quality evaluation of a neighbor is one of the key features that differentiate the different proposed local search heuristic approaches. To evaluate the quality of a move, the straightforward approach is to perform the move and compute the value of the optimized objective function (e.g., Nowicki & Smutnicki, 1996). This exact evaluation can be performed by using the procedure in Algorithm 2, which appears to make the evaluation the most expensive component in heuristic approaches to solving shop scheduling problems (Taillard, 1994). An alternative recently regaining attention is to incrementally maintain the longest path lengths and only update the information of the nodes affected by the move. Bierwirth & Kuhpfahl (2017); Mati et al. (2011); Nowicki & Smutnicki (2005); Sobeyko & Mönch (2016) and Madraki & Judd (2021) propose some algorithms that speed up the evaluation without reaching the full potential of this approach. Those algorithms might visit nodes that are not affected by the move. The algorithms proposed in Katriel et al. (2005) satisfy the attractive property of visiting only affected nodes but cannot be applied to most of the neighborhood structures of job-shop scheduling problems.

Instead of an exact evaluation, it is more common in the scheduling literature to consider an approximate evaluation while accepting some accuracy loss. The available approximate evaluation procedures may verify the lower bound property (Dauzère-Pérès & Paulli, 1997; Taillard, 1994), the upper bound property (Mastrolilli & Gambardella, 2000) or none of the two properties (Balas & Vazacopoulos, 1998; Dell'Amico & Trubian, 1993). While most quality evaluation procedures focus on the makespan, Mati et al. (2011), resp. Braune et al. (2013), propose general procedures for any regular objective function, resp. min-sum objective functions. In addition to the lower bound properties, the upper bound properties and the evaluated objective function, it is possible to classify the quality evaluation procedures based on the move they can be applied to. Some evaluation procedures are restricted to swap moves (e.g. Mati et al., 2011; Taillard, 1994) and some can be applied to more general insertion moves (e.g. Dauzère-Pérès & Paulli, 1997; Dell'Amico & Trubian, 1993).

When analyzing the procedures proposed in the literature, for efficiency considerations, the evaluation relies on the length of paths (old and new ones) that include the nodes directly impacted by the move but also of paths that do not include such nodes. Such evaluation was initially proposed in Taillard (1994) for the case of the job-shop scheduling problem with makespan minimization. To evaluate the effect of swapping a single critical arc (u, v) (see Fig. 2), Taillard (1994) computes the exact value of the longest path which contains at least one of the nodes u and v in the graph associated with the new solution. The length of this path is a valid lower bound on the objective function value of the new solution. In addition to the nodes directly impacted by the move, Mati et al. (2011) proposes to improve the lower bound accuracy by considering a subset of paths that do not go through u or v.

As highlighted earlier, the existence or absence of a path between two nodes is a property used more implicitly when evaluating the quality of a move instead of being used more explicitly in the move feasibility evaluation. For example, the absence of some paths can be indirectly deduced by classifying the moves based on the position of u in its new insertion position (v, w). The distinction between *forward* and *backward* moves can be found in several papers such as Balas & Vazacopoulos (1998); Braune et al. (2013) and García-León et al. (2015). This distinction allows to implicitly assert the absence of paths between some nodes. For the job-shop scheduling problem (Balas & Vazacopoulos, 1998), and (Braune et al., 2013), these notions are characterized through the move direction of the resequenced operation on its machine sequence. In the case of the forward move, it can be ensured that there cannot be a path from w to u only by assuming that the current solution is feasible. For the flexible job-shop scheduling problem (García-León et al., 2015), the characterization is performed through the use of the notion of the level of the operation to be moved u and its new resource predecessor v: Forward insertion when $l_u \leq l_v$ and backward insertion when $l_u > l_v$. Other similar classifications of the moves can be found in the literature.

In this work, we aim at designing a generic procedure that can be used to compute valid lower bounds for any regular objective function. The calculation of the lower bound on the objective function is based on the calculation of lower bounds on the heads and the tails of a set of nodes, most of the time those affected by the move to perform. Estimating the new heads and tails is based on their values in the original solution. An important piece of information to have when calculating a node's estimated heads and tails is whether the node under consideration is reachable from another node whose head or tail changes due to the insertion or removal of an incident arc. This information can be found using the results in Section 4.

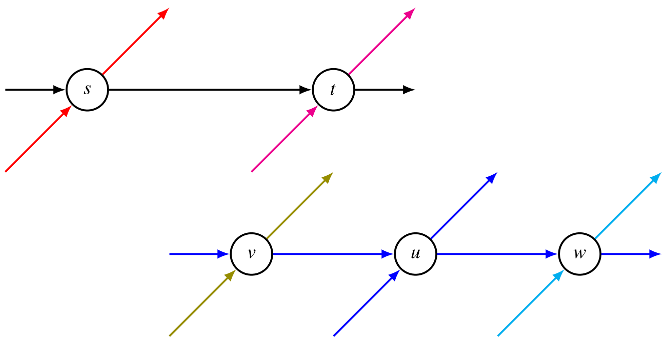

Fig. 6. Moving node u from between s and t to between v and w.

6.2. Procedure for move quality evaluation

Similar to a procedure that evaluates the feasibility of a move, a quality evaluation procedure strongly depends on the definition of a move. We use the move defined in Section 3.2. Let $\mathcal{M} = \{u, O, D\}$ denote a move specified by the neighborhood structure, i.e., a node u is moved from between O_1 and O_2 to between D_1 and D_2. Given a graph G that models a feasible solution, the goal is to compute a valid lower bound on the optimized objective function of the neighbor obtained after applying $\mathcal{M}$ and modeled by $\tilde{G}$. As we consider any regular objective function, the computation of a lower bound on the optimized objective function requires the estimation of the length of the longest paths from a source node α to a set of sink nodes $\{\omega_j\} \subset V$. Therefore, we focus below on the length estimation of the longest path from α to some sink node ω. The exact evaluations related to the neighbor $\tilde{G}$ are referred to by the symbol $\tilde{\square}$, while the approximate evaluations are signaled by the use of symbol $\hat{\square}$. For example, the length of the longest path from α to ω in $\tilde{G}$ is denoted by $\tilde{L}(\alpha, \omega)$, while its lower bound is denoted by $\hat{L}(\alpha, \omega)$. The reverse graph of $\tilde{G}$ is denoted $\tilde{G}'$.

The quality evaluation procedures proposed in the literature rely on the length of paths (before and after a move) that include the nodes directly impacted by the move, but also on paths that do not include such nodes. We focus here on the nodes directly impacted by the move, i.e., u, O_1, O_2, D_1 and D_2. To obtain a valid lower bound on the length of the longest paths from α to ω, it is sufficient to compute valid lower bounds on the paths going through the set of the directly affected nodes. Assume that the feasible move to apply is $\mathcal{M} = \{u, O = (s, t), D = (v, w)\}$. Among the different ways of defining the subsets of the set of considered paths, we choose the one illustrated in Fig. 6, which helps the most in Section 6.3 to illustrate the generality of the evaluation procedure. The subset of paths going through $O_1 = s$ or $O_2 = t$ comprises three subsets: 1) The set of paths going through the newly inserted arc (s, t) (black), 2) The set of paths going through s (red) and 3) The set of paths going through t (magenta). Note that set 1) is a subset of the two other sets 2) and 3). We allow this overlapping to facilitate mapping the different lower bounds proposed in the literature to those computed by our procedure. Also, the computation of the lower bound of a redundant set does not significantly impact the computational cost of the move quality evaluation. The subset of paths going through $D_1 = v$, u or $D_2 = w$ comprises: 4) The set of paths going through u (blue), 5) The set of paths going through v *but not* the newly inserted arc (v,u) (olive) and 6) The set of paths going through w *but not* the newly inserted arc (u,w) (cyan). The general idea of the proposed procedure is to compute a lower bound for each of the six subsets of paths and takes the maximum as a valid lower bound for the longest path length from α to ω.

If there is a path from α to ω through x, then, by definition, $L(\alpha,x)+l_x+L(x,\omega)$ is a lower bound of $L(\alpha,\omega)$. Therefore, to compute a valid lower bound on the longest path length from α to ω, valid lower bounds on the lengths $\tilde{L}(\alpha,x)$ and $\tilde{L}'(\omega,x)$ must be computed, where x is one of the directly affected nodes. Given a DAG G and its reverse graph G', the same results can be used to establish a lower bound on $\tilde{L}(\alpha,x)$ and $\tilde{L}'(\omega,x)$. Therefore, the following results are stated for the general case of a DAG G with non-negative weights. Assuming a feasible move $\mathcal{M}=\{u,O=(s,t),D=(v,w)\}$ is applied on the solution graph G modeling a feasible schedule, the same result helps establishing, for example, a lower bound on $\tilde{L}(\alpha,s)$ and $\tilde{L}'(\omega,t)$ as $O_1=s$ in G and $O'_1=t$ in G'. As highlighted above, the efficiency of the evaluation procedure is based on the idea of reusing all relevant information of the schedule before the move to establish lower bounds. More specifically, to compute a lower bound on $\tilde{L}(\alpha,y)$ of one of the directly affected nodes y, we first aim at computing lower bounds on the longest path from α to all y predecessors in $\tilde{G}$. Then, a lower bound on $\tilde{L}(\alpha,y)$ can easily be computed as in Lines 9–10 in Algorithm 2.

Given a general feasible move $\mathcal{M}=\{u,O,D\}$, Theorem 3 helps establishing different lower bounds on $\tilde{L}(\alpha,x)$ depending on the existence of paths from some of the directly affected nodes $(\{u,O_2,D_2\})$ by move $\mathcal{M}$.

Theorem 3. *Let $\mathcal{M}=\{u,O,D\}$ be a feasible move specified by the neighborhood structure transforming G to $\tilde{G}$. Let us assume that the set of arcs $\mathcal{D}$ are deleted from A (i.e., $A=A\setminus\mathcal{D}$). For each $x\in V\setminus\{u,O_2,D_2\}$:*

1. *If there is no path from u to x, from O_2 to x and from D_2 to x in G, then $\tilde{L}(\alpha,x)=L(\alpha,x)$.*
2. *If there is no path from u to x and from O_2 to x in G, then $\tilde{L}(\alpha,x)\geq L(\alpha,x)$.*
3. *If there is no path from u to x and from D_2 to x in G, then $L(\alpha,x)\geq\tilde{L}(\alpha,x)\geq L(\alpha,x)-\delta_{O_2}$.*

Using Theorem 3, Corollary 2 establishes lower bounds on the different graph nodes. First, instead of always using Algorithm 4 to assert the absence of a path between two nodes, such a question can be directly answered based only on the feasibility of the current solution and the move. For example, considering that G is a DAG implies that there cannot be a path from u to O_1 and from O_2 to O_1.

Corollary 2. *Let $\mathcal{M}=\{u,O,D\}$ be a feasible move specified by the neighborhood structure transforming G to $\tilde{G}$. Let us assume that the set of arcs $\mathcal{D}$ are deleted from A (i.e., $A=A\setminus\mathcal{D}$). Let $\hat{\delta}_{O_2}$ denote an upper bound on $\delta_{O_2}=L(\alpha,O_2)-\tilde{L}(\alpha,O_2)$. We have the following identities:*

1. *$\hat{L}(\alpha,x)=\tilde{L}(\alpha,x)=L(\alpha,x)$, for each $x\in\mathcal{B}(u)$*
2. *$\hat{L}(\alpha,O_1)=L(\alpha,O_1)$*
3. *If there is no path from u to x and from O_2 to x in G, then $\hat{L}(\alpha,x)=L(\alpha,x)$.*
4. *If there is no path from u to x and from D_2 to x in G, then $\hat{L}(\alpha,x)=L(\alpha,x)-\hat{\delta}_{O_2}$.*

For any of the five directly affected nodes $y\in\{u,O_1,O_2,D_1,D_2\}$, Algorithm 6 computes a lower bound on the longest path from α to y by exploiting the results in Theorem 3 and Corollary 2. In addition to the search cutoffs k and k', the procedure requires also the move $\mathcal{M}$ in G, the vector of longest path lengths from α and an upper bound on the decrease $\hat{\delta}_{O_2}$ in the longest path from α to O_2 (i.e., $\hat{\delta}_{O_2}\geq L(\alpha,O_2)-\tilde{L}(\alpha,O_2)$). Conditions in Lines 2 and 4 correspond to identities 1 and 2 in Corollary 2, respectively. If none of the previous two conditions is satisfied, the procedure computes the lower bounds on the longest path from α to each of the y predecessors using results 3 or 4 in Corollary 2. The lower bound on the longest path from α to y is computed in Lines 14–16 based on the lower bounds of each y predecessor. By calling Algorithm 4, the running time of Algorithm 6 is also $O(b^{\max\{k,k'\}})$. Note that, despite being only used on the nodes directly impacted by the move, Algorithm 6 is still valid for each node of the graph.

Algorithm 6 Procedure to compute a lower bound on longest path length from a reference node.

```
1:  procedure COMPUTELB($G$, $\alpha$, $y$, $k$, $k'$, $\mathcal{M}=\{u,O,D\}$, $\mathcal{L}_\alpha$, $\hat{\delta}_{O_2}$)  ▷
    This procedure assumes that the arcs in $\mathcal{D}$ are deleted from $A$
2:      if $y=u$ then
3:          return $\max_{(x,y)\in in(y)}\{L(\alpha,x)+l_x+l_{x,y}\}$
4:      if $y=O_1$ then
5:          return $L(\alpha,y)$
6:      for $x\in\mathcal{B}_y$ do
7:          $\hat{L}(\alpha,x)\leftarrow-\infty$
8:          $C\leftarrow$ ASSERTNOPATH($G,u,x,k,k',\mathcal{L}_\alpha$)
9:          if C $\wedge$ ASSERTNOPATH($G,O_2,x,k,k',\mathcal{L}_\alpha$) then
10:             $\hat{L}(\alpha,x)\leftarrow L(\alpha,x)$
11:             continue
12:         if C $\wedge$ ASSERTNOPATH($G,D_2,x,k,k',\mathcal{L}_\alpha$) then
13:             $\hat{L}(\alpha,x)\leftarrow L(\alpha,x)-\hat{\delta}_{O_2}$
14:     $\hat{L}(\alpha,y)\leftarrow-\infty$
15:     for $(x,y)\in(y)$ do
16:         $\hat{L}(\alpha,y)\leftarrow\max\{\hat{L}(\alpha,y),\hat{L}(\alpha,x)+l_x+l_{x,y}\}$
17:     return $\hat{L}(\alpha,y)$
```

The generic procedure in Algorithm 7 can be used to estimate a valid lower bound on any regular objective function resulting from applying move $\mathcal{M}$. This procedure computes a lower bound for each of the six subsets of paths identified above and illustrated in Fig. 6. Below is a short description of the procedure:

- The procedure starts by checking if there is no path from u to ω and from t to ω. Note that the assertion of the absence of a path only uses $\mathcal{L}_\alpha$. It is also possible to use $\mathcal{L}'_\omega$ instead or combine them. More importantly, the arcs in $\mathcal{D}$ are not deleted yet. Therefore, the absence of a path from u to ω also implies the absence of a path from t to ω. If the condition is satisfied, the move $\mathcal{M}$ does not affect the longest path from α to ω according to the first implication in Theorem 3.
- Lines 4–9 prepare the appropriate conditions to use Algorithm 6 by locally deleting the arcs in $\mathcal{D}$ from A and computing upper bounds on the decrease in the longest path length from α to t and from ω to s. The assignments in Line 6 result from identity 2 in Corollary 2. To compute a tighter upper bound on the decrease in the longest path length from α to t, it is necessary first to compute a lower bound on the longest path length from α to t in $\tilde{G}$. Such longest path might go through the new predecessor s, for which $\hat{L}(\alpha,s)$ is already computed in Line 6. Such longest path might also go through the unchanged predecessors $\mathcal{B}(t)$ after the deletion of $\mathcal{D}$ arcs. In this last case, Algorithm 6 is called. As there cannot be a path from t to any of its predecessors in $\mathcal{B}(t)$, each of the predecessor $x\in\mathcal{B}(t)$ can have either $L(\alpha,x)$ or $-\infty$ as a lower bound on its longest path length from α. In other words, $\hat{\delta}_t$ is not relevant when using

Algorithm 7 General procedure for quality evaluation.

```
1: procedure EVALUATEMOVEQUALITY($G$, $\mathcal{M} = \{u, 0 = (s, t), D = (v, w)\}$, $k$, $k'$, $\mathcal{L}_\alpha$, $\mathcal{L}'_\omega$)
2:     if AssertNoPath($G, u, \omega, k, k', \mathcal{L}_\alpha$)∧AssertNoPath($G, w, \omega, k, k', \mathcal{L}_\alpha$) then
3:         return $L(\alpha, \omega)$
4:     $A \leftarrow A \setminus \{(s, u), (u, t), (v, w)\}$        ▷ $A$ should be restored at the end of the proceudre
5:     $\hat{\delta}_t \leftarrow +\infty$, $\hat{\delta}'_s \leftarrow +\infty$
6:     $\hat{L}(\alpha, s) \leftarrow L(\alpha, s)$, $\hat{L}'(\omega, t) \leftarrow L'(\omega, t)$
7:     $\hat{L}(\alpha, t) \leftarrow \max\{\hat{L}(\alpha, s) + l_s + l_{s,t}$, COMPUTELB($G$, $\alpha$, $t$, $k$, $k'$, $\mathcal{M} = \{u, 0, D\}$, $\hat{\delta}_t$, $\mathcal{L}_\alpha$) }
8:     $\hat{L}'(\omega, s) \leftarrow \max\{\hat{L}'(\omega, t) + l_t + l'_{t,s}$, COMPUTELB($G'$, $\omega$, $s$, $k$, $k'$, $\mathcal{M}' = \{u, 0\prime, D'\}$, $\hat{\delta}'_s$, $\mathcal{L}'_\omega$) }
9:     $\hat{\delta}_t \leftarrow L(\alpha, t) - \hat{L}(\alpha, t)$, $\hat{\delta}'_s \leftarrow L'(\omega, s) - \hat{L}'(\omega, s)$
10:    for $y \in \{u, v, w\}$ do
11:        $\hat{L}(\alpha, y) \leftarrow$ COMPUTELB($G$, $\alpha$, $y$, $k$, $k'$, $\mathcal{M} = \{u, 0 = (s, t), D = (v, w)\}$, $\delta_t$, $\mathcal{L}_\alpha$)
12:        $\hat{L}'(\omega, y) \leftarrow$ COMPUTELB($G'$, $\omega$, $y$, $k$, $k'$, $\mathcal{M}' = \{u, 0\prime = (t, s), D' = (w, v)\}$, $\delta_s$, $\mathcal{L}_\omega$)
13:    $L_1 \leftarrow \hat{L}(\alpha, s) + l_s + l_{s,t} + l_t + \hat{L}'(\omega, t)$        ▷ Paths going through arc $(s, t)$
14:    $L_2 \leftarrow \hat{L}(\alpha, t) + l_t + \hat{L}'(\omega, t)$        ▷ Paths going through $t$
15:    $L_3 \leftarrow \hat{L}(\alpha, s) + l_s + \hat{L}'(\omega, s)$        ▷ Paths going through $s$
16:    $L_4 \leftarrow \hat{L}(\alpha, v) + l_v + \hat{L}'(\omega, v)$        ▷ Paths going through $v$ but not through arc $(v, u)$
17:    $L_5 \leftarrow \hat{L}(\alpha, w) + l_w + \hat{L}'(\omega, w)$        ▷ Paths going through $w$ but not through arc $(u, w)$
18:    $\hat{L}(\alpha, u) \leftarrow \max\{\hat{L}(\alpha, u), \hat{L}(\alpha, v) + l_v + l_{v,u}\}$
19:    $\hat{L}'(\omega, u) \leftarrow \max\{\hat{L}'(\omega, u), \hat{L}'(\omega, w) + l_w + l'_{w,u}\}$
20:    $L_6 \leftarrow \hat{L}(\alpha, u) + l_u + \hat{L}'(\omega, u)$        ▷ Paths going through $u$
21:    return $\max_{1 \le i \le 6}\{L_i\}$
```

Algorithm 6 for t. The same arguments apply for the computation of $\hat{L}'(\omega, s)$ as it is symmetrical to $\hat{L}(\alpha, t)$. The computation in Line 9 of the upper bounds on the decrease uses already computed $\hat{L}(\alpha, t)$ and $\hat{L}'(\omega, s)$.

- Lines 10–12 compute, for each of the remaining directly affected nodes $y \in \{u, v, w\}$, lower bounds on the longest path length from α to y in G and the longest path length from ω to y in G'.
- Lines 13–20 compute lower bounds on the longest path from α to ω for each subset of paths identified above and illustrated in Fig. 6. Recall that the subset of paths for which the length is estimated using L_1 is a subset of paths going through t or u. Therefore, it can easily be shown that $L_1 \le \min\{L_2, L_3\}$. However, the cost of considering such a redundant set is insignificant as it involves few arithmetic operations, while it is useful in Section 6.3. In general, depending on the specific problem to be solved, it should be possible to customize the procedure by not computing some of the lower bounds if they can always be shown to be lower than others.

The procedure returns the maximum among the six lower bounds calculated if the performed move can impact the longest path from α to ω. The computational cost of this procedure is determined by the values of k and k' used by the procedure in Algorithms 4 and 6. The running time of Algorithm 7 is also $O(b^{\max\{k,k'\}})$. Increasing the values of the two control parameters may increase the computational cost and the accuracy of the resulting lower bounds. Taking into account the properties on which the different results are built, this procedure can be used in Line 7 of Algorithm 1 to compute lower bounds on any regular objective function when solving any scheduling problem for which a solution can be modeled by a DAG with non-negative weights. When minimizing the makespan, it suffices to let ω represent the dummy end node to calculate valid lower bounds for this objective function. For a general regular objective function, the calculation of a lower bound can be carried out by calling the procedure for each sink node. In the same way as Mati et al. (2011), it is possible to improve the accuracy of the lower bounds by extending the set of paths considered to those passing through nodes not impacted by the move. Moreover, depending on the scheduling problem and the neighborhood structure, some steps of the proposed procedure may be unnecessary: The same subset of paths is considered more than once, or there is a lower bound L_i always lower than or equal to another L_j.

6.3. Relationship with results of the literature

This section aims to illustrate the generality of the results in Section 6.2. We show that a selection of well-known literature results can be reproduced by Algorithm 7 when appropriate values for the parameters are chosen. Note that, as all the selected literature results optimize the makespan, there is always a path from any node to the last dummy node ω, which makes the condition in line 2 of Algorithm 7 always false. Therefore, we mainly focus on showing that the proposed estimations functions in the literature correspond to one of the six estimations in Algorithm 7. Table 3 provides the parameter setting for each reference. The other papers in Section 5.2 not considered here are those using quality evaluations not satisfying the lower bound property. As in Table 2, the problem solved by each reference is denoted in Table 3 using the classical notation $\alpha|\beta|\gamma$ of Graham et al. (1979). For each reference, Column *Result* reports the original results to be shown as a special case of the procedure in Algorithm 7. Column *Lower Bound* specifies the lower bounds in Algorithm 7 that can be mapped to the lower bounds proposed in each reference. Finally, the last column provides the appropriate parameters (k and k' being the most relevant) to use if one aims at reproducing the lower bounds proposed in each reference. The interested reader can find in the Supplementary Material accompanying this article the proofs mapping the lower bounds of each reference to the six lower bounds computed by our procedure and how the settings reported in the last column can reproduce the corresponding literature result.

Below are a few remarks regarding the analysis of Table 3 and the referenced works:

- The diversity of the problems solved by the selected references illustrates the generality of the proposed procedure for move evaluation. Beyond the diversity in terms of constraints, the proposed procedure can also provide a valid

Table 3
Some literature results dealing with move quality evaluation.

Reference	Problem	Result	Lower bounds	EVALUATEMOVEQUALITY
Taillard (1994)	$J\|\|C_{\max}$	Section 2	$\max\{L_2, L_6\}$	$(G, \mathcal{M}, 0, 0, \mathcal{L}_\alpha, \mathcal{L}'_\omega)$
Dauzère-Pérès & Paulli (1997)	$FJ\|\|C_{\max}$	Theorem 1 Remark 1	$\max\{L_1, L_6\}$	$(G, \mathcal{M}, \|V\|, 0, \mathcal{L}_\alpha, \mathcal{L}'_\omega)$
Dauzère-Pérès et al. (1998)	$FMRJ\|bom\|C_{max}$	Theorem 1	L_6	$(G, \mathcal{M}, 1, 1, \mathcal{L}_\alpha, \mathcal{L}'_\omega)$
Shen et al. (2018)	$FJ\|s\|C_{\max}$	Proposition 4.3 Proposition 4.4	L_6	$(G, \mathcal{M}, \|V\|, 0, \mathcal{L}_\alpha, \mathcal{L}'_\omega)$
Kasapidis et al. (2021)	$FJ\|bom\|C_{\max}$	Theorem 2	$\max\{L_1, L_6\}$	$(G, \mathcal{M}, \|V\|, 0, \mathcal{L}_\alpha, \mathcal{L}'_\omega)$

lower bound on any regular criterion such as total flow time or maximum total weighted tardiness. To achieve this, it is sufficient to use Algorithm 7 to estimate the longest path from the dummy origin node α to each of the dummy sink nodes ω_i.

- The depth search cutoffs necessary to reproduce the literature results are either very small or very large. The notation $|V|$ represents the cardinality of the node set in G, and refers to situations where the absence of a path between two nodes is asserted with certainty. In Dauzère-Pérès & Paulli (1997), it is proposed to compute at each iteration of the local search algorithm for each node v the set of all nodes that belong to all the paths from the start dummy node α to v and the set of all nodes belonging to all the paths from v to the end dummy node ω. Computing the sets is expensive in terms of computational time and memory. However, the resulting cost is still low compared to the cost of evaluating the quality of each neighbor exactly. For example, Kasapidis et al. (2021), where the lower bound proposed by Dauzère-Pérès & Paulli (1997) is adapted to tackle the flexible job-shop scheduling problem with arbitrary precedence constraints, report that using the lower bounds leads to a speed-up factor of more than 100 in their experimental study. Shen et al. (2018) also report brief experimental results showing the significant positive effect of the use of the lower bound on the local search algorithm efficiency.
- Contrary to Shen et al. (2018) and Kasapidis et al. (2021), where the set of all predecessors and successors are used as in Dauzère-Pérès & Paulli (1997); Dauzère-Pérès et al. (1998) drop the computation of such sets as it has a significant negative impact on the efficiency of the local search algorithm when dealing with a flexible job-shop scheduling problem with multiple resources per operation and arbitrary precedence constraints. As shown in Table 3, it is enough to choose $k = k' = 1$ so that Algorithm 7 produces a lower bound that is at least as tight as the one in Dauzère-Pérès et al. (1998). The contrast between the lower bound in Dauzère-Pérès & Paulli (1997) and the lower bound in Dauzère-Pérès et al. (1998) shows the capability of the procedure in Algorithm 7 to adapt to the most appropriate trade-off between accuracy and efficiency.

7. Conclusions and perspectives

A framework was proposed in this paper that unifies and generalizes the contributions of many well-known papers of the last decades, which heuristically solve the job-shop and flexible job-shop scheduling problems. First, a parameterized procedure to evaluate the feasibility of a move was introduced. Besides encompassing multiple results from the literature, the procedure also innovates by explicitly allowing the management of the trade-off between the possibility of rejecting feasible moves and the computational time of the procedure. The second proposed parameterized procedure computes a valid lower bound on the length from a source node to a sink node after performing a move. This lower bound can be used to quickly evaluate the quality of a neighbor with any regular objective function. Again, by increasing the value of control parameters, the quality of the lower bound can be increased at the expense of the computational time of the procedure. The two parameterized procedures are shown to encompass different contributions from the literature (see Tables 2 and 3), thus ensuring that the numerical results presented for instance in Dauzère-Pérès & Paulli (1997); Mastrolilli & Gambardella (2000) and Shen et al. (2018) can be obtained with our framework.

Several directions for future research are discussed below. First, an ongoing work aims to experimentally investigate the applicability and generalization of the proposed framework on various classical and complex job-shop scheduling problems. Second, one way to make the framework even more general is to consider more complex moves which change more than one sequence of a single node or which involve more than one node (see e.g., Kis, 2003). Third, the efficiency of heuristic approaches can be further improved if bounded incremental algorithms for the evaluation of solutions, such as the one proposed in Katriel et al. (2005), can be generalized to handle complex moves.

Appendix A. Proofs

Proof of Lemma 1. As the condition cannot hold when $L(\alpha, u) = -\infty$, it can be used only when $L(\alpha, u) \geq 0$, i.e., there is a path from α to u. Let us assume there is a path from u to v, which implies the existence of at least one path from α to v going through u. Let $P_{\alpha,v} = (\alpha, w_1, w_2, \ldots, u, \ldots, v)$ be a path satisfying the condition $l(\alpha, w_1, w_2, \ldots, u) = L(\alpha, u)$. By definition, $l(P_{\alpha,v}) = l(\alpha, w_1, w_2, \ldots, u) + l_u + l(u, \ldots, v) = L(\alpha, u) + l_u + l(u, \ldots, v)$. As the path from u to v must contain one of the outgoing edges of u and given the assumption on non-negative weights, $l(u, \ldots, v) \geq l_{out(u)}$. Using respectively the definition of the longest path, and the above inequality, $L(\alpha, v) \geq l(P_{\alpha,v}) \geq L(\alpha, u) + l_u + l_{out(u)}$, which contradicts the condition. □

Proof of Corollary 1. By definition, G' is obtained by reversing the arcs of G. Therefore, deciding on the existence or not of a path from u to v in G is equivalent to deciding on the existence or not of a path from v to u in G'. As G' is a DAG with non-negative weights, Lemma 1 assert the absence of a path from v to u in G' if the condition $L'(\omega, u) < L'(\omega, v) + l_v + l_{out'(v)}$ is satisfied. Consequently, the satisfied condition asserts also the absence of path from u to v in G. □

Proof of Property 2. Let $d(u, v)$ denote the length of the shortest path from the BFS-tree root u to a node v. Let us assume there is a path $P(u, v)$ such that $P(u, v) \cap \mathcal{C}_u = \emptyset$ and $d(u, v) > k$. If such a path exists, then there is an arc $(x, y) \in P(u, v)$ such that $d(u, x) \leq k - 1$ and $d(u, y) \geq k + 1$. However, as BFS computes the shortest path from u to all its reachable nodes, the shortest path from u to y cannot be longer than the shortest path from u to x followed by the edge (x, y). In other words, $d(u, y) \leq (u, x) + 1$, which contradict the inequality $d(u, y) \geq k + 1$. □

Proof of Lemma 2. Let us assume there is a path from u to v in G. Given that $\mathcal{C}_u$ is a layer and using Property 2, if there is a path from u to v, then there is a path from $y \in \mathcal{C}_u$ to v. Property 1 ensures that $L(\alpha, v) \geq L(\alpha, y) + l_y + l_{out(y)}$. As $L(\alpha, y) + l_y + l_{out(y)} \geq \min_{x \in \mathcal{C}_u}(L(\alpha, x) + l_x + l_{out(x)})$, this contradicts the condition. □

Proof of Lemma 3. Property 1 ensures that $L(\alpha, u) + l_u + l_{out(u)} \leq \min_{x \in \mathcal{C}_u}(L(\alpha, x) + l_x + l_{out(x)})$. Therefore, $L(\alpha, v) < \min_{x \in \mathcal{C}_u}(L(\alpha, x) + l_x + l_{out(x)})$ is satisfied when $L(\alpha, v) < L(\alpha, u) + l_u + l_{out(u)}$ is true. □

Proof of Theorem 1. Let us assume there is a path from u to v. The layer property (Property 2) implies the existence of $x \in \mathcal{C}_u$ and $y \in \mathcal{C}'_v$ such that there is a path from x to y. Lemma 1 implies that, in case there is a path from x to y, $L(\alpha, y) \geq L(\alpha, x) + l_x + l_{out(x)}$, which contradicts the condition. □

Proof of Theorem 2. Let us assume that Algorithm 5 characterizes a move as feasible (*True* is returned) and the resulting directed graph $\tilde{G}$ includes a cycle. Two cases must be studied:

1. A cycle is created by a separate insertion of one arc, and
2. A cycle is created by the simultaneous insertion of two or more arcs in $\mathcal{I}$.

If there is a single inserted arc $(x, y) \in \mathcal{I}$ such that a cycle is created in the graph, then there is a path from y to x in G. This cannot happen as the path would be detected by Algorithm 4. Now, let us prove the case where the cycle is created by the simultaneous insertion of two or more arcs in $\mathcal{I}$. Inserting both (D_1, u) and (u, D_2) would create a cycle only if there is a path from D_2 to D_1. This contradicts the assumption of the absence of a cycle in G. □

Proof of Theorem 3. The general intuition behind the results of this theorem is that the longest path length between two nodes can only be changed if at least one path connecting the two nodes has been altered.

First implication: If there is no path from u, O_2 and D_2 to x in G, then none of the paths from α to x is altered. Therefore, $\tilde{L}(\alpha, x) = L(\alpha, x)$.

Second implication: If there is no path from u and O_2 to x in G, then the deletion of (O_1, u) and (u, O_2) and the insertion of (O_1, O_2) cannot affect any path from α to x. Therefore, a move $\mathcal{M}$ can have an impact on the longest path from α to x only after the inserting (D_1, u) and (u, D_2). Considering the triangle inequality assumption (i.e., $l_{D_1,D_2} \leq l_{D_1,u} + l_u + l_{u,D_2}$), the longest path length from α to x can only increase, i.e., $\tilde{L}(\alpha, x) \geq L(\alpha, x)$.

Third implication: If there is no path from u and D_2 to x in G, inserting the arcs (D_1, u) and (u, D_2) does not affect the longest path length from α to x. Therefore, considering the triangle inequality assumption, $\tilde{L}(\alpha, x) \leq L(\alpha, x)$. If a decrease in the longest path from α to x occurs then there is a path from O_2 to x. Note that, if there is a path from O_2 to x, a path cannot exist from u or D_2 to O_2 in G, otherwise this contradicts the assumption on the absence of a path from u to x in G. Therefore, $\delta_{O_2} = L(\alpha, O_2) - \tilde{L}(\alpha, O_2) \geq 0$.

Two situations can be considered here:

1. O_2 is not part of the longest path from α to x. In this case, $\tilde{L}(\alpha, x) = L(\alpha, x)$ as the longest path from α to x is not altered. Therefore, given that $\delta_{O_2} \geq 0$, the condition $\tilde{L}(\alpha, x) = L(\alpha, x) \geq L(\alpha, x) - \delta_{O_2}$ is satisfied.
2. O_2 is part of the longest path from α to x before the move, which implies that $L(\alpha, x) = L(\alpha, O_2) + l_{O_2} + L(O_2, x)$. As the longest path between O_2 to x cannot change, $\tilde{L}(O_2, x) = L(O_2, x) = L(\alpha, x) - L(\alpha, O_2) - l_{O_2}$ (1). After the move, O_2 may no longer be on the longest path, i.e., $\tilde{L}(\alpha, x) \geq \tilde{L}(\alpha, O_2) + l_{O_2} + \tilde{L}(O_2, x)$ (2). By replacing (1) in (2), $\tilde{L}(\alpha, x) \geq \tilde{L}(\alpha, O_2) + L(\alpha, x) - L(\alpha, O_2)$. Given that $\delta_{O_2} = L(\alpha, O_2) - \tilde{L}(\alpha, O_2)$, we get $\tilde{L}(\alpha, x) \geq L(\alpha, x) - \delta_{O_2}$. □

Proof of Corollary 2.. Each of the identity is restated and proved using the hypotheses and results in Theorem 3.

1. $\hat{L}(\alpha, x) = \tilde{L}(\alpha, x) = L(\alpha, x)$, for each $x \in \mathcal{B}(u)$: First, given that G is a DAG before deleting the arcs in $\mathcal{D}$, there could not be a path from u or O_2 to any of the predecessors of u, i.e., $x \in \mathcal{B}(u)$. Also, given that $\mathcal{M}$ is a feasible move, there could not be also a path from D_2 to any of u predecessors. Therefore, the identity is true according to the first implication in Theorem 3.
2. $\hat{L}(\alpha, 0_1) = L(\alpha, 0_1)$: Given that G is a DAG before deleting the arcs in $\mathcal{D}$, there could not be a path from u or O_2 to O_1. Therefore, $L(\alpha, 0_1)$ is a lower bound on $\tilde{L}(\alpha, 0_1)$ according to the second implication in Theorem 3, i.e., $\tilde{L}(\alpha, 0_1) = \hat{L}(\alpha, 0_1) = L(\alpha, 0_1)$.
3. If there is no path from u and O_2 to x in G, then $\hat{L}(\alpha, x) = L(\alpha, x)$: Straightforward result from the second implication in Theorem 3.
4. If there is no path from u and D_2 to x in G, then $\hat{L}(\alpha, x) = L(\alpha, x) - \hat{\delta}_{O_2}$: Straightforward result from the third implication in Theorem 3. □

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.ejor.2023.05.018.

References

Balas, E., & Vazacopoulos, A. (1998). Guided local search with shifting bottleneck for job shop scheduling. *Management Science, 44*(2), 262–275.

Bierwirth, C., & Kuhpfahl, J. (2017). Extended grasp for the job shop scheduling problem with total weighted tardiness objective. *European Journal of Operational Research, 261*(3), 835–848.

Błażewicz, J., Domschke, W., & Pesch, E. (1996). The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research, 93*(1), 1–33.

Bowman, E. H. (1959). The schedule-sequencing problem. *Operations Research, 7*(5), 621–624.

Braune, R., Zäpfel, G., & Affenzeller, M. (2013). Enhancing local search algorithms for job shops with min-sum objectives by approximate move evaluation. *Journal of Scheduling, 16*(5), 495–518.

Chaudhry, I. A., & Khan, A. A. (2016). A research survey: Review of flexible job shop scheduling techniques. *International Transactions in Operational Research, 23*(3), 551–591.

Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2008). *Algorithms*. McGraw-Hill Higher Education New York.

Dauzère-Pérès, S., & Paulli, J. (1997). An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. *Annals of Operations Research, 70*, 281–306.

Dauzère-Pérès, S., Roux, W., & Lasserre, J. (1998). Multi-resource shop scheduling with resource flexibility. *European Journal of Operational Research, 107*(2), 289–305.

Dell'Amico, M., & Trubian, M. (1993). Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research, 41*(3), 231–252.

García-León, A., Dauzère-Pérès, S., & Mati, Y. (2015). Minimizing regular criteria in the flexible job-shop scheduling problem. In *7th multidisciplinary international scheduling conference: Theory & applications, prague*.

Gonzalez, T., & Sahni, S. (1978). Flowshop and jobshop schedules: Complexity and approximation. *Operations Research, 26*(1), 36–52.

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Kan, A. R. (1979). Optimization and approximation in deterministic sequencing and scheduling: A survey. In *Annals of discrete mathematics: vol. 5* (pp. 287–326). Elsevier.

Kasapidis, G. A., Paraskevopoulos, D. C., Repoussis, P. P., & Tarantilis, C. D. (2021). Flexible job shop scheduling problems with arbitrary precedence graphs. *Production and Operations Management, 30*(11), 4044–4068.

Katriel, I., Michel, L., & Van Hentenryck, P. (2005). Maintaining longest paths incrementally. *Constraints, 10*(2), 159–183.

Kis, T. (2003). Job-shop scheduling with processing alternatives. *European Journal of Operational Research, 151*(2), 307–332.

Klemmt, A., Kutschke, J., & Schubert, C. (2017). From dispatching to scheduling: Challenges in integrating a generic optimization platform into semiconductor shop floor execution. In *2017 winter simulation conference (WSC)* (pp. 3691–3702). IEEE.

Knopp, S., Dauzère-Pérès, S., & Yugma, C. (2017). A batch-oblivious approach for complex job-shop scheduling problems. *European Journal of Operational Research, 263*(1), 50–61.
Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence, 27*(1), 97–109.
Lamorgese, L., & Mannino, C. (2019). A noncompact formulation for job-shop scheduling problems in traffic management. *Operations Research, 67*(6), 1586–1609.
Madraki, G., & Judd, R. P. (2021). Accelerating the calculation of makespan used in scheduling improvement heuristics. *Computers and Operations Research, 130*, 105233.
Mastrolilli, M., & Gambardella, L. M. (2000). Effective neighbourhood functions for the flexible job shop problem. *Journal of Scheduling, 3*(1), 3–20.
Mati, Y., Dauzère-Pérès, S., & Lahlou, C. (2011). A general approach for optimizing regular criteria in the job-shop scheduling problem. *European Journal of Operational Research, 212*(1), 33–42.
Nowicki, E., & Smutnicki, C. (1996). A fast taboo search algorithm for the job shop problem. *Management Science, 42*(6), 797–813.
Nowicki, E., & Smutnicki, C. (2005). An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling, 8*(2), 145–159.
Pham, D.-N., & Klinkert, A. (2008). Surgical case scheduling as a generalized job shop scheduling problem. *European Journal of Operational Research, 185*(3), 1011–1025.
Pinedo, M. (2016). *Scheduling: Theory, algorithms, and systems*. Basel: Springer International Publishing AG.
Shen, L., Dauzère-Pérès, S., & Neufeld, J. S. (2018). Solving the flexible job shop scheduling problem with sequence-dependent setup times. *European Journal of Operational Research, 265*(2), 503–516.
Sobeyko, O., & Mönch, L. (2016). Heuristic approaches for scheduling jobs in large-scale flexible job shops. *Computers and Operations Research, 68*, 97–109.
Taillard, E. D. (1994). Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing, 6*(2), 108–117.
Talbi, E.-G. (2009). *Metaheuristics: From design to implementation*: vol. 74. John Wiley & Sons.
Tamssaouet, K., Dauzère-Pérès, S., Knopp, S., Bitar, A., & Yugma, C. (2022). Multiobjective optimization for complex flexible job-shop scheduling problems. *European Journal of Operational Research, 296*(1), 87–100.
Van Laarhoven, P. J., Aarts, E. H., & Lenstra, J. K. (1992). Job shop scheduling by simulated annealing. *Operations Research, 40*(1), 113–125.
Zhang, C., Li, P., Guan, Z., & Rao, Y. (2007). A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem. *Computers and Operations Research, 34*(11), 3229–3242.
Zoghby, J., Barnes, J. W., & Hasenbein, J. J. (2005). Modeling the reentrant job shop scheduling problem with setups for metaheuristic searches. *European Journal of Operational Research, 167*(2), 336–348.