RAIRO-Oper. Res. 56 (2022) 2621–2649
https://doi.org/10.1051/ro/2022124

RAIRO Operations Research
www.rairo-ro.org

MINIMIZING THE TOTAL WEIGHTED EARLINESS AND TARDINESS FOR A SEQUENCE OF OPERATIONS IN JOB SHOPS

Baby Sivanandan Girish[1,*], Habibullah[1] and Jessamma Dileeplal[2]

Abstract. This paper proposes exact algorithms to generate optimal timing schedules for a given sequence of operations in job shops to minimize the total weighted earliness and tardiness. The algorithms are proposed for two job shop scheduling scenarios, one involving due dates only for the last operation of each job and the other involving due dates for all operations on all the jobs. Computational experiments on benchmark problem instances reveal that, in the case of the scheduling scenario involving due dates only for the last operation of each job, the proposed exact algorithms generate schedules faster than those generated using a popular optimization solver. In the case of the scheduling scenario involving due dates for all operations on all the jobs, the exact algorithms are competitive with the optimization solver in terms of computation time for small and medium size problems.

Mathematics Subject Classification. 90B35.

Received January 23, 2022. Accepted July 13, 2022.

1. Introduction

Job shop scheduling problem (JSP) is one of the important machine scheduling problems and is well known as NP-hard [21, 33]. The problem involves scheduling n jobs on a set of m machines. Each job has a chain of ordered operations to be performed on specific machines, and the processing order on the machines can be different for different jobs. The most commonly used scheduling objective in the literature on job shops is to minimize makespan [21]. Since the problem is NP-hard, the state-of-the-art solution methodologies mainly include heuristic and metaheuristic approaches. Giffler and Thompson (GT) algorithm is a well-known procedure to construct active schedules for a given priority order of operations in JSP with makespan, tardiness and flowtime-based objectives [17, 26]. The schedule generation mechanism in the GT algorithm also allows it to be used with dynamic priority dispatching rules, in which the job priorities change continuously over time during schedule generation [1, 8, 17].

The research on scheduling job shops to minimize total weighted earliness and tardiness (TWET) has gained considerable importance in recent years. The TWET minimization objective in JSP is important to manufacturing industries operating in a just-in-time (JIT) environment [22]. The aim is to reduce inventory costs and simultaneously satisfy customer demands with the timely delivery of products. The problem involves a due date

Keywords. Job shop scheduling, total weighted earliness-tardiness, optimal timing schedule, just-in-time manufacturing.

[1] Department of Aerospace Engineering, Indian Institute of Space Science and Technology, Thiruvananthapuram 695547, Kerala, India.

[2] Department of Mechanical Engineering, College of Engineering Perumon, Kollam 691601, Kerala, India.

*Corresponding author: girish@iist.ac.in

and weights for earliness and tardiness associated with each job (or its operations). Generating a schedule with a minimum TWET involves completing the jobs (or their operations) as close as possible to their respective due dates. Since the GT algorithm generates active schedules by left-aligning the operations to the earliest start time, it cannot generate optimal schedules for JSP with TWET minimization objective. This paper studies JSP with the minimization of TWET as the objective and proposes exact algorithms to generate optimal schedules for a given sequence of operations.

The rest of the paper is organized as follows. Section 2 presents the literature review, Section 3 presents the formulation of the problems, Section 4 presents the proposed optimal timing algorithms, Section 5 presents the computational study of the proposed algorithms and its comparison with the results obtained from a popular optimization solver, and Section 6 concludes with the scope for future work.

2. Literature review

The early research on TWET minimization objective in JSP considered the due date and earliness-tardiness weights only for the last operation of each job. Beck and Refalo [7] referred to this problem as the early/tardy scheduling problem (ETSP) and proposed a hybrid technique using constraint programming and linear programming to solve the problem. Danna *et al.* [13] adopted the mixed integer programming (MIP) formulation from [7] and proposed three strategies to solve the problem, namely, local branching, relaxation induced neighbourhood search, and guided dives. Kelbel and Hanzalek [23] presented a greedy search tree initialization procedure for solving the ETSP applied within a constraint programming framework. They used a slice-based search strategy available in a commercial optimization solver to explore the search tree generated by their procedure. Since the above approaches for ETSP use mathematical programming models to solve the problem, they inherently generate the optimal schedules and do not require a separate algorithm for schedule generation. Yang *et al.* [32] presented an enhanced genetic algorithm to solve ETSP with distinct due dates and a common deadline for all the jobs. They used an operation-based scheme to represent the chromosomes. They used a three-stage decoding procedure to decode each chromosome to a feasible schedule. Though their decoding procedure generates a feasible schedule, it is not proven to provide optimal schedules in all cases. To the best of our knowledge, there is no exact algorithm reported in the literature other than the mathematical programming-based approaches to generate optimal schedules for a given sequence of operations in ETSP.

Recently, the research on JSP with a due date and earliness-tardiness weights associated with each operation has gained importance. In this problem, the operations on all the jobs are scheduled to minimize the weighted sum of earliness and tardiness associated with the deviation of completion time of each operation from its respective due date. Baptiste *et al.* [5] were the first to introduce this problem and referred to it as the just-in-time job shop scheduling problem (JIT-JSP). They presented a mathematical programming formulation for the problem and found the lower bounds for 72 problem instances using two Lagrangian relaxations methods. They implemented simple heuristics to derive upper bounds using the Lagrangian relaxations and further improved them using a local search algorithm. Monette *et al.* [25] introduced a constraint programming approach for JIT-JSP that relies on a branch and bound procedure, a global filtering algorithm, and two search heuristics to solve the problem. Metaheuristic approaches have also been implemented to solve JIT-JSP. Araujo *et al.* [4] implemented a combination of genetic algorithm and local search procedure to solve JIT-JSP in two sequential phases. They generated the schedules for a given sequence of operations by left aligning the operations to their earliest start time. Dos Santos *et al.* [14] presented a hybrid method that combines an evolutionary algorithm, a mathematical programming model, and a local search procedure. The mathematical programming model is used to determine the optimal schedule for a given sequence of operations using a commercial optimization solver. Yang *et al.* [31] implemented an improved genetic algorithm that utilizes an operation-based scheme to represent the chromosomes. Each chromosome is decoded to generate the schedules using a three-stage decoding mechanism which initially generates a semi-active schedule and then improves the schedule by reducing earliness cost using greedy insertion mechanisms. Though their decoding procedure generates a feasible schedule, it is not proven to provide the optimal schedule for a given sequence of operations. Wang and Li [30] proposed a

combination of variable neighbourhood search and mathematical programming to solve JIT-JSP. They used the mathematical programming model to generate an optimal schedule for a given sequence of operations. Ahmadian and Salehipour [2] presented a matheuristic algorithm to solve JIP-JSP, which operates by decomposing the problem into smaller sub-problems and solving the subproblems using a commercial optimization solver to obtain optimal or near-optimal schedules. Ahmadian *et al.* [3] developed a variable neighbourhood search algorithm to solve JIT-JSP. They implemented four neighbourhood structures to generate improved solutions. They used a commercial optimization solver to generate and improve schedules in their algorithm.

The above literature review on JIT-JSP reveals that most researchers have developed metaheuristic algorithms that require a mathematical programming model to generate an optimal schedule for a given sequence of operations. To the best of our knowledge, there is no exact approach reported in the literature other than the mathematical programming approaches to generate optimal schedules. An exact algorithm for schedule generation will also be useful in developing and implementing priority dispatching rules in scheduling job shop with TWET objective. This paper proposes exact algorithms to generate optimal timing schedules for a given sequence of operations in JIT-JSP. The proposed optimal timing algorithms for JIT-JSP are extended to generate optimal schedules for a given sequence of operations in ETSP.

The proposed optimal timing (OT) algorithms for JSP is based on the OT algorithms presented in the literature for various other scheduling problems. The OT algorithms were initially introduced to generate optimal timing schedules for a given job sequence in single machine scheduling problem (SMSP) to minimize TWET. Garey *et al.* [16] presented an OT algorithm with $O(nlogn)$ time complexity for the SMSP with symmetric weights for earliness and tardiness. Szwarc and Mukhopadhyay [27] proposed an OT algorithm with $O(n^2)$ complexity for the SMSP with asymmetric earliness and tardiness weights. Lee and Choi [24] and Wan and Yen [29] also presented OT algorithms that were used to generate optimal timing schedules within their proposed metaheuristic algorithms for the SMSP. Chretienne [10] extended the OT algorithm proposed by Garey *et al.* [16] to the asymmetric and task-independent costs case in SMSP without increasing its worst-case complexity. He also proposed an $O(n^3logn)$ OT algorithm for the general case of asymmetric and task-dependent costs in SMSP. Bauman and Jozefowska [6] presented an $O(nlogn)$ OT algorithm for the SMSP. Hendel and Sourd [19] proposed an OT algorithm for the earliness-tardiness SMSP with a linear piece-wise cost function for each job. They showed that OT algorithms for SMSP can be extended to minimize TWET in the permutation flow shop scheduling problem with earliness-tardiness penalties for the last operation of each job. Feng and Lau [15] presented an OT algorithm for the SMSP and showed it to be more efficient than the OT algorithms presented in [16,27].

Besides SMSP, OT algorithms have also been implemented for other scheduling problems, like the resource-constrained project scheduling problem [28], parallel machine scheduling problem [9], PERT scheduling problem [11], and scheduling of aircraft landing problem [18].

The OT algorithms presented in the literature are similar in identifying and shifting the job clusters to minimize TWET. However, they differ in their implementation to handle the problem's specific constraints or make the algorithms run faster in practice. The proposed OT algorithms for JSP are similar in principle to the existing OT algorithms. They differ mainly in the mechanisms used for handling the specific constraints of the problem.

3. Problem formulation

3.1. Just-in-time job shop scheduling problem

The just-in-time job shop scheduling problem (JIT-JSP) can be described as follows [5]. There are a set of m machines, $M = \{M_1, M_2, \ldots, M_m\}$, and a set of n jobs, $J = \{J_1, J_2, \ldots, J_n\}$ to be processed. Let i is the index for jobs and k is the index for machines, *i.e.* $i = 1, 2, \ldots, n$ and $k = 1, 2, \ldots, m$. Each job J_i requires a set of n_i sequentially ordered operations, $O_i = \{O_{i1}, O_{i2}, \ldots, O_{in_i}\}$ to be performed. Let j is the index for operations, *i.e.* $j = 1, 2, \ldots, n_i$. Each operation O_{ij} is performed on a specified machine $M(O_{ij}) \in M$ and the processing time is given by p_{ij}. For each machine $M_k \in M$, $O(M_k)$ represents the set of all operations that are performed

on M_k. Each operation O_{ij} has a due date d_{ij} such that an early or late completion incurs a penalty which is proportional to the amount of deviation from d_{ij}. Each operation O_{ij} has two penalty coefficients, α_{ij} and β_{ij}, to penalize its early and tardy completion, respectively. If c_{ij} represents the scheduled completion time of operation O_{ij}, e_{ij} its earliness, and t_{ij} its tardiness, then $e_{ij} = \max(0, d_{ij} - c_{ij})$ and $t_{ij} = \max(0, c_{ij} - d_{ij})$. The objective of JIT-JSP is to determine an optimal schedule that minimizes the total cost due to deviation of completion of all the operations from their respective due dates, which is given by $\sum_{i=1}^{n}\sum_{j=1}^{m}(\alpha_{ij}e_{ij} + \beta_{ij}t_{ij})$. The mathematical formulation for the problem is as follows.

Objective:

$$\text{Minimize} \sum_{i=1}^{n}\sum_{j=1}^{m}(\alpha_{ij}e_{ij} + \beta_{ij}t_{ij}) \tag{3.1}$$

Subject to:

$$e_{ij} \geq d_{ij} - c_{ij} \quad \forall i, j \tag{3.2}$$
$$t_{ij} \geq c_{ij} - d_{ij} \quad \forall i, j \tag{3.3}$$
$$c_{i1} \geq p_{i1} \quad \forall i \tag{3.4}$$
$$c_{ij} \geq c_{ij-1} + p_{ij} \quad \forall i, j : j \neq 1 \tag{3.5}$$
$$c_{ij} \geq c_{i'j'} + p_{ij} \quad or \quad c_{i'j'} \geq c_{ij} + p_{i'j'} \quad \forall i, j, i', j', k : O_{ij} \in O(M_k), O_{i'j'} \in O(M_k), i \neq i' \tag{3.6}$$
$$e_{ij} \geq 0, t_{ij} \geq 0 \quad \forall i, j. \tag{3.7}$$

Constraints (3.2) and (3.3) relate the earliness and tardiness of each operation with its completion time and due date. Constraint (3.4) ensures that the first operation of each job starts after time 0. Constraint (3.5) imposes a precedence relationship between the two consecutive operations of the same job. Disjunctive constraint (3.6) ensures that two operations cannot be processed simultaneously if they belong to two different jobs and require processing on the same machine. For a given sequence of operations, the disjunctive constraint (3.6) transforms into a simple linear constraint, and the mathematical formulation transforms into a linear programming model.

3.2. Early/tardy job shop scheduling problem

The problem environment of the early/tardy job shop scheduling problem (ETSP) is the same as JIT-JSP, except that only the last operation of each job has a due date, and only its early or tardy completion is penalized. If d_i is the due date of the last operation of job J_i, e_i represents its earliness, and t_i represents its tardiness, then $e_i = \max(0, d_i - c_{in_i})$ and $t_i = \max(0, c_{in_i} - d_i)$, where c_{in_i} is the completion time of the last operation of job i. Each job J_i has two penalty coefficients, α_i and β_i, to penalize its early and tardy completion, respectively. The objective of ETSP is to determine the optimal schedule that minimizes the total cost due to the deviation of completion of the last operation of all the jobs from their respective due dates, which is given by $\sum_{i=1}^{n}(\alpha_i e_i + \beta_i t_i)$. The mathematical formulation for the problem is as follows.

Objective:

$$\text{Minimize} \sum_{i=1}^{n}(\alpha_i e_i + \beta_i t_i) \tag{3.8}$$

Subject to:

$$e_i \geq d_i - c_{in_i} \quad \forall i \tag{3.9}$$
$$t_i \geq c_{in_i} - d_i \quad \forall i \tag{3.10}$$
$$c_{i1} \geq p_{i1} \quad \forall i \tag{3.11}$$
$$c_{ij} \geq c_{ij-1} + p_{ij} \quad \forall i, j : j \neq 1 \tag{3.12}$$
$$c_{ij} \geq c_{i'j'} + p_{ij} \quad or \quad c_{i'j'} \geq c_{ij} + p_{i'j'} \quad \forall i, j, i'j', k : O_{ij} \in O(M_k), O_{i'j'} \in O(M_k), i \neq i' \tag{3.13}$$
$$e_i \geq 0, t_i \geq 0 \quad \forall i \tag{3.14}$$

Constraints (3.9) and (3.10) relate the earliness and tardiness of the last operation of each job with its completion time and due date. The constraints (3.11) to (3.13) are the same as the constraints (3.4) to (3.6) in JIT-JSP. For a given sequence of operations, the disjunctive constraint (3.13) transforms into a simple linear constraint, and the mathematical formulation transforms into a linear programming model.

4. The proposed optimal timing algorithms

This section first presents the implementation of the proposed OT algorithms for JIT-JSP. The extension of the OT algorithms to ETSP is presented subsequently.

4.1. The proposed OT algorithms for JIT-JSP

Let SEQ be the given sequence of N number of operations in JIT-JSP, where $N = \sum_{i=1}^{n} n_i$. Let each operation in SEQ is represented by a unique identifier $j (j = 1, 2, ..N)$ based on its position in the sequence. Therefore, each operation identifier j can be mapped to one of the operations in $O_i = \{O_{i1}, O_{i2}, \ldots, O_{in_i}\}, i = 1, 2, \ldots, n$ (see Sect. 3.1). Let the ordered set σ represent the set of operation identifiers in a sequence corresponding to the operations in SEQ. Let P_j be the processing time, D_j is the due date, and C_j is the completion time of the jth operation in σ. Let the early and tardy penalty coefficients be represented by g_j and h_j, respectively. Corresponding to the jth operation in σ, let the singleton sets $JP(j)$ and $JS(j)$ respectively contain their immediately preceding and succeeding operations on the same job. If the jth operation is the first operation of the job, then $JP(j) = \emptyset$ and if the jth operation is the last operation, then $JS(j) = \emptyset$. Let the singleton sets, $MP(j)$ and $MS(j)$, respectively, contain the immediately preceding and succeeding operations of the jth operation performed on the same machine. If jth operation is the first operation on the machine, then $MP(j) = \emptyset$, and if jth operation is the last operation, then $MS(j) = \emptyset$. Let $PR(j) = (JP(j) \cup MP(j))$ denote the set of preceding operations and $SU(j) = (JS(j) \cup MS(j))$ denote the set of succeeding operations on the same job or the same machine corresponding to jth operation in σ.

Let σ_i is the partial sequence that contains the first i operations in σ, and let $S_i = \{C_1, C_2, \ldots, C_i\}$ is the partial schedule corresponding to σ_i. Let f_i is the total weighted earliness and tardiness corresponding to the partial schedule S_i. In an optimum partial schedule S_i with minimum f_i, the operations will be aligned as close as possible to their respective due dates. This results in either the operations being scheduled at their due dates or forming clusters as shown in the Gantt chart in Figure 1. Each cluster consists of a set of contiguously scheduled operations called a block. A pair of operations (j, j'), where j precedes j' in σ_i, is said to be contiguous and belong to the same block if $C_{j'} = C_j + P_{j'}$, provided that (j, j') are either the two consecutive operations of the same job or the two consecutive operations on the same machine (*i.e.* $j \in PR(j')$). Let B_i be the block comprising of the cluster of operations which are contiguously scheduled and contains the ith operation in σ_i. In Figure 1, the block B_{12} is a maximum cardinality set formed by the cluster of operations {6,7,8,9,10,11,12} which are contiguously scheduled and contains the operation 12 (*i.e.* $O_{2,4}$). The operations in the set {1,2,3,4,5} are not contiguously scheduled with any of the operations in B_{12} and, therefore, are not included in the set B_{12}.

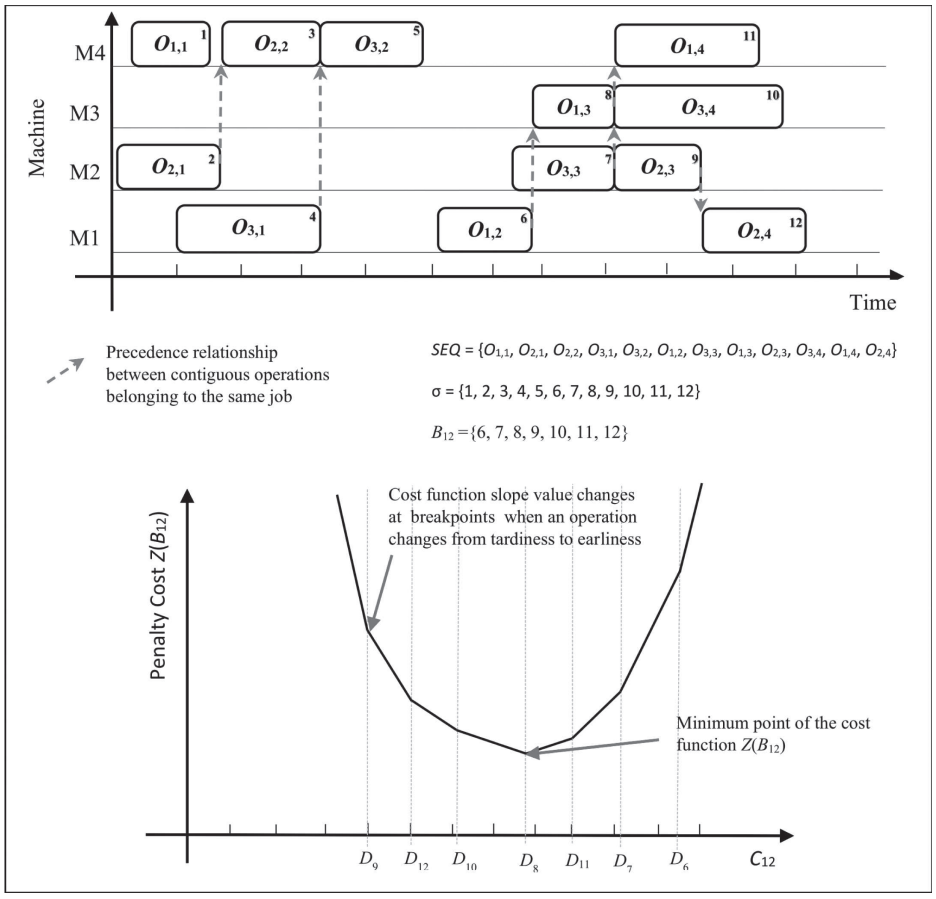

FIGURE 1. A typical cost function plot obtained by left shifting a set of operations in a block.

Let $f(B_i)$ is the total weighted earliness and tardiness function corresponding to the set of operations in B_i defined in terms of the completion time of the ith operation in σ_i. Then $f(B_i)$ will always be a piece-wise linear convex cost function when all the operations in B_i are shifted by the same amount of time. This can be explained using the following theorem.

Theorem 4.1. *The weighted sum of earliness and tardiness cost function $f(B_i)$ corresponding to a set of operations $B_i \subseteq \sigma_i$ with cardinality N' will always be a piece-wise linear convex function with at most N' breakpoints.*

Proof. The weighted sum of earliness and tardiness for the set of operations B_i with cardinality N' is given by

$$f(B_i) = \sum_{j \in B_i} (g_j \max(0, D_j - C_j) + h_j \max(0, C_j - D_j)) \tag{4.1}$$

The cost function (4.1) can be written in terms of the completion time C_i as

$$f(B_i) = \sum_{j \in B_i} (g_j \max(0, D_j - C_i + T_j) + h_j \max(0, C_i - T_j - D_j)) \tag{4.2}$$

where T_j is the time gap between the completion time of the operation i and operation j in B_i, *i.e.* $T_j = C_i - C_j$. The cost function (4.2) can be written as

$$f(B_i) = \sum_{j \in EY} g_j(D_j - C_i + T_j) + \sum_{j \in TY} h_j(C_i - T_j - D_j) \tag{4.3}$$

where EY represents the set of early operations and TY represents the set of tardy operations for a given value of C_i. The cost function (4.3) can be further rewritten as

$$f(B_i) = \left(\sum_{j \in TY} h_j - \sum_{j \in EY} g_j \right) C_i + \sum_{j \in EY} g_j(D_j + T_j) - \sum_{j \in TY} h_j(D_j + T_j) \tag{4.4}$$

The above cost function will be a straight line equation with slope $s(B_i) = \sum_{j \in TY} h_j - \sum_{j \in EY} g_j$ when all the operations are left-shifted by the same amount of time, *i.e.* T_j remains constant for all the operations in B_i. A typical plot of $f(B_i)$ *versus* C_i is shown in Figure 1. The 3 jobs-4 machines JIT-JSP instance shown in Figure 1 contains 7 operations in block B_{12}. The sets, EY and TY, change when an operation in B_i changes from tardiness to earliness at its due time while reducing C_i. This leads to a change in the slope of the cost function, resulting in a breakpoint in the plot as shown in Figure 1. Since there are N' number of operations in B_i, there can be a maximum of N' number of breakpoints, each occurring at the due date of one of the operations in B_i. As the operations in B_i change from tardiness to earliness while reducing C_i, the slope of the cost function monotonically decreases after each breakpoint. This is evident from the slope equation $s(B_i) = \sum_{j \in TY} h_j - \sum_{j \in EY} g_j$. The value of the cost function slope becomes negative after the breakpoint with the least value of $f(B_i)$. Therefore, left shifting the operations to the breakpoint where the cost function slope changes from a positive value to a non-positive value provides the optimal $f(B_i)$. This property forms the basis for optimizing the cost function in the OT algorithm. This property also holds for any subset of contiguously scheduled operations in B_i that can be left-shifted without violating the precedence constraints. □

The proposed OT algorithm for JIT-JSP can be described as follows. Initially, the first operation in σ is assigned its completion time as $C_1 = D_1$ and the partial schedule is generated as $S_1 = \{C_1\}$. In this case, f_1 corresponding to S_1 will be zero. Subsequently, the partial schedule $S_i (2 \leq i \leq N)$ is generated from S_{i-1} by assigning the completion times of all the operations from S_{i-1} to S_i. The completion time of the ith operation in S_i is determined as $C_i = \max(D_i, Y_i + P_i)$, where $Y_i = \max_{j \in PR(i)} C_j$. If $PR(i) = \emptyset$, then $Y_i = 0$.

If $C_i = D_i$, then the penalty cost of the ith operation in σ_i will be zero, and S_i will be optimal with $f_i = f_{i-1}$. On the other hand, if $C_i > D_i$, then the ith operation will have a penalty cost due to lateness and the partial schedule S_i needs to be optimized based on Theorem 4.1 discussed above. This involves the block B_i containing the ith operation in σ_i to be generated and a left shifting procedure, namely *LEFT_SHIFT*, is invoked to optimize the partial schedule S_i. Algorithm 1 shows the pseudocode of the proposed OT algorithm for JIT-JSP.

In OT algorithms applied to the single machine scheduling problem, all the jobs in a block corresponding to the last operation in the partial sequence are left-shifted to the minimum point of its cost function [24,29]. Since JIT-JSP involves multiple machines, multiple shiftable blocks can be generated from B_i, each containing the ith operation in σ_i. A shiftable block $R_x (R_x \subseteq B_i)$ comprises of a set of operations, such that if an operation j is included in R_x, then its immediately preceding contiguous operations in $PR(j)$ are also included in R_x. This allows the shiftable block R_x to be left-shifted by at least one unit of time without violating the precedence constraints of any of its operations with the respective immediately preceding operations. In other words, each shiftable block R_x is generated by eliminating a set of operations from B_i, such that the operations in the set R_x can be left-shifted by at least one unit of time. The shiftable blocks generated from B_{12} for the illustration problem shown in Figure 1 are $R_1 = \{12, 9, 7\}$, $R_2 = \{12, 9, 7, 10, 8, 6\}$ and $R_3 = \{12, 9, 7, 10, 8, 6, 11\}$, which are subsets of B_{12} and can be left shifted by at least one unit of time.

The shiftable block with the highest cost function positive slope value is chosen for left shifting among all the other shiftable blocks that can be formed from B_i. Let B_i^* denotes the block with the highest positive slope

Algorithm 1: OT algorithm for JIT-JSP

Data: $N, \sigma, D_j, P_j, g_j, h_j, PR(j), SU(j) \quad \forall j \in \sigma$

for $i = 1$ *to* N **do**
  if $i = 1$ **then**
    $C_1 \leftarrow D_1$
    $S_1 \leftarrow \{C_1\}$
  else
    if $PR(i) \neq \emptyset$ **then**
      $Y_i \leftarrow \max_{j \in PR(i)} C_j$
    else
      $Y_i \leftarrow 0$
    end
    $C_i \leftarrow \max(D_i, Y_i + P_i)$
    $S_i \leftarrow S_{i-1} \cup \{C_i\}$
    if $C_i > D_i$ **then**
      $LEFT\_SHIFT()$ ▷ Function call for the left shifting procedure
    end
  end
end
$f^* \leftarrow \sum_{j=1}^{N} (g_j \max(0, D_j - C_j) + h_j \max(0, C_j - D_j))$ ▷ optimal TWET

value among all the shiftable blocks in B_i. To optimize the partial schedule S_i, the block B_i^* is left-shifted towards the minimum point of its cost function until the nearest breakpoint is reached or an operation $j \in B_i^*$ becomes contiguous with an operation in $PR(j)$ that does not belong to B_i^*. In case if any of these two events occur, the block B_i and its corresponding shiftable blocks are regenerated using the improved partial schedule S_i and the block B_i^* with the highest positive cost function slope value ($s(B_i^*)$) is again chosen for further left shifting. This left shifting process continues until a shiftable block with positive cost function slope value cannot be created from B_i. The above procedure optimizes the partial schedule S_i and provides optimal f_i. This can be explained using the following Theorems 4.2 and 4.3.

Theorem 4.2. *Only the shiftable blocks, which are a subset of block B_i and contain the ith operation in σ_i, can have a positive cost function slope value.*

Proof. The cost function slope value of any shiftable block generated from B_i can be positive if and only if it contains the ith operation in σ_i. This can be explained by the fact that the left shifting procedure is implemented sequentially for the first $i-1$ operations in σ_i to find the optimal partial schedules $S_1, S_2, \ldots, S_{i-1}$. Therefore, considering Theorem 4.1, the left shifting of any shiftable block formed without the ith operation will have a negative cost function slope value and will lead to an increase in penalty cost. Similarly, any operation or a set of contiguously scheduled operations not belonging to B_i will have a negative cost function slope value. □

Theorem 4.3. *If $\{R_1, R_2, \ldots, R_p\}$ is the set of all the shiftable blocks that can be formed from block B_i with positive cost function slope value and containing the ith operation in σ_i, then left shifting the block with the highest slope value towards the nearest breakpoint or until its shiftable point without violating the precedence constraints, optimizes the partial schedule S_i.*

Proof. If there exists a shiftable block R_x with positive slope value $s(R_x)$ corresponding to its cost function $f(R_x)$, then based on Theorem 4.1, it can be concluded that left shifting the block R_x improves the penalty cost due to earliness and tardiness. The shiftable block with the highest positive cost function slope value provides the highest improvement in penalty cost per unit time and optimizes the partial schedule S_i. Though left shifting the other shiftable blocks with positive cost function slope value also improves the partial schedule, it may eventually result in sub-optimal schedules. This can be explained with the following example.

Let R^* ($R^* \subset B_i$) be the set of operations with the highest positive cost function slope value. Let R' ($R' \subset B_i$) be a set of operations that are not contained in R^* (*i.e.* R' and R^* are disjoint sets) and can be left shifted along with R^*. Since the operations in R' are not included in R^*, its cost function slope value $s(R')$ will be non-positive. Let us assume that $s(R') < 0$ and $s(R^*) + s(R') > 0$. Obviously, $s(R^*) > s(R^*) + s(R')$. Though, left shifting the operations in R' along with the operations in R^* will improve the total penalty cost (since $s(R^*) + s(R') > 0$), the rate of improvement of the penalty cost by left shifting the set $(R' \cup R^*)$ will be less compared to left shifting R^* alone, as $S(R') < 0$. This indicates that left shifting the operations in R' along with the operations in R^* results in a sub-optimal partial schedule. Therefore, selecting the block with the highest positive cost function slope value for left shifting optimizes partial schedule S_i. Regenerating B_i at each breakpoint or every time an operation becomes contiguous with a preceding operation, followed by left shifting the shiftable block with the highest positive cost function slope value, eventually optimizes f_i. □

Algorithm 2: Left shifting procedure to optimize the partial schedule

```
1  Function LEFT_SHIFT()
2      B_i^* ← OPT_BLOCK(i)                          ▷ Function call to find the optimal block
3      while B_i^* ≠ ∅ do
4          t_1 ← min_{j∈B_i^*}(C_j − P_j − C_{j'} : j' ∈ PR(j)  &  j' ∉ B_i^*)
5          t_2 ← min_{j∈B_i^*}(C_j − D_j : C_j > D_j)
6          t_3 ← min_{j∈B_i^*}(C_j − P_j)
7          C_j ← C_j − min(t_1, t_2, t_3)   ∀j ∈ B_i^*
8          B_i^* ← OPT_BLOCK(i)                      ▷ Function call to find the optimal block
9      end
10 end
```

Algorithm 2 shows the pseudocode of the left shifting procedure. The function OPT_BLOCK in the pseudocode generates the shiftable block with the highest positive cost function slope value, which is also referred to as the optimal block. We propose two methods to generate the optimal block. The first method is an enumeration procedure that generates all possible shiftable blocks with non-negative cost function slope values. Subsequently, the shiftable block with the highest cost function slope value is selected for left shifting. We consider even the optimal block with slope value equal to zero for left shifting as the objective value of the resulting schedule will remain the same. The second method is an improvement over the first method that uses dominance rules to ignore certain shiftable blocks in the process of finding the optimal block for left shifting.

4.1.1. Enumeration method

This method first generates a tree of sub-blocks in the forward pass. Subsequently, the sub-blocks with non-negative cost function slope values are recombined in the backward pass to form all possible shiftable blocks. An illustrative example of the procedure is shown in Figures 2 and 3.

The procedure starts with generating the sub-block b_1 by including the ith operation of σ_i as the first element in b_1. The preceding contiguous operations $j' \in PR(j)$ corresponding to each operation $j \in b_1$ are then iteratively included in b_1. Subsequently, the succeeding contiguous operations $j' \in SU(j) : C_{j'} > D_{j'}$ and $PR(j') = \emptyset$ corresponding to each operation $j \in b_1$, are iteratively included in b_1. Including the succeeding operations $j' \in SU(j) : C_{j'} > D_{j'}$ and $PR(j') = \emptyset$ in b_1, increases its cost function slope value. This procedure generates the shiftable sub-block $b_1 \subseteq B_i$. In Figure 2, the set b_1 is generated by first assigning the operation 24 to it. Subsequently, its preceding contiguous operation 20 is assigned to b_1. The operations 13 is then assigned to b_1 followed by operations 8 and 9, which are the preceding contiguous operations to the operation 13. The operation 4 is subsequently assigned to b_1 as it is preceding and contiguous to operation 9. There are no other operations in B_{24} which are preceding as well as contiguous to any of the assigned operations in b_1. The

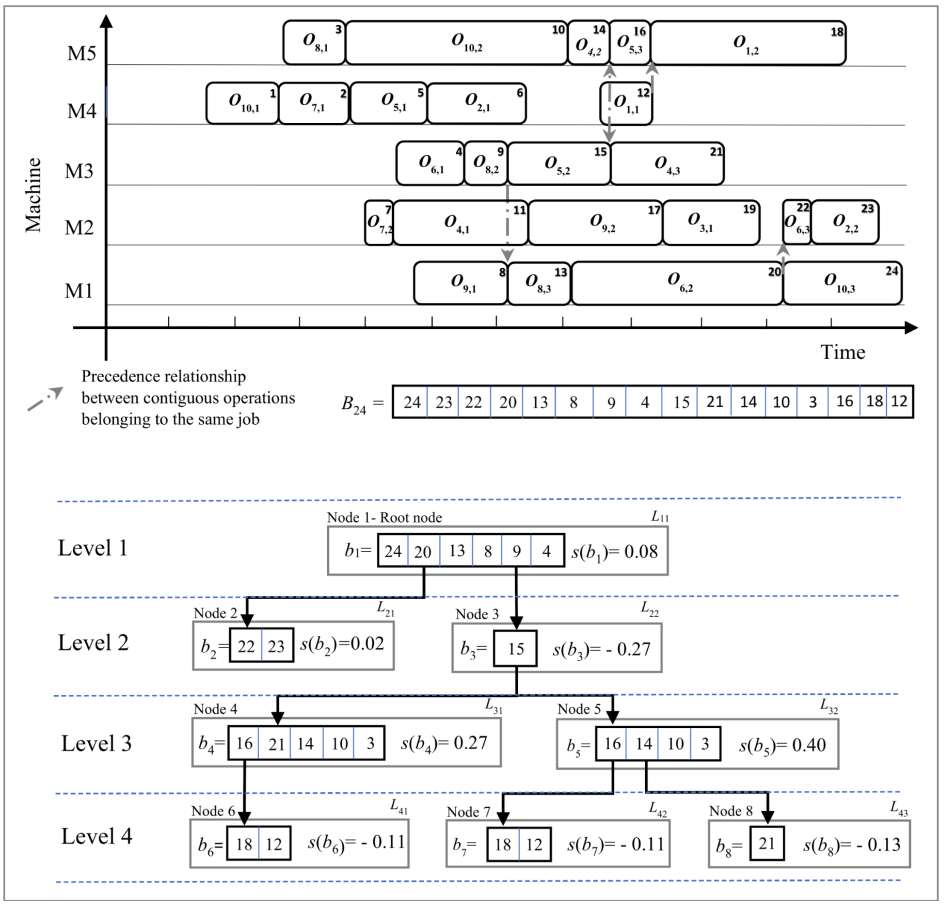

FIGURE 2. An illustrative example showing the generation of a tree of sub-blocks in the forward pass of the optimal block generation procedure

operations 22 and 15 are not assigned to b_1, though they are succeeding and contiguous to operations 20 and 9, respectively, as $C_{22} < D_{22}$ and $C_{15} < D_{15}$. There are no other operations in B_{24} which are succeeding as well as contiguous to any of the assigned operations in b_1. The sub-block b_1 generated by the above procedure can be left-shifted by at least one unit of time.

The succeeding contiguous operations $j' \in SU(j) : C_{j'} \leq D_{j'}$ or $PR(j') \neq \emptyset$ corresponding to each operation $j \in b_1$ are subsequently identified, and a sub-block is generated corresponding to each one of them using the abovementioned procedure. In Figure 2, the operation 15 is the succeeding contiguous operation on the same machine to the operation 9 in b_1. Since $C_{15} < D_{15}$, operation 15 was not included in b_1, and is assigned to the sub-block b_3. The operation 14 belonging to b_4 has a succeeding contiguous operation 16 on the same machine. Since operation 16 has no preceding contiguous operation other than the operations {21,14,10,3} already assigned in b_4 and $C_{16} > D_{16}$, it is assigned to the block b_4. However, operation 18 is not included in b_4 as it is contiguous with its preceding operation 12 on the same job. Therefore, it is assigned to b_6. In each newly formed sub-block, only the operations in B_i which were not allocated in the preceding sub-blocks are included. The succeeding contiguous operations corresponding to the operations included in the newly formed sub-blocks are further chosen to form new sub-blocks. This branching procedure is repeated until no more sub-blocks can be further generated. This branching procedure generates a tree of sub-blocks, as shown in Figure 2. Each

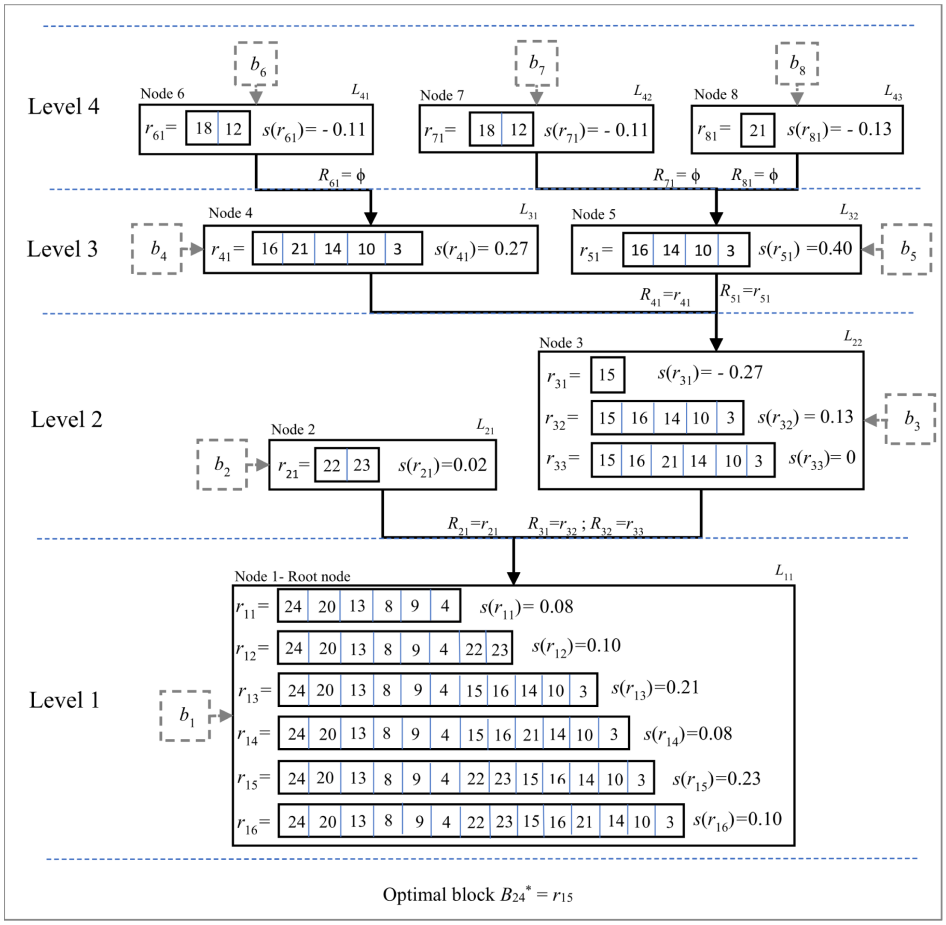

FIGURE 3. Generating all possible shiftable blocks in the backward pass for the illustration problem.

sub-block can be left-shifted only if its preceding sub-blocks in the tree are also left-shifted by the same amount of time. However, a sub-block has an option of not being left-shifted while its preceding sub-blocks in the tree are left-shifted.

Each sub-block in the branching procedure can be called a node, with block b_1 becoming the root node. Let each node is identified with a unique identifier k. Let A_k be the set of operations already assigned in the preceding nodes of node k. For *e.g.*, in Figure 2, $A_2 = \{24, 20, 13, 8, 9, 4\}$ and $A_4 = A_5 = \{24, 20, 13, 8, 9, 4, 15\}$, where the elements in the sets A_2, A_4 and A_5 are operation identifiers that belong to σ_i. Let T_k denote the set of all the sub-blocks generated in the forward pass originating from node k. For example, in Figure 2, $T_1 = \{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8\}$, $T_2 = \{b_2\}$ and $T_3 = \{b_3, b_4, b_5, b_6, b_7, b_8\}$. Let G_k denote the set of immediately succeeding nodes corresponding to a node k in the tree of sub-blocks. For *e.g.*, in Figure 2, $G_1 = \{2, 3\}$ and $G_3 = \{4, 5\}$, where the elements 2, 3, 4 and 5 are node identifiers. As shown in Figure 2, the nodes in the tree of sub-blocks can be categorized into levels such that the succeeding node corresponding to a node will be in the immediately succeeding level and its preceding node in the immediately preceding level. Let *levels* denote the total number of levels, and L_l denote the set of nodes at each level $l (l = 1, 2, \ldots, levels)$. For *e.g.*, in Figure 2,

$L_3 = \{4, 5\}$ and $L_4 = \{6, 7, 8\}$, where the elements 4, 5, 6, 7 and 8 are node identifiers. The nodes at each level are also denoted as L_{lk} where l is the index for level and k is the index for the nodes in the particular level.

In the backward pass shown in Figure 3, each node k generates a set of blocks $\{r_{k1}, r_{k2}, \ldots, r_{kq}\}$ by combining the sub-block b_k with the sets of blocks $\{R_{k'1}, R_{k'2}, \ldots, R_{k'p_{k'}}\}$ returned by its respective child nodes $k' \in G_k$. Only the set of blocks with non-negative cost function slope values in $\{r_{k1}, r_{k2}, \ldots, r_{kq}\}$ are returned to its respective parent node. If the cost function slope value of all the blocks in the set $\{r_{k1}, r_{k2}, \ldots, r_{kq}\}$ are negative, the node k returns a null set to the parent node. In Figure 3, the blocks R_{61}, R_{71}, and R_{81} are null sets since the corresponding sets r_{61}, r_{71}, and r_{81} have negative cost function slope values. This procedure ensures that only the blocks with non-negative cost function slope values are combined at each node.

The set of recombined blocks $\{r_{k1}, r_{k2}, \ldots, r_{kq}\}$ at each node k are generated by forming all possible combinations of the blocks returned by the child nodes, ensuring that the operations in a block are not repeated. In Figure 3, the recombined blocks $\{r_{11}, r_{12}, \ldots, r_{16}\}$ are generated by appending the sub-block b_1 with the combination of blocks R_{21} and $\{R_{31}, R_{32}\}$ returned by child nodes 2 and 3, respectively. The recombined block with the highest non-negative cost function slope value at the root node is assigned to B_i^*, which is left-shifted to optimize S_i. If all the recombined blocks generated at the root node have negative cost function slope values, the partial schedule S_i and the corresponding f_i are optimal.

Algorithm 3 shows the overall framework of the optimal block generation procedure in the form of pseudocode. The pseudocode shows two function calls, one to generate the tree of sub-blocks in the forward pass and the other to recombine the sub-blocks in the backward pass to find all possible shiftable blocks with non-negative cost function slope values. Subsequently, the optimal block with the highest non-negative cost function slope value is selected (lines 4 to 13 in the pseudocode) and returned to Algorithm 2. Algorithm 4 shows the pseudocode to generate the tree of sub-blocks in the forward pass using a recursive function. Algorithm 5 shows the pseudocode to recombine the sub-blocks in the backward pass. The pseudocodes use the same notations used in the illustrative example described above.

Algorithm 3: Pseudocode to generate the optimal block for left shifting

```
1  Function OPT_BLOCK(x)
2      FORWARD_PASS(x, 0, ∅)                          ▷ Function call to generate the tree of sub-blocks
3      node ← BACKWARD_PASS()   ▷ Function call to find the shiftable blocks with non-negative cost function slope
       values
4      z = 0
5      if node ≠ 0 then
6          max = 0
7          for y = 1 to p_1 do
8              if s(R_1y) > max then
9                  max = s(R_1y)
10                 z ← y
11             end
12         end
13     end
14     if z = 0 then
15         return ∅
16     else
17         return R_1z
18     end
19 end
```

Lines 7 to 27 in Algorithm 4 generate a sub-block corresponding to an operation x considering the already assigned operations in set A_k. The child nodes corresponding to the node are generated in lines 28 to 33.

Algorithm 4: Generating the tree of sub-blocks in the forward pass

```
1  Initialize global variables: k̂ = 0, levels = 0, H_j = 0   ∀j = 1, 2, ....i
2  Function FORWARD_PASS(x, l, Â)
3      l ← l + 1                                              ▷ l: level of the node in the tree
4      k̂ ← k̂ + 1                                              ▷ k: node identifier
5      Initialize: k ← k̂, e ← 1, b_k ← {x}, U_k ← ∅, G_k ← ∅, A_k ← ∅   ▷ b_k:ordered set
6      A_k ← Â ∪ {x}
7      while e ≤ |b_k| do
8          j ← b_k[e]                                         ▷ b_k[e]:e^th element in the ordered set b_k
9          for j' ∈ (PR(j) ∪ SU(j)) : j' ∉ A_k do
10             if j' ∈ PR(j) & C_j = C_j' + P_j then
11                 if C_j' = P_j' then
12                     return (0)                             ▷ return from function
13                 else
14                     A_k ← A_k ∪ {j'}
15                     b_k ← b_k ∪ {j'}
16                 end
17             else if j' ∈ SU(j) & C_j' = C_j + P_j' & C_j' > P_j' then
18                 if PR(j') = ∅ & C_j' > D_j' then
19                     A_k ← A_k ∪ {j'}
20                     b_k ← b_k ∪ {j'}
21                 else
22                     U_k ← j'
23                 end
24             end
25         end
26         e ← e + 1
27     end
28     for j' ∈ U_k : j' ∉ A_k do
29         node ← FORWARD_PASS(j', l, A_k)                    ▷ creates a child node
30         if node ≠ 0 then
31             G_k ← G_k ∪ {node}                             ▷ G_k: set of succeeding nodes of node k in the tree
32         end
33     end
34     if l > levels then
35         levels ← l                                         ▷ levels: number of levels in the tree
36         L_l ← ∅                                            ▷ L_l: set of nodes at the lth level in the tree
37     end
38     L_l ← L_l ∪ {k}
39     for j ∈ b_k do
40         H_j ← H_j + 1                                      ▷ H_j: number of times operation j appears in the tree
41     end
42     return (k)
43 end
```

The child node identifiers corresponding to the parent node k are updated in the set G_k. The level identifier corresponding to the node is updated in line 3 and the total number of levels are updated in lines 34 to 37. In line 38, the set L_l stores the node identifier k. Lines 39 to 41 update the number of times each operation $j \in b_k$ appears in the tree of sub-blocks T_1. The updated value for each operation j in H_j is used in the improved method discussed later.

Algorithm 5: Pseudocode to recombine the sub-blocks in the backward pass using the enumeration method

```
1  Function BACKWARD_PASS()
2      for l = levels to 1 do
3          for k ∈ L_l do
4              q = 1
5              r_kq ← b_k
6              for k' ∈ G_k do
7                  u ← q
8                  for y = 1 to p_k' do
9                      for x = 1 to q do
10                         if r_kx ∩ R_k'y = ∅ then
11                             u ← u + 1
12                             r_ku ← r_kx ∪ R_k'y
13                         end
14                     end
15                 end
16                 q ← u
17             end
18             p_k = 0
19             for x = 1 to q do
20                 if s(r_kx) ≥ 0 then
21                     p_k ← p_k + 1
22                     R_kp_k ← r_kx
23                 end
24             end
25         end
26     end
27 end
```

The pseudocode shown in Algorithm 5 accesses the levels in the tree of sub-blocks in the decreasing order and finds all possible combinations of blocks corresponding to each node. Line 5 in the pseudocode assigns b_k to set r_{k1}. Subsequently, the lines 6 to 17 generate all possible combinations of blocks $\{r_{k1}, r_{k2},\ldots, r_{kq}\}$ by accessing the sets of blocks $\{R_{k'1}, R_{k'2}, \ldots, R_{k'p'_k}\}$ from the child nodes $k' \in G_k$. The condition in line 10 ensures that each block generated does not have any repeating operations. Lines 18 to 24 select the set of recombined blocks $\{R_{k1}, R_{k2}, \ldots, R_{kp_k}\}$ with non-negative cost function slope values from the set $\{r_{k1}, r_{k2}, \ldots, r_{kq}\}$ to be returned to its parent node.

The total number of shiftable blocks generated from N number of operations can be theoretically considered as a problem of generating k-combinations of N elements for all values of k (*i.e.* $1 \leq k \leq N$), which is 2^N [12]. However, the actual number of shiftable blocks can be far less because if an operation j is not included in the shiftable block, then all the operations in the tree of sub-blocks T_k originating from node k will get excluded. Therefore, many combinations will not be feasible, and the proposed OT algorithm ensures that only feasible shiftable blocks are generated. The worst-case time complexity of the OT algorithm can be estimated as an exponential function of the problem size (N) due to the exponential increase in the number of shiftable blocks. We present the computational performance of the proposed OT algorithm on benchmark instances with up to 30 jobs and 20 machines in Section 5.

4.1.2. Improved method

Algorithms 6 and 7 show the pseudocodes to generate the optimal block using the improved method. The improved method uses dominance rules to ignore certain recombined blocks at each node in the backward pass, which will not result in the optimal block. This reduces the search space and improves the computation time

required to generate the optimal schedule. Therefore, this method is an implicit enumeration method. We use the following two dominance rules.

i. The first dominance rule, namely OPT_SELECT, checks if the operations present in the set of recombined blocks $\{r_{k1}, r_{k2}, \ldots, r_{kq}\}$ at a node $k \in L_l$ do not exist in the recombined blocks generated by the other nodes at the same level, *i.e.* $k' \in L_l : k' \neq k$. If the condition is satisfied, the recombined block with the highest non-negative cost function slope value is selected at node k and returned to its parent node. If the condition is not satisfied, then all the recombined blocks with non-negative cost function slope values are selected and returned to its parent node, as in the enumeration method.

ii. The second dominance rule, namely $OPT\_COMBINE$, is applied when the sub-block b_k at node k is combined with the blocks $(R_{k'1}, R_{k'2}, \ldots, R_{k'p_{k'}})$ returned by the child nodes $k' \in G_k$. The recombined blocks returned by the child nodes in the set G_k, that satisfied the condition in the OPT_SELECT rule, are directly appended to the sub-block b_k at node k. No new set of blocks is generated at the parent node using the blocks returned by such child nodes.

Lines 5 to 17 in the pseudocode shown in Algorithm 6 check whether the node k at level L_l satisfies the condition in the OPT_SELECT rule. Q_{kj} represents the number of times the operation j appears in the set of all the sub-blocks in the tree of sub-blocks T_k originating from node k. Lines 5 to 12 determine Q_{kj} corresponding to each operation j. The condition in line 14 of the pseudocode verifies that each operation j belonging to T_k exists only within it and does not exist in the other trees of sub-blocks at the same level. The value of I_k denotes whether the node k satisfies the condition in the OPT_SELECT rule. $I_k = 1$ indicates that the node k satisfies the condition in the OPT_SELECT rule, and $I_k = 0$ if not. Lines 4 to 14 in Algorithm 7 show the procedure to select the recombined block with the highest non-negative cost function slope value when the node k satisfies the condition in the OPT_SELECT rule (*i.e.* $I_k = 1$). Lines 16 to 21 show the procedure to select all the recombined blocks with non-negative cost function slope value when the node does not satisfy the condition in the OPT_SELECT rule (*i.e.* $I_k = 0$). Subsequently, in the immediate lower level L_{l-1}, the nodes that satisfied the condition in the OPT_SELECT rule in level L_l, are directly appended to the sub-block as shown in lines 18 to 22 in Algorithm 6. On the other hand, the child nodes that did not satisfy the condition, their returned blocks combine in all possible ways with the sub-block at the parent node, as shown in lines 25 to 38 of the pseudocode.

Figure 4 shows the optimal block generated using the improved method for the illustration problem shown in Figure 2. The recombined blocks generated at node 4 and node 5 at level 3 of the backward pass shown in Figure 4 have common operations. Therefore, they do not satisfy the condition in the $OPT\_COMBINE$ rule and the blocks R_{41}, R_{51} and b_3 are combined in all possible ways to form blocks r_{31}, r_{32} and r_{33} at node 3. The node 3 at level 2 satisfies the condition in the OPT_SELECT rule as none of the operations in the tree of sub-blocks T_3 exist in T_2. Therefore, only one block $R_{31} = r_{32}$ with highest non-negative cost function slope value ($s(r_{32}) = 0.13$) is selected among the set of recombined blocks $\{r_{32}, r_{33}\}$ with non-negative cost function slope values. Similarly, node 2 also satisfies the condition in the OPT_SELECT rule as none of the operations in the tree of sub-blocks T_2 exist in T_3. Consequently, at node 1, the blocks R_{21} and R_{31} are directly appended to the sub-block b_1 using the $OPT\_COMBINE$ rule to form the block r_{11}, which is also the optimal block for the given instance.

The following Theorems 4.4 and 4.5 prove the two dominance rules.

Theorem 4.4. *If a node $k \in L_l$ satisfies the condition that the operations in the tree of sub-blocks T_k do not exist in the other tree of sub-blocks $T_{k'}, k' \in L_l$, then the recombined block with the highest cost function slope value from the set $\{r_{k1}, r_{k2}, \ldots, r_{kq}\}$ dominates the remaining blocks.*

Proof. Suppose that there are two blocks, R_{u1} and R_{v1}, with a set of common operations that are returned to the parent node k from the child nodes u and v, respectively. Since the child nodes originate from the same parent node, the set of already assigned operations in the preceding nodes, u and v, will be the same, *i.e.* $A_u = A_v$. Therefore, the remaining operations, $(B_i - A_u)$ and $(B_i - A_v)$, available to form the respective tree of sub-blocks, T_u and T_v, in their forward pass will also be the same. Consequently, the resulting blocks R_{u1}

Algorithm 6: Pseudocode to combine sub-blocks in the backward pass using the improved method

```
1  Global variables: I_k = 1, Q_kj = 0,   ∀k = 1,2,...,k̂   ∀j = 1,2,...,i
2  Function BACKWARD_PASS()
3      for l = levels to 1 do
4          for k ∈ L_l do
5              for j ∈ b_k do
6                  Q_kj = 1
7              end
8              for k' ∈ G_k do
9                  for j = 1 to i do
10                     Q_kj ← Q_kj + Q_k'j
11                 end
12             end
13             for j = 1 to i do
14                 if Q_kj > 0 and Q_kj ≠ H_j then
15                     I_k = 0
16                 end
17             end
18             for k' ∈ G_k do
19                 if I_k' = 1 then
20                     b_k ← b_k ∪ R_k'p_k'        ▷ Node k' satisfies the condition in the OPT_COMBINE rule
21                 end
22             end
23             q = 1
24             r_kq ← b_k
25             for k' ∈ G_k do
26                 if I_k' = 0 then
27                     u ← q
28                     for y = 1 to p_k' do
29                         for x = 1 to q do
30                             if r_kx ∩ R_k'y = ∅ then
31                                 u ← u + 1
32                                 r_ku ← r_kx ∪ R_k'y
33                             end
34                         end
35                     end
36                     q ← u
37                 end
38             end
39             SELECT(k)        ▷ Function call to select the recombined blocks
40         end
41     end
42 end
```

and R_{v1} will be the subsets of the set $(B_i - A_u)$ (or $(B_i - A_v)$). As per the forward pass of the optimal block generation procedure shown in lines 28 to 32 of Algorithm 4, b_u and b_v are formed by two different succeeding operations corresponding to the set of operations in b_k. Since the sets R_{u1} and R_{v1} have common operations, the set of all the operations in T_u will be the same as in T_v. This is also evident from the illustrative example shown in Figure 2, where the set of operations {16,14,10,3,18,12,21} are the same in the trees of sub-blocks, $T_4 = \{b_4, b_6\}$ and $T_5 = \{b_5, b_7, b_8\}$. This is because a tree of sub-blocks is generated by identifying the preceding and the succeeding chain of contiguous operations. As a result, any two trees of sub-blocks originating from the

Algorithm 7: Pseudocode to select the set of recombined blocks with non-negative cost function slope value at each node

```
Function SELECT(k)
    p_k = 0
    if I_k = 1 then                                   ▷ Node k satifies the condition in the OPT_SELECT rule
        max = 0
        for x = 1 to q do
            if s(r_kx) ≥ max then
                max ← s(r_kx)
                p_k ← 1
                y ← x
            end
        end
        if p_k = 1 then
            R_kp_k ← r_ky
        end
    else
        for x = 1 to q do
            if s(r_kx) ≥ 0 then
                p_k ← p_k + 1
                R_kp_k ← r_kx
            end
        end
    end
end
```

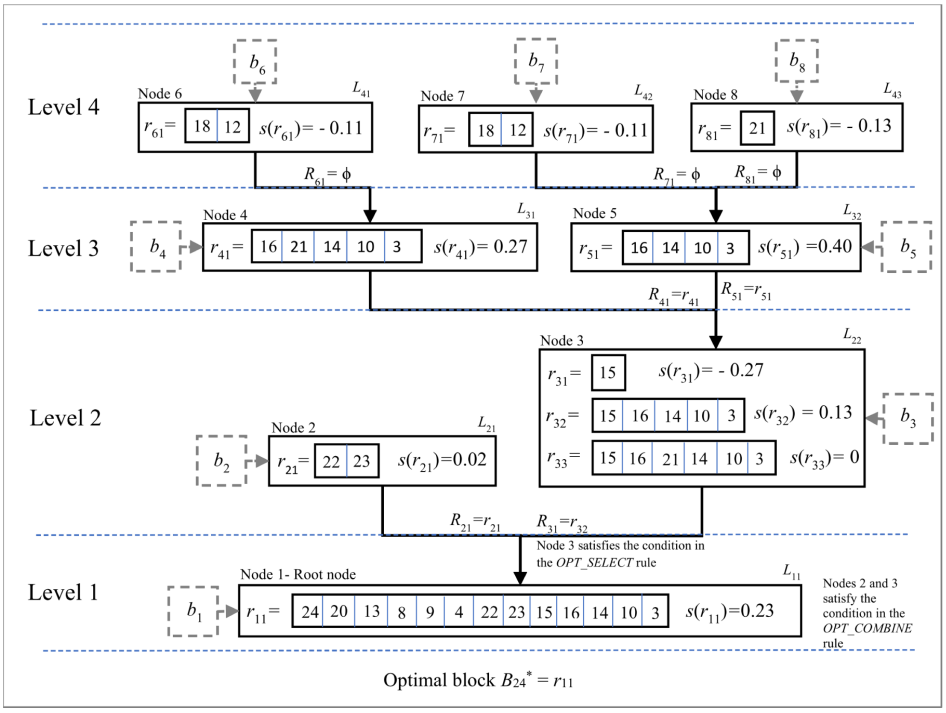

FIGURE 4. Generating optimal block using the improved method in the illustration problem.

same parent node will either have a different set of operations (*e.g.*, the tree of sub-blocks T_2 and T_3 in Fig. 2) or they will have the same set of operations (*e.g.*, the tree of sub-blocks T_4 and T_5 in Fig. 2). Nevertheless, the grouping of operations within the sub-blocks within a particular tree of sub-blocks can differ from the other trees of sub-blocks. This is evident from Figure 2, where the grouping of operations within the sub-blocks in $T_4 = \{b_4, b_6\}$ differ from the grouping of operations in $T_5 = \{b_5, b_7, b_8\}$. Consequently, the cost function slope values of the sub-blocks also differ from each other, as evident from Figure 2. Therefore, the optimal combination of sub-blocks belonging to different trees of sub-blocks originating from the same parent node eventually optimizes the total cost. Hence, all possible combinations of the recombined blocks are maintained until the trees of sub-blocks with a common set of operations converge at a parent node.

Suppose that the trees of sub-blocks with common operations converge at a node k. Then each recombined block generated at node k will be a combination of sub-blocks contained in T_k. The recombined block containing the optimum combination of sub-blocks will have the highest cost function slope value compared to all other recombined blocks generated at node k. Therefore, the recombined block with the highest cost function slope value will dominate all other recombined blocks generated at node k. □

Theorem 4.5. *The recombined block $R_{k'p_{k'}}$ with $p_{k'} = 1$, which satisfied the condition in the OPT_SELECT rule at child node $k' \in L_{l+1}$, can be appended to all the recombined blocks at its parent node $k \in L_l$ and does not require the generation of recombined blocks without $R_{k'p_{k'}}$ to find the optimal block.*

Proof. As discussed in the proof of Theorem 4.4, all possible combinations of the recombined blocks are maintained until the trees of sub-blocks with a common set of operations converge at a parent node. Therefore, the blocks returned by the child nodes having common operations with other nodes at the same level are combined in all possible ways to eventually obtain the optimum combination of operations in the shiftable block with the highest cost function slope value. Since the blocks that satisfied the condition in the OPT_SELECT rule are already the optimum combination of sub-blocks at their respective child nodes, they do not require to be recombined in all possible ways. They can be appended to all the recombined blocks generated at the parent node. □

4.2. The proposed OT algorithm for ETSP

The problem environment of ETSP is the same as JIT-JSP, except that the due dates and earliness-tardiness penalties are associated only with the last operation of each job. Let F ($F \subseteq \sigma$) represent the set containing the last operation of each job. We use the notations for earliness-tardiness penalties, g_j and h_j, and due date D_j for all the operations in σ. The earliness-tardiness penalties and due dates corresponding to the last operation (i.e. g_j, h_j, D_j, $j \in F$) are the inputs to the problem. The earliness-tardiness penalties and the due dates for the remaining operations are set as zero, *i.e.* $g_j = h_j = D_j = 0$, $j \in (\sigma - F)$. All the other notations used in the description of OT algorithm for ETSP are the same as those used in JIT-JSP.

Since in ETSP, only the last operation of each job has a due date, the first $n_i - 1$ operations of each job i are scheduled according to their position in the given sequence σ at their earliest start times. The last operation of each job i is scheduled as per the given sequence either at its due date or at its earliest possible start-time, whichever is later. The left shifting procedure is subsequently invoked to optimize its partial schedule. The left shifting procedure is applied only if at least one job with its last operation is not contiguously scheduled with any of its immediately preceding operations either on the same job or the same machine. Suppose that the last operations of the jobs in the partial schedule are contiguous with their respective preceding operations on the same job or the same machine. In that case, left shifting is not possible as all the operations, except the last operation of the jobs, are already scheduled at their earliest possible start times. Algorithm 8 shows the pseudocode of the OT algorithm for ETSP.

The OPT_BLOCK function call in Algorithm 8 is directed to Algorithm 3 presented in Section 4.1. The OPT_BLOCK function is invoked only if at least one job with its last operation in σ_i is not contiguously scheduled with any of its immediately preceding operations. Either the enumeration method (Algorithm 5) or

the improved method (Algorithms 6 and 7) can be implemented in the backward pass to generate the set of shiftable blocks. Since we considered $g_j = h_j = D_j = 0$, $j \in (\sigma - F)$ and the condition $s(R_{kj}) \geq 0, 1 \leq j \leq p_k$ to return the set $\{R_{k1}, R_{k2}, \ldots, R_{kp_k}\}$ to the parent node, the operations belonging to set $(\sigma - F)$ will remain left-aligned at their earliest start time during left shifting.

Algorithm 8: OT algorithm for ETSP

Data: $N, \sigma, P_j, PR(j), SU(j) \quad \forall j \in \sigma$
$F = \{$set of last operations of all the jobs: $F \subseteq \sigma\}, D_j, g_j, h_j \quad j \in F$
Initialize: $S_i = \emptyset \quad i = 0, 1, 2, \ldots, N, g_j = h_j = D_j = 0 \quad j \in (\sigma - F)$
for $i = 1$ *to* N **do**
  if $i \notin F$ **then**
    $C_i \leftarrow P_i + max_{j \in PR(i)} C_j$
    $S_i \leftarrow S_{i-1} \cup \{C_i\}$
  else
    if $PR(i) \neq \emptyset$ **then**
      $Y_i \leftarrow \max_{j \in PR(i)} C_j$
    else
      $Y_i \leftarrow 0$
    end
    $C_i \leftarrow \max(D_i, Y_i + P_i)$
    $S_i \leftarrow S_{i-1} \cup \{C_i\}$
    if $C_i > D_i$ *and at least one of the last operations of the jobs in* σ_i *is not contiguous with any preceding operation* **then**
      $B_i^* \leftarrow OPT\_BLOCK(i)$ ▷ Function call to find the optimal block
      while $B_i^* \neq \emptyset$ **do**
        $t_1 \leftarrow \min_{j \in B_i^*}(C_j - P_j - C_{j'} : j' \in PR(j) \ \ \& \ \ j' \notin B_i^*)$
        $t_2 \leftarrow \min_{j \in B_i^*}(C_j - D_j : C_j > D_j)$
        $t_3 \leftarrow \min_{j \in B_i^*}(C_j - P_j)$
        $C_j \leftarrow C_j - \min(t_1, t_2, t_3) \quad \forall j \in B_i^*$
        if *at least one of the last operations of the jobs in* σ_i *is not contiguous with any preceding operation* **then**
          $B_i^* \leftarrow OPT\_BLOCK(i)$ ▷ Function call to find the optimal block
        end
      end
    end
  end
end
$f^* \leftarrow \sum_{j \in F}(g_j \max(0, D_j - C_j) + h_j \max(0, C_j - D_j))$ ▷ optimal TWET

The total number of shiftable blocks generated from N number of operations can be theoretically considered as a problem of generating k-combinations of N elements for all values of k (*i.e.* $1 \leq k \leq N$), which is 2^N. This is the same as the computational complexity of generating the shiftable blocks in the case of JIT-JSP. However, only those operations in the set $(\sigma - F)$ are included in the shiftable blocks that are between the last operation of the job that appears first in σ and the last operation of the job that appears last in σ. Therefore, in a practical scenario, the computational complexity of the OT algorithms for ETSP will be much less than that of JIT-JSP. The computational performance of the proposed OT algorithms on ETSP instances with up to 50 jobs and 30 machines is presented in the subsequent section.

The operations belonging to the set $(\sigma - F)$ can also be scheduled using the Giffler and Thomspon (GT) algorithm to generate active schedules. However, if an operation $j \in (\sigma - F)$ is sequenced after operation $j' \in F$ in σ_i on same machine, j' should not be scheduled prior to j. The resulting optimal schedule will be either same

TABLE 1. Computational results for the JIT-JSP instances from literature with 2 machines.

Problem	Computation time (in seconds)					TF	
	CPLEX	OT1		OT2			
	AVG	AVG	MAX	AVG	MAX	OT1	OT2
I-10-2-tight-equal-1	6.6E-03	1.6E-04	1.6E-04	8.1E-05	8.1E-05	41.1	81.8
I-10-2-tight-equal-2	4.2E-03	5.6E-05	5.6E-05	5.0E-05	5.0E-05	74.5	83.5
I-10-2-tight-tard-1	8.3E-03	9.1E-05	9.1E-05	8.4E-05	8.4E-05	91.0	98.6
I-10-2-tight-tard-2	5.3E-03	9.6E-05	9.6E-05	8.8E-05	8.8E-05	55.1	60.2
I-10-2-loose-equal-1	6.0E-03	4.1E-04	5.1E-04	3.5E-04	3.5E-04	14.7	17.2
I-10-2-loose-equal-2	4.2E-03	1.4E-04	1.4E-04	1.3E-04	1.3E-04	29.5	33.3
I-10-2-loose-tard-1	4.0E-03	7.3E-05	7.3E-05	6.8E-05	6.8E-05	55.3	59.4
I-10-2-loose-tard-2	4.0E-03	8.4E-05	8.4E-05	8.0E-05	8.0E-05	47.9	50.3
I-15-2-tight-equal-1	5.6E-03	7.3E-05	7.3E-05	6.7E-05	6.7E-05	76.2	83.0
I-15-2-tight-equal-2	5.0E-03	5.9E-05	5.9E-05	5.3E-05	5.3E-05	85.3	94.9
I-15-2-tight-tard-1	5.3E-03	1.3E-04	1.3E-04	1.2E-04	1.2E-04	42.1	46.1
I-15-2-tight-tard-2	5.1E-03	8.4E-05	8.4E-05	7.8E-05	7.8E-05	60.9	65.6
I-15-2-loose-equal-1	9.0E-03	3.4E-04	3.4E-04	3.2E-04	3.2E-04	26.7	27.8
I-15-2-loose-equal-2	5.1E-03	2.6E-04	2.6E-04	2.7E-04	2.7E-04	19.2	18.9
I-15-2-loose-tard-1	7.8E-03	4.2E-04	4.2E-04	3.8E-04	3.8E-04	18.8	20.7
I-15-2-loose-tard-2	5.0E-03	3.3E-04	3.3E-04	2.6E-04	2.6E-04	15.0	18.9
I-20-2-tight-equal-1	7.6E-03	9.8E-05	9.8E-05	9.5E-05	9.5E-05	77.4	79.9
I-20-2-tight-equal-2	7.6E-03	2.3E-04	2.3E-04	2.0E-04	2.0E-04	33.3	38.9
I-20-2-tight-tard-1	7.7E-03	1.4E-04	1.4E-04	1.3E-04	1.3E-04	54.3	58.0
I-20-2-tight-tard-2	8.4E-03	2.8E-04	2.8E-04	2.4E-04	2.4E-04	30.5	35.4
I-20-2-loose-equal-1	1.7E-02	4.5E-04	4.5E-04	4.3E-04	4.3E-04	38.8	40.6
I-20-2-loose-equal-2	7.0E-03	1.6E-04	1.6E-04	1.5E-04	1.5E-04	44.0	47.9
I-20-2-loose-tard-1	7.5E-03	1.6E-04	1.6E-04	1.5E-04	1.5E-04	46.9	51.0
I-20-2-loose-tard-2	6.8E-03	1.2E-04	1.2E-04	1.1E-04	1.1E-04	58.1	60.7
Average	6.7E-03	1.8E-04	1.9E-04	1.7E-04	1.7E-04	47.4	53.0

or better than the optimal schedule generated strictly following the sequence of operations in σ. Since, the GT algorithm cannot be implemented in the optimization solver, we did not use the GT algorithm within the OT algorithms for effective comparison with the optimization solver in the computational study.

5. Computational results

The performance of the proposed OT algorithms for JIT-JSP and ETSP is evaluated using a set of benchmark instances from the literature [5]. The problem set consists of 72 instances. Each instance is named in the pattern $I-n-m-DD-W-ID$. Notations n and m, respectively, indicate the number of jobs and number of machines in the instance, where $n \in \{10, 15, 20\}$ and $m \in \{2, 5, 10\}$. Jobs are processed exactly once on each machine. However, the processing order of jobs on the machines varies. Processing times are in the range [10, 30]. DD represents the due date tightness and is either specified as *tight* or *loose*. W is specified either as *equal* or *tard*. *equal* indicates that the earliness and tardiness penalties are chosen randomly in the range [0.1, 1]. *tard* indicates that the tardiness penalty is chosen in the range [0.1, 1], whereas the earliness penalty is chosen in the range [0.1, 0.3]. There are two instances ($ID = 1$ and $ID = 2$) for each combination of the above parameters. Since the instances used in the literature for ETSP are not publicly available, we use the above 72 JIT-JSP instances and consider the due dates and earliness-tardiness penalties only for the last operation of each job.

The performance of the OT algorithms is compared with the results obtained by solving the linear programming (LP) formulation modeled using CPLEX solver [20]. The OT algorithms were coded in C language and run using Visual C++ on a PC with 3.6 GHz Intel Core i7-9700K octa-core processor, 16GB RAM, and

TABLE 2. Computational results for the JIT-JSP instances from literature with 5 machines.

Problem	Computation time (in seconds)					TF	
	CPLEX	OT1		OT2			
	AVG	AVG	MAX	AVG	MAX	OT1	OT2
I-10-5-tight-equal-1	8.5E-03	4.2E-04	4.2E-04	3.7E-04	3.7E-04	20.4	23.4
I-10-5-tight-equal-2	7.9E-03	1.6E-04	1.6E-04	1.5E-04	1.5E-04	48.1	52.2
I-10-5-tight-tard-1	8.8E-03	4.4E-04	4.4E-04	4.0E-04	4.0E-04	19.8	22.1
I-10-5-tight-tard-2	8.8E-03	2.0E-04	2.0E-04	1.8E-04	1.8E-04	43.4	48.1
I-10-5-loose-equal-1	1.1E-02	8.1E-04	8.1E-04	7.2E-04	7.2E-04	13.3	15.0
I-10-5-loose-equal-2	8.2E-03	4.2E-04	4.2E-04	3.7E-04	3.7E-04	19.3	22.2
I-10-5-loose-tard-1	8.1E-03	3.8E-04	3.8E-04	3.6E-04	3.6E-04	21.1	22.3
I-10-5-loose-tard-2	8.2E-03	1.5E-04	1.5E-04	1.4E-04	1.4E-04	53.8	58.0
I-15-5-tight-equal-1	1.6E-02	2.9E-04	7.6E-04	2.9E-04	6.5E-04	54.0	54.9
I-15-5-tight-equal-2	1.5E-02	2.4E-04	5.6E-04	2.3E-04	6.0E-04	64.5	66.1
I-15-5-tight-tard-1	1.5E-02	3.0E-04	7.7E-04	2.9E-04	6.5E-04	50.6	52.0
I-15-5-tight-tard-2	1.6E-02	8.4E-04	2.2E-03	7.4E-04	1.9E-03	18.6	20.9
I-15-5-loose-equal-1	1.6E-02	2.8E-04	1.8E-03	2.7E-04	4.1E-03	55.8	58.3
I-15-5-loose-equal-2	1.4E-02	3.0E-04	6.3E-04	2.9E-04	5.8E-04	48.5	50.7
I-15-5-loose-tard-1	1.4E-02	4.2E-04	9.4E-04	3.9E-04	8.8E-04	34.5	36.8
I-15-5-loose-tard-2	1.4E-02	1.5E-03	3.8E-03	1.3E-03	3.1E-03	9.3	11.1
I-20-5-tight-equal-1	2.5E-02	2.2E-03	6.3E-03	1.9E-03	5.3E-03	11.5	13.3
I-20-5-tight-equal-2	2.5E-02	2.2E-04	5.9E-04	2.2E-04	5.7E-04	112.8	114.4
I-20-5-tight-tard-1	2.5E-02	2.8E-04	6.8E-04	2.7E-04	6.5E-04	90.9	91.9
I-20-5-tight-tard-2	2.7E-02	2.2E-03	9.4E-03	1.9E-03	1.4E-02	11.9	14.3
I-20-5-loose-equal-1	2.5E-02	3.2E-04	1.6E-03	3.1E-04	1.3E-03	78.4	79.9
I-20-5-loose-equal-2	2.4E-02	3.2E-04	9.6E-04	3.2E-04	9.0E-04	75.1	76.3
I-20-5-loose-tard-1	2.5E-02	5.0E-04	2.5E-03	4.8E-04	1.5E-03	49.8	51.5
I-20-5-loose-tard-2	2.4E-02	2.5E-04	5.9E-04	2.5E-04	6.1E-04	95.1	95.5
Average	1.6E-02	5.6E-04	1.5E-03	5.0E-04	1.7E-03	45.9	48.0

Windows 10 operating system. The LP model was coded in C language using callable libraries from CPLEX concert technology and embedded within the OT algorithm code to compare the results.

To study the performance of the OT algorithms, a simple local search (LS) algorithm is used to generate sequences of operations corresponding to each problem instance. Algorithm 9 shows the pseudocode of the LS algorithm. In the LS algorithm, an initial solution is first generated by arranging the operations in the increasing order of its due date. The initial solution (X_{ini}) is set as the current solution (X_{cur}) and a set of neighbourhood solutions is generated corresponding to it. The best neighbourhood (with the least TWET value) replaces the current solution if it is an improved solution. Generating neighbourhoods and selecting the best neighbourhood to replace the current solution is repeated until there is no further improvement in the objective value. We used the pair-wise interchange mechanism for generating neighbourhoods. Two operations in the current solution (X_{cur}) are swapped if they do not violate the precedence relationships between the operations in the sequence. An operation is paired with another operation for swapping so that the distance between their positions in the sequence is within a specified limit. We set this limit as 50 for the computational study.

Since the proposed OT algorithms and CPLEX are exact methods, the objective values obtained with the approaches will always be the same. Therefore, the computational performance is evaluated based on the computation time. The average (AVG) and the maximum (MAX) computation time required to generate schedules using the OT algorithms for the sequences generated by the LS procedure are considered for comparison. CPLEX was found to generate schedules with a marginal variability in its computation time to solve different sequences generated using the LS algorithm. Therefore, only the average computation time (AVG) required by

TABLE 3. Computational results for the JIT-JSP instances from literature with 10 machines.

Problem	Computation time (in seconds)					TF	
	CPLEX	OT1		OT2			
	AVG	AVG	MAX	AVG	MAX	OT1	OT2
I-10-10-tight-equal-1	2.6E-02	5.9E-04	1.4E-03	5.6E-04	1.3E-03	44.2	46.0
I-10-10-tight-equal-2	2.2E-02	1.8E-03	3.2E-03	1.6E-03	2.9E-03	12.6	14.2
I-10-10-tight-tard-1	2.2E-02	2.2E-03	6.0E-03	1.9E-03	4.8E-03	10.3	11.6
I-10-10-tight-tard-2	2.2E-02	2.9E-03	1.2E-02	2.6E-03	6.2E-03	7.6	8.5
I-10-10-loose-equal-1	2.2E-02	7.6E-04	4.0E-03	7.2E-04	2.9E-03	28.6	30.1
I-10-10-loose-equal-2	2.1E-02	6.3E-04	1.7E-03	6.1E-04	1.6E-03	32.8	34.2
I-10-10-loose-tard-1	2.1E-02	1.5E-03	2.8E-02	1.3E-03	1.8E-02	13.9	16.3
I-10-10-loose-tard-2	2.0E-02	5.9E-04	1.3E-03	5.9E-04	1.1E-03	34.9	34.7
I-15-10-tight-equal-1	4.7E-02	3.8E-04	3.5E-03	3.7E-04	2.4E-03	124.4	125.7
I-15-10-tight-equal-2	4.8E-02	4.6E-03	3.9E-02	3.8E-03	1.8E-02	10.3	12.7
I-15-10-tight-tard-1	4.9E-02	2.8E-02	1.7E-01	2.2E-02	1.3E-01	1.7	2.2
I-15-10-tight-tard-2	4.9E-02	1.4E-02	2.1E-01	1.1E-02	1.7E-01	3.6	4.5
I-15-10-loose-equal-1	4.7E-02	4.6E-03	2.7E-02	3.7E-03	1.9E-02	10.1	12.6
I-15-10-loose-equal-2	4.5E-02	5.0E-03	2.7E-02	4.1E-03	2.1E-02	9.0	10.9
I-15-10-loose-tard-1	4.5E-02	2.5E-03	1.2E-02	2.3E-03	1.4E-02	18.1	19.5
I-15-10-loose-tard-2	4.5E-02	1.3E-03	5.1E-03	1.2E-03	3.5E-03	34.6	36.6
I-20-10-tight-equal-1	8.5E-02	8.1E-04	2.2E-03	8.1E-04	2.0E-03	105.1	105.6
I-20-10-tight-equal-2	8.7E-02	2.4E-02	1.5E-01	1.8E-02	1.1E-01	3.7	4.8
I-20-10-tight-tard-1	8.5E-02	3.0E-03	1.6E-02	2.6E-03	1.6E-02	28.4	32.6
I-20-10-tight-tard-2	8.7E-02	5.4E-02	1.6	4.1E-02	1.3	1.6	2.1
I-20-10-loose-equal-1	8.5E-02	5.6E-03	1.1E-01	4.6E-03	9.6E-02	15.0	18.3
I-20-10-loose-equal-2	8.0E-02	2.8E-03	1.8E-02	2.6E-03	1.7E-02	28.4	31.3
I-20-10-loose-tard-1	8.3E-02	7.4E-04	4.5E-03	7.4E-04	9.7E-03	111.7	112.0
I-20-10-loose-tard-2	8.2E-02	4.5E-02	1.1	3.4E-02	9.4E-01	1.8	2.4
Average	5.1E-02	8.6E-03	1.5E-01	6.8E-03	1.2E-01	28.9	30.4

Algorithm 9: Pseudocode of the local search algorithm

```
X_ini ← GenerateInitialSolution
X_cur ← X_ini
do
    f_best ← f(X_cur)
    X_best ← X_cur
    NH ← GenerateNeighbourhoods(X_cur)
    X_cur ← SelectBestNeighbour(NH)
while f(X_cur) < f_best
```

CPLEX to generate schedules has been considered for the performance comparison. We use the ratio between the average computation time required by CPLEX and the average computation time required by OT algorithm as a measure for performance comparison between the OT algorithms and CPLEX. We call this performance measure as times faster (TF) to indicate how many times the OT algorithm is faster than CPLEX. TF will be a useful measure to assess the performance improvement if OT algorithms are used instead of CPLEX for schedule generation. TF is determined as follows.

$$TF = \frac{\text{Average computation time (AVG) required by CPLEX}}{\text{Average computation time (AVG) required by the OT algorithm}}. \tag{5.1}$$

TABLE 4. Computational results for larger size JIT-JSP instances.

Problem	Computation time (in seconds)					TF	
	CPLEX	OT1		OT2			
	AVG	AVG	MAX	AVG	MAX	OT1	OT2
N-25-10-tight-equal	1.5E-01	1.3E-03	7.4E-03	1.2E-03	5.4E-03	114.0	132.0
N-25-10-tight-tard	1.5E-01	8.9E-04	2.8E-03	8.0E-04	2.5E-03	165.7	183.7
N-25-10-loose-equal	1.5E-01	1.2E-03	5.2E-03	1.0E-03	3.5E-03	120.4	141.2
N-25-10-loose-tard	1.4E-01	7.0E-04	6.4E-03	6.5E-04	3.1E-03	204.4	221.0
N-25-15-tight-equal	2.8E-01	2.7E-02	6.2	2.7E-02	6.1	10.2	10.3
N-25-15-tight-tard	2.8E-01	4.0E-03	2.1E-02	3.3E-03	1.8E-02	69.6	82.5
N-25-15-loose-equal	2.8E-01	9.5E-02	4.8	9.1E-02	4.7	2.9	3.1
N-25-15-loose-tard	2.8E-01	1.2E-01	309.6	1.1E-01	309.4	2.3	2.6
N-25-20-tight-equal	4.7E-01	8.6E-02	3.3	8.4E-02	3.3	5.5	5.6
N-25-20-tight-tard	4.7E-01	2.0	190.5	1.8	184.5	**0.2**	**0.3**
N-25-20-loose-equal	4.7E-01	8.0E-01	198.5	7.7E-01	197.7	**0.6**	**0.6**
N-25-20-loose-tard	4.6E-01	5.0	220.2	4.9	207.1	**0.1**	**0.1**
N-30-10-tight-equal	1.9E-01	1.8E-03	6.9E-03	1.6E-03	6.8E-03	107.7	123.2
N-30-10-tight-tard	1.9E-01	1.1E-03	2.4E-03	9.6E-04	2.2E-03	183.2	201.5
N-30-10-loose-equal	1.9E-01	1.7E-03	3.2E-03	1.4E-03	2.9E-03	117.2	139.5
N-30-10-loose-tard	1.9E-01	3.7E-03	2.4E-03	1.2E-03	1.0E-03	52.3	164.3
N-30-15-tight-equal	3.9E-01	4.7E-02	9.9E-01	4.7E-02	9.7E-01	8.2	8.3
N-30-15-tight-tard	4.0E-01	6.4E-03	3.5E-02	5.1E-03	3.5E-02	62.9	78.7
N-30-15-loose-equal	3.9E-01	1.7E-01	11.0	1.7E-01	10.6	2.3	2.3
N-30-15-loose-tard	3.9E-01	1.7E-02	3.8	1.5E-02	3.8	22.7	25.6
N-30-20-tight-equal	6.8E-01	5.5E-01	35.0	5.0E-01	16.0	1.2	1.3
N-30-20-tight-tard	6.7E-01	5.2E-01	80.0	5.1E-01	79.7	1.3	1.3
N-30-20-loose-equal	6.6E-01	5.3	604.5	3.6	506.5	**0.1**	**0.2**
N-30-20-loose-tard	6.6E-01	1.9	162.0	1.9	80.3	**0.3**	**0.4**

The following sections present the performance comparison between the OT algorithms and CPLEX on JIT-JSP and ETSP instances.

5.1. Performance comparison on JIT-JSP instances

Tables 1, 2 and 3 show the results obtained with the OT algorithms and CPLEX for the JIT-JSP instances from literature with 2, 5 and 10 machines, respectively. OT1 represents the enumeration method, and OT2 represents the improved method. The comparison between the average TF values of OT1 and OT2 with CPLEX reveals that the OT algorithms are approximately 30 to 50 times faster than CPLEX in generating the schedules. With the increase in the number of machines, the average TF values of the OT algorithms decreases. The comparison between the average TF values of OT1 and OT2 reveals that the performance of the OT2 algorithm is slightly better than that of the OT1 algorithm. The comparison between the AVG values of the OT algorithms with that of CPLEX reveals that the OT algorithms consistently perform better than CPLEX. The comparison of MAX values of the OT algorithms with the corresponding AVG values of CPLEX reveals that, for problem instances involving 2 and 5 machines, the OT algorithms consistently perform better than CPLEX. However, for a few instances with 10 machines shown in Table 3, the MAX values of the OT algorithms are inferior compared to the corresponding AVG values of CPLEX. This indicates that for a few larger size instances (*i.e.* instances with 15 and 20 jobs) with 10 machines, the OT algorithms required higher computation time than CPLEX to generate schedules for some of the sequences generated with the LS algorithm. However, the AVG values of the OT algorithms for those instances are marginally better than that of CPLEX. Hence, it can be concluded that the proposed OT algorithms are competitive with CPLEX in terms of computation time for small and medium

TABLE 5. Computational results for the ETSP instances from literature with 2 machines.

Problem	Computation time (in seconds)					TF	
	CPLEX	OT1		OT2			
	AVG	AVG	MAX	AVG	MAX	OT1	OT2
I-10-2-tight-equal-1	7.6E-03	3.2E-05	3.2E-05	3.1E-05	3.1E-05	237.4	245.1
I-10-2-tight-equal-2	3.5E-03	2.1E-05	2.1E-05	1.9E-05	1.9E-05	168.5	186.2
I-10-2-tight-tard-1	3.6E-03	3.4E-05	3.4E-05	3.3E-05	3.3E-05	105.5	108.7
I-10-2-tight-tard-2	3.6E-03	4.6E-05	4.6E-05	4.1E-05	4.1E-05	78.3	87.9
I-10-2-loose-equal-1	3.6E-03	1.5E-05	1.5E-05	1.3E-05	1.3E-05	237.5	274.0
I-10-2-loose-equal-2	3.6E-03	4.8E-05	4.8E-05	4.4E-05	4.4E-05	75.8	82.7
I-10-2-loose-tard-1	3.6E-03	2.9E-05	2.9E-05	2.7E-05	2.7E-05	124.7	133.9
I-10-2-loose-tard-2	3.6E-03	7.1E-05	7.1E-05	3.6E-05	3.6E-05	50.3	99.2
I-15-2-tight-equal-1	5.5E-03	2.4E-05	2.4E-05	2.1E-05	2.1E-05	230.6	263.5
I-15-2-tight-equal-2	5.1E-03	3.5E-05	3.5E-05	3.0E-05	3.0E-05	145.6	169.8
I-15-2-tight-tard-1	5.2E-03	4.4E-05	4.4E-05	4.0E-05	4.0E-05	117.1	128.8
I-15-2-tight-tard-2	4.6E-03	2.3E-05	2.3E-05	2.0E-05	2.0E-05	198.0	227.7
I-15-2-loose-equal-1	5.2E-03	2.4E-05	2.4E-05	2.2E-05	2.2E-05	215.0	234.5
I-15-2-loose-equal-2	5.1E-03	5.1E-05	5.1E-05	4.7E-05	4.7E-05	100.4	109.0
I-15-2-loose-tard-1	5.3E-03	3.6E-05	3.6E-05	3.3E-05	3.3E-05	146.1	159.3
I-15-2-loose-tard-2	5.0E-03	7.5E-05	7.5E-05	6.9E-05	6.9E-05	67.0	72.8
I-20-2-tight-equal-1	6.3E-03	3.1E-05	3.1E-05	2.8E-05	2.8E-05	202.4	224.1
I-20-2-tight-equal-2	7.1E-03	5.3E-05	5.3E-05	4.7E-05	4.7E-05	134.6	151.7
I-20-2-tight-tard-1	7.3E-03	5.8E-05	5.8E-05	5.4E-05	5.4E-05	126.4	135.7
I-20-2-tight-tard-2	7.1E-03	6.5E-05	6.5E-05	5.7E-05	5.7E-05	108.6	123.9
I-20-2-loose-equal-1	6.3E-03	3.1E-05	3.1E-05	2.9E-05	2.9E-05	203.6	217.7
I-20-2-loose-equal-2	7.1E-03	4.9E-05	4.9E-05	4.5E-05	4.5E-05	144.5	157.4
I-20-2-loose-tard-1	6.6E-03	5.0E-05	5.0E-05	4.7E-05	4.7E-05	131.0	139.4
I-20-2-loose-tard-2	6.7E-03	5.2E-05	5.2E-05	4.8E-05	4.8E-05	129.1	139.8
Average	5.3E-03	4.2E-05	4.2E-05	3.7E-05	3.7E-05	144.9	161.4

size problems. For larger size problems with 10 machines, the proposed OT algorithms are not consistent in their performance. They require higher computation time than CPLEX to generate schedules for some of the sequences generated by the LS algorithm.

The above results obtained with the benchmark instances from literature reveal that, with the increase in problem size, the performance of the OT algorithms deteriorates compared to CPLEX. To analyze the performance bounds beyond which the CPLEX would get closer or perform better than the OT algorithms in terms of the average computational time (AVG), we generated larger size instances with up to 30 jobs and 20 machines. The problem instances were generated based on the procedure used in the literature [5], described in Section 5 of this paper. The newly generated instances are named in the pattern $N-n-m-DD-W$, where the notations n, m, DD and W are the same as described previously for the instances from the literature, shown in Tables 1, 2 and 3. Table 4 shows the results obtained with OT algorithms and CPLEX for the newly generated larger size instances. The results reveal that, for some of the instances involving 20 machines, the TF values are less than 1, which are highlighted in bold in Table 4. This shows that, as the problem size increases to 25 jobs and 20 machines, CPLEX performs relatively better than the OT algorithms. The MAX values obtained with the OT algorithms for instances involving 20 machines are also much higher, as shown in Table 4. The reasons can be attributed to the exponential complexity of the OT algorithms. This limits their application to small and medium-sized JIT-JSP instances, particularly while implementing within heuristic and metaheuristic algorithms.

TABLE 6. Computational results for the ETSP instances from literature with 5 machines.

Problem	Computation time (in seconds)					TF	
	CPLEX	OT1		OT2			
	AVG	AVG	MAX	AVG	MAX	OT1	OT2
I-10-5-tight-equal-1	9.2E-03	1.8E-05	1.8E-05	1.5E-05	1.5E-05	513.1	615.7
I-10-5-tight-equal-2	7.8E-03	1.8E-05	1.8E-05	1.5E-05	1.5E-05	432.3	518.8
I-10-5-tight-tard-1	8.5E-03	2.4E-05	2.4E-05	2.1E-05	2.1E-05	354.2	404.8
I-10-5-tight-tard-2	7.8E-03	1.7E-05	1.7E-05	1.4E-05	1.4E-05	456.5	554.4
I-10-5-loose-equal-1	8.6E-03	1.5E-05	1.5E-05	1.4E-05	1.4E-05	575.3	616.4
I-10-5-loose-equal-2	7.8E-03	1.9E-05	1.9E-05	1.4E-05	1.4E-05	408.5	554.4
I-10-5-loose-tard-1	7.4E-03	4.2E-05	4.2E-05	4.0E-05	4.0E-05	175.3	184.1
I-10-5-loose-tard-2	9.2E-03	1.4E-05	1.4E-05	1.3E-05	1.3E-05	658.3	708.9
I-15-5-tight-equal-1	1.5E-02	2.2E-05	5.8E-05	2.2E-05	8.0E-05	671.2	671.2
I-15-5-tight-equal-2	1.4E-02	2.3E-05	7.7E-05	2.3E-05	6.4E-05	624.0	624.0
I-15-5-tight-tard-1	1.4E-02	2.1E-05	6.0E-05	2.1E-05	5.8E-05	672.2	672.2
I-15-5-tight-tard-2	1.4E-02	2.3E-05	6.9E-05	2.2E-05	5.8E-05	606.0	633.6
I-15-5-loose-equal-1	1.4E-02	2.2E-05	7.3E-05	2.2E-05	7.7E-05	636.3	636.3
I-15-5-loose-equal-2	1.4E-02	2.0E-05	6.3E-05	2.0E-05	6.0E-05	698.4	698.4
I-15-5-loose-tard-1	1.4E-02	2.1E-05	6.5E-05	2.0E-05	5.4E-05	667.6	701.0
I-15-5-loose-tard-2	1.5E-02	6.4E-05	1.6E-04	6.7E-05	1.6E-04	241.5	230.7
I-20-5-tight-equal-1	2.3E-02	3.2E-05	1.4E-04	3.2E-05	1.2E-04	724.1	724.1
I-20-5-tight-equal-2	2.3E-02	3.4E-05	2.2E-04	3.4E-05	1.1E-04	689.4	689.4
I-20-5-tight-tard-1	2.3E-02	2.9E-05	1.0E-04	2.9E-05	1.1E-04	800.1	800.1
I-20-5-tight-tard-2	2.3E-02	3.2E-05	1.2E-04	3.3E-05	1.1E-04	721.6	699.8
I-20-5-loose-equal-1	2.3E-02	2.9E-05	1.2E-04	2.9E-05	1.0E-04	801.2	801.2
I-20-5-loose-equal-2	2.3E-02	3.2E-05	2.1E-04	3.1E-05	1.1E-04	723.7	747.0
I-20-5-loose-tard-1	2.3E-02	2.9E-05	1.7E-04	2.9E-05	1.0E-04	795.6	795.6
I-20-5-loose-tard-2	2.4E-02	2.9E-05	1.2E-04	2.9E-05	1.2E-04	835.4	835.4
Average	1.5E-02	2.6E-05	8.4E-05	2.5E-05	6.8E-05	603.4	629.9

5.2. Performance comparison on ETSP instances

Tables 5, 6 and 7 show the results obtained with the OT algorithms and CPLEX for the ETSP instances from literature with 2, 5 and 10 machines, respectively. We have considered both the enumeration and the improved methods represented in the tables as OT1 and OT2, respectively. The TF values of OT1 and OT2 in the tables reveal that the OT algorithms are approximately 50 to 1500 times faster than CPLEX in generating the schedules. The comparison between the average values of AVG of the OT algorithms for the instances with 2, 5 and 10 machines reveals that the performance of the OT algorithms has negligible influence on the increase in number of machines. However, the average TF values of the OT algorithms increase with the increase in the number of machines due to the increase in AVG values of CPLEX. This shows that CPLEX is more influenced by the increase in problem size than the OT algorithms. The comparison between the average TF values of OT1 and OT2 reveals that the performance of the OT2 algorithm is slightly better than that of the OT1 algorithm. The comparison in terms of AVG and MAX values of the OT algorithms with that of CPLEX reveals that irrespective of the problem size, the OT algorithms consistently outperform CPLEX.

In addition to the problem instances from literature, we generated larger size ETSP instances with upto 50 jobs and 30 machines to analyze if the performance of CPLEX would get closer or perform better than the OT algorithms. Table 8 shows the results obtained with OT algorithms and CPLEX for the newly generated larger size instances. The results reveal that, with the increase in problem size, the OT algorithms perform much better than CPLEX. The OT algorithms generated schedules approximately 15000 times faster than that of CPLEX for instances with 50 jobs and 30 machines.

TABLE 7. Computational results for the ETSP instances from literature with 10 machines.

Problem	Computation time (in seconds)					TF	
	CPLEX	OT1		OT2			
	AVG	AVG	MAX	AVG	MAX	OT1	OT2
I-10-10-tight-equal-1	2.4E-02	1.6E-05	9.7E-05	1.6E-05	5.5E-05	1527.2	1527.2
I-10-10-tight-equal-2	2.1E-02	1.7E-05	5.5E-05	1.7E-05	4.7E-05	1237.6	1237.6
I-10-10-tight-tard-1	2.1E-02	1.5E-05	6.3E-05	1.5E-05	4.5E-05	1405.5	1405.5
I-10-10-tight-tard-2	2.1E-02	3.0E-05	7.7E-05	2.9E-05	8.2E-05	704.3	728.6
I-10-10-loose-equal-1	2.1E-02	3.8E-05	8.0E-05	3.7E-05	1.2E-04	551.1	565.9
I-10-10-loose-equal-2	2.1E-02	3.2E-05	6.4E-05	3.1E-05	4.7E-05	660.0	681.3
I-10-10-loose-tard-1	2.1E-02	3.6E-05	8.6E-05	3.5E-05	1.2E-04	584.3	601.0
I-10-10-loose-tard-2	2.2E-02	2.6E-05	8.4E-05	2.5E-05	7.7E-05	837.3	870.8
I-15-10-tight-equal-1	4.5E-02	2.3E-05	9.0E-05	2.3E-05	1.0E-04	1969.7	1969.7
I-15-10-tight-equal-2	4.5E-02	2.3E-05	1.1E-04	2.3E-05	9.4E-05	1957.3	1957.3
I-15-10-tight-tard-1	4.5E-02	5.9E-05	1.7E-04	5.5E-05	1.4E-04	758.6	813.8
I-15-10-tight-tard-2	4.5E-02	2.4E-05	1.1E-04	2.4E-05	8.1E-05	1878.4	1878.4
I-15-10-loose-equal-1	4.5E-02	4.7E-05	1.7E-04	4.4E-05	2.2E-04	955.5	1020.6
I-15-10-loose-equal-2	4.5E-02	4.0E-05	1.4E-04	3.7E-05	1.3E-04	1122.7	1213.7
I-15-10-loose-tard-1	4.5E-02	5.6E-05	2.1E-04	5.2E-05	1.9E-04	802.3	864.1
I-15-10-loose-tard-2	4.6E-02	2.6E-05	1.0E-04	2.6E-05	3.1E-04	1785.4	1785.4
I-20-10-tight-equal-1	7.8E-02	3.3E-05	3.1E-04	3.3E-05	7.4E-04	2375.2	2375.2
I-20-10-tight-equal-2	7.8E-02	3.3E-05	1.2E-04	3.3E-05	1.8E-04	2362.6	2362.6
I-20-10-tight-tard-1	7.9E-02	3.6E-05	1.9E-04	3.6E-05	1.4E-04	2201.3	2201.3
I-20-10-tight-tard-2	7.8E-02	3.5E-05	9.2E-04	3.5E-05	1.5E-04	2241.7	2241.7
I-20-10-loose-equal-1	7.9E-02	3.3E-05	2.2E-04	3.3E-05	2.1E-04	2393.1	2393.1
I-20-10-loose-equal-2	7.8E-02	3.7E-05	1.4E-04	3.6E-05	1.2E-04	2120.2	2179.1
I-20-10-loose-tard-1	7.9E-02	3.3E-05	2.1E-04	3.3E-05	1.7E-04	2382.2	2382.2
I-20-10-loose-tard-2	8.2E-02	5.8E-05	2.7E-04	5.5E-05	2.8E-04	1415.6	1492.8
Average	4.9E-02	3.4E-05	1.7E-04	3.3E-05	1.6E-04	1509.5	1531.2

6. CONCLUSIONS

In this paper, we presented exact algorithms to generate optimal timing schedules for two job shop scheduling scenarios, namely JIT-JSP and ETSP. In JIT-JSP, each operation has a due date and the associated weights to penalize its earliness and tardiness. The scheduling objective of JIT-JSP involves minimization of the weighted sum of earliness and tardiness associated with the deviation of completion time of each operation from its respective due date. On the other hand, in ETSP, only the last operation of each job has a due date and the associated weights to penalize its earliness and tardiness. The scheduling objective of ETSP involves minimization of the weighted sum of earliness and tardiness associated with the deviation of completion time of each job from its respective due date. We proposed two OT algorithms to generate optimal schedules, which can be used for both scheduling scenarios. The first method, namely OT1, is an enumeration method. The second method, namely OT2, improves the first method that uses dominance rules to reduce the solution space, thereby improving the computation time. The performance of the OT Algorithms, OT1 and OT2, was compared with the CPLEX solver for several JIT-JSP and ETSP instances. The computational experiments revealed that the improved method (OT2) performed slightly better than the enumeration method (OT1) on all the problem instances. Though the OT algorithms have exponential complexity, the computational study revealed that they are practical in generating schedules in reasonable computation time and competitive with CPLEX for small and medium size JIT-JSP instances. In the case of ETSP instances, the OT algorithms generated schedules in short computation time and consistently outperformed CPLEX in all the problem instances.

TABLE 8. Computational results for larger size ETSP instances.

Problem	Computation time (in seconds)					TF	
	CPLEX	OT1		OT2			
	AVG	AVG	MAX	AVG	MAX	OT1	OT2
N-25-20-tight-equal	0.47	8.5E-05	4.7E-04	8.5E-05	2.9E-04	5516.5	5516.5
N-25-20-tight-tard	0.47	8.6E-05	3.2E-04	8.6E-05	2.5E-04	5465.6	5465.6
N-25-20-loose-equal	0.47	8.3E-05	2.5E-04	8.2E-05	2.6E-04	5629.8	5698.5
N-25-20-loose-tard	0.46	8.7E-05	3.7E-04	8.8E-05	3.1E-04	5344.4	5283.7
N-30-20-tight-equal	0.66	1.1E-04	3.7E-04	1.1E-04	3.2E-04	6014.9	6014.9
N-30-20-tight-tard	0.66	1.0E-04	3.4E-04	1.0E-04	3.0E-04	6450.5	6450.5
N-30-20-loose-equal	0.66	1.1E-04	1.6E-03	1.1E-04	6.4E-04	6253.9	6253.9
N-30-20-loose-tard	0.66	1.0E-04	3.3E-04	1.0E-04	2.9E-04	6359.5	6421.2
N-35-20-tight-equal	0.89	1.4E-04	7.8E-03	1.4E-04	5.9E-04	6535.1	6583.5
N-35-20-tight-tard	0.90	1.5E-04	5.3E-04	1.5E-04	4.9E-04	6001.3	6041.5
N-35-20-loose-equal	0.90	1.3E-04	3.7E-04	1.3E-04	3.5E-04	7005.1	7060.3
N-35-20-loose-tard	0.90	1.3E-04	1.1E-02	1.3E-04	9.4E-03	6872.6	6872.6
N-40-30-tight-equal	2.57	2.4E-04	8.1E-04	2.3E-04	6.1E-04	10909.4	11002.6
N-40-30-tight-tard	2.57	2.4E-04	6.1E-04	2.4E-04	7.1E-04	10561.9	10649.6
N-40-30-loose-equal	2.57	2.2E-04	5.8E-04	2.2E-04	5.5E-04	11837.4	11892.2
N-40-30-loose-tard	2.57	1.9E-04	6.1E-04	1.9E-04	5.7E-04	13326.7	13396.1
N-45-30-tight-equal	3.30	2.5E-04	5.4E-03	2.5E-04	1.3E-03	13379.3	13433.6
N-45-30-tight-tard	3.22	2.3E-04	1.8E-02	2.3E-04	1.1E-02	14198.0	14260.8
N-45-30-loose-equal	3.29	2.3E-04	1.6E-02	2.3E-04	1.6E-02	14250.8	14189.4
N-45-30-loose-tard	3.24	2.5E-04	6.5E-04	2.5E-04	6.1E-04	12758.2	12758.2
N-50-30-tight-equal	3.98	2.8E-04	1.2E-03	2.8E-04	6.8E-04	14354.4	14406.4
N-50-30-tight-tard	4.03	3.3E-04	5.3E-03	3.3E-04	4.0E-03	12129.3	12165.9
N-50-30-loose-equal	4.02	2.8E-04	1.4E-03	2.8E-04	1.5E-03	14373.8	14271.8
N-50-30-loose-tard	3.99	2.7E-04	3.3E-03	2.7E-04	2.5E-03	14557.2	14610.5

To the best of our knowledge, this is the first reported study on exact approaches for generating optimal schedules in job shop scheduling problems with TWET minimization objective. Future research can be directed towards improving the proposed OT algorithms to reduce their computational complexity. The schedule generation mechanism in the proposed OT algorithms allows them to be used with priority dispatching rules. Therefore, a future research direction would be developing and implementing priority dispatching rules for the static and dynamic job shop scheduling problems. The proposed OT algorithms can be employed to generate schedules within heuristic and metaheuristic approaches. Therefore, future research can also be directed towards developing efficient heuristic and metaheuristic approaches incorporating the proposed OT algorithms. Future research can also directed towards extending the proposed OT algorithms to generate schedules in other related multi-machine scheduling problems.

Acknowledgements. The authors are thankful to the Area Editor and the anonymous reviewers for giving constructive comments for improving this paper.

REFERENCES

[1] N.R. Adam and J. Surkis, Priority update intervals and anomalies in dynamic ratio type job shop scheduling rules. *Manage. Sci.* **26** (1980) 1227–1237.

[2] M.M. Ahmadian and A. Salehipour, The just-in-time job shop scheduling problem with distinct due-dates for operations. *J. Heuristics* **27** (2021) 175–204.

[3] M.M. Ahmadian, A. Salehipour and T.C.E. Cheng, A meta-heuristic to solve the just-in-time job-shop scheduling problem. *Eur. J. Oper. Res.* **288** (2021) 14–29.

[4] R.P. Araujo, A.G. dos Santos and J.E.C. Arroyo, Genetic Algorithm and Local Search for Just-in-Time Job–Shop Scheduling. In: Proceedings of the 2009 IEEE Congress on Evolutionary Computation (CEC 2009) (2009) 955–961.
[5] P. Baptiste, M. Flamini and F. Sourd, Lagrangian bounds for just-in-time job-shop scheduling. *Comput. Oper. Res.* **35** (2008) 906–915.
[6] J. Bauman and J. Józefowska, Minimizing the earliness–tardiness costs on a single machine. *Comput. Oper. Res.* **33** (2006) 3219–3230.
[7] J.C. Beck and P. Refalo, A hybrid approach to scheduling with earliness and tardiness costs. *Ann. Oper. Res.* **118** (2003) 49–71.
[8] J.H. Blackstone, D.T. Phillips and G.L. Hogg, A state-of-the-art survey of dispatching rules for manufacturing job shop operations. *Int. J. Prod. Res.* **20** (1982) 27–45.
[9] F.D. Croce and M. Trubian, Optimal idle time insertion in early-tardy parallel machines scheduling with precedence constraints. *Prod. Plan. Control.* **13** (2002) 133–142.
[10] P. Chretienne, Minimizing the earliness and tardiness cost of a sequence of tasks on a single machine. *RAIRO-Oper. Res.* **35** (2001) 165–187.
[11] P. Chretienne and F. Sourd, PERT scheduling with convex cost functions. *Theor. Comput. Sci.* **292** (2003) 145–164.
[12] T. Cleveland, Number Theory. Ed-Tech Press (2020).
[13] E. Danna, E. Rothberg and C.L. Pape, Integrating mixed integer programming and local search: A case on Job-shop scheduling problems. In: Proceedings of the Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR'03) (2003) 65–79.
[14] A.G. dos Santos, R.P. Araujo and J.E.C. Arroyo, A combination of evolutionary algorithm, mathematical programming, and a new local search procedure for the just-in-time job-shop scheduling problem. In Vol. 6073 of Learning and Intelligent Optimization (LION 2010), edited by C. Blum and R. Battiti, Lecture Notes in Computer Science, Springer-Verlag, Berlin-Heidelberg (2010) 10–24.
[15] G. Feng and H.C. Lau, Efficient algorithms for machine scheduling problems with earliness and tardiness penalties. *Ann. Oper. Res.* **159** (2008) 83–95.
[16] M.R. Garey, R.E. Tarjan and G.T. Wilfong, One-Processor Scheduling with Symmetric Earliness and Tardiness Penalties. *Math. Oper. Res.* **13** (1988) 330–348.
[17] B. Giffler and G.L. Thompson, Algorithms for solving production-scheduling problems. *Oper. Res.* **8** (1960) 487–503.
[18] B.S. Girish, An efficient hybrid particle swarm optimization algorithm in a rolling horizon framework for the aircraft landing problem. *Appl. Soft Comput.* **44** (2016) 200–221.
[19] Y. Hendel and F. Sourd, An improved earliness–tardiness timing algorithm. *Comput. Oper. Res.* **34** (2007) 2931–2938.
[20] IBM software, IBM ILOG CPLEX Optimization Studio CPLEX User's Manual version 12 Release 8, *IBM*, 2017. Available online: https://www.ibm.com/support/pages/introduction-concert-technology.
[21] A.S. Jain and S. Meeran, Deterministic job-shop scheduling: Past, present and future. *Eur. J. Oper. Res.* **113** (1999) 390–434.
[22] J. Józefowska, Just-in-time concept in manufacturing and computer systems, In: Just-In-Time Scheduling: Models and Algorithms for Computer and Manufacturing Systems, Vol. 106 of International Series In Operations Research, Springer, Boston (2007) 1–23.
[23] J. Kelbel and Z. Hanzalek, Solving production scheduling with earliness/tardiness penalties by constraint programming. *J. Intell. Manuf.* **22** (2011) 553–562.
[24] C.Y. Lee and J.Y. Choi, A genetic algorithm for job sequencing problems with distinct due dates and general early-tardy penalty weights. *Comput. Oper. Res.* **22** (1995) 857–869.
[25] J. Monette, Y. Deville and P.V. Hentenryck, Just-in-time scheduling with constraint programming. In: Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009) (2009) 241–248.
[26] S.G. Ponnambalam, N. Jawahar and B.S. Girish, Giffler and Thompson procedure based genetic algorithms for scheduling job shops, In: Computational intelligence of flow shop and job shop scheduling, In Vol. 230 of Studies in Computational Intelligence, Springer-Verlag, Berlin-Heidelberg (2010) 229–259.
[27] W. Szwarc and S.K. Mukhopadhyay, Optimal timing schedules in earliness-tardiness single machine sequencing. *Nav. Res. Logist.* **42** (1995) 1109–1114.
[28] M. Vanhoucke, E. Demeulemeester and W. Herroelen, An exact procedure for the resource-constrained weighted earliness–Tardiness project scheduling problem. *Ann. Oper. Res.* **102** (2001) 179–196.
[29] G. Wan and B.P.C. Yen, Tabu search for single machine scheduling with distinct due windows and weighted earliness/tardiness penalties. *Eur. J. Oper. Res.* **142** (2002) 271–281.
[30] S. Wang and Y. Li, Variable neighbourhood search and mathematical programming for just-in-time job-shop scheduling problem. *Math. Probl. Eng.* **2014** (2014) 1–9.
[31] H. Yang, J. Li and L. Qi, An Improved Genetic Algorithm For Just-In-Time Job-Shop Scheduling Problem. *Adv. Mat. Res.* **472–475** (2012) 2462–2467.
[32] H. Yang, Q. Sun, C. Saygin and S. Sun, Job shop scheduling based on earliness and tardiness penalties with due dates and deadlines: an enhanced genetic algorithm. *Int. J. Adv. Manuf. Technol.* **61** (2012) 657–666.

[33] J. Zhang, G. Ding, Y. Zhou, S. Qin and J. Fu, Review of job shop scheduling research and its new perspectives under industry 4.0. *J. Intell. Manuf.* **30** (2019) 1809–1830.