mathematics

MDPI

Article

Scheduling of Software Test to Minimize the Total Completion Time †

Man-Ting Chao and Bertrand M. T. Lin *

Institute of Information Management, National Yang Ming Chiao Tung University, Hsinchu 300, Taiwan; manting417@gmail.com

* Correspondence: bmtlin@nycu.edu.tw

† This document is based upon the thesis of Man-Ting Chao submitted for her master degree.

Abstract: This paper investigates a single-machine scheduling problem of a software test with shared common setup operations. Each job has a corresponding set of setup operations, and the job cannot be executed unless its setups are completed. If two jobs have the same supporting setups, the common setups are performed only once. No preemption of any processing is allowed. This problem is known to be computationally intractable. In this study, we propose sequence-based and position-based integer programming models and a branch-and-bound algorithm for finding optimal solutions. We also propose an ant colony optimization algorithm for finding approximate solutions, which will be used as the initial upper bound of the branch-and-bound algorithm. The computational experiments are designed and conducted to numerically appraise all of the proposed methods.

Keywords: single-machine scheduling; shared common setups; total completion time; integer programming; branch-and-bound; ant colony optimization

MSC: 68M20; 90B35; 90C57

Citation: Chao, M.-T.; Lin, B.M.T. Scheduling of Software Test to Minimize the Total Completion Time. *Mathematics* **2023**, *11*, 4705. https://doi.org/10.3390/math11224705

Academic Editor: Ripon Kumar Chakrabortty

Received: 9 October 2023
Revised: 11 November 2023
Accepted: 15 November 2023
Published: 20 November 2023

1. Introduction

Scheduling is the decision-making process used by many manufacturing and service industries to allocate resources to economic activities or tasks over he planning horizon [1,2]. This paper studies a scheduling model that is inspired by real-life applications, where supporting operations need to be prepared before regular jobs are processed. The specific application context is the scheduling of asoftware test at an IC design company, where the software system is modular and can be tested module by module and level by level. Before starting a module test, we need to install software utilities and libraries as well as adjust system parameters to shape an appropriate system environment. The setup operation corresponds to the installation of utilities and libraries, which are supporting tasks for the job. Different tests may require part of the same environment settings. If two jobs have common supporting tasks, the common setups are performed only once. The abstract model was also studied by Kononov, Lin, and Fang [3] as a single-machine scheduling problem formulated from the production scheduling of multimedia works. In the context of multimedia scheduling, when we want to play multimedia, we need to download their content first, including audio tracks, subtitles, and images, which can correspond to setup operations and jobs for this study, respectively. Once the setup operations of the multimedia objects are prepared, they can be embedded in upper-level objects without multiple copies, as in physical products. This unique property is different from the manufacture of tangible products, such as vehicles and computers. Following the standard three-field notation [4], we denote the model by $1|bp - prec|\sum_j C_j$, where the one indicates the single-machine setting, $bp - prec$ indicates the bipartite precedence relation between shared setups and test jobs, and $\sum_j C_j$ is the objective to minimize the total completion time.

This paper is organized into seven sections. In Section 2, the problem definition is presented with a numerical example for illustrations. The literature review follows. Section 3 introduces two integer programming models based on different formulation approaches. Section 4 is dedicated to the development of a branch-and-bound algorithm, including the development of upper bound, lower bound, and tree traversal methods. In Section 5, an ant colony optimization algorithm is proposed. Section 6 presents the computational experiments on the proposed methods. Finally, conclusions and suggestions for future works are given in Section 7.

2. Problem Definition and Literature Review

2.1. Problem Statements

We first present the notation that will be used in this paper. Note that all parameters are assumed to be non-negative integers.

n	number of jobs;
m	number of setup operations;
$T = \{t_1, t_2, \ldots, t_n\}$	set of jobs to be processed;
$S = \{s_1, s_2, \ldots, s_m\}$	set of setup operations;
$\mathcal{R} = \{(s_i, t_j) \mid s_i \in S \; supports \; t_j \in T\}$	relation indicating whether the setup operation s_i is required for each job t_j;
p_j	processing time of job t_j on the machine;
sp_i	processing time of setup s_i on the machine;
$\sigma = (\sigma_1, \sigma_2, ..., \sigma_n)$	particular sequence of the jobs;
σ^*	optimal schedule sequence;
C_j	completion time of job t_j;
$Z(\sigma) = \sum_{j \in T}$	total job completion time under schedule σ.

The subject of our research is dedicated to studying the single-machine scheduling problem with shared common setup operations. The objective is to minimize the total completion time of the jobs, i.e., ΣC_j. The problem can be described as follows:

From time zero onwards, two disjoint sets of activities S and T are to be processed on a machine. Each job t_j has a set of setup operations that job t_j can only start after its setups are completed. All setup operations and jobs can be performed on the machine at any time. Although all setup operations need to be processed once, they do not contribute to the objective function because their role is only the preparatory operations for jobs under the priority relation. At any time, the machine can process at most one setup operation or job. No preemption of any processing is allowed. In software test scheduling, jobs t_j represent the software to be tested, and setups s_i refers to the preparation of a programming language or compilation environment that needs to be installed in advance so that the test software can be executed. For example, if t_1 is an Android application that needs to be tested, and t_1 needs setups s_1 and s_4, then s_1 could be JAVA, s_4 could be Android Studio, etc.

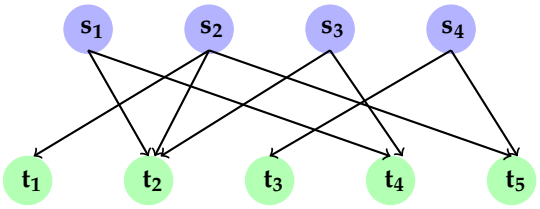
To illustrate the problem of our study, we give numerical examples of four setup operations, five test jobs, the relation $\mathcal{R}$, and its corresponding graph. The parameters and relation are shown in Figure 1.

Two example schedules are produced and shown below in the form of Gantt charts. The feasible schedule shown in Figure 2 is $\sigma = (t_1, t_2, t_3, t_4, t_5)$ with a total completion time of 115. Setups s_2, s_3 precede job t_1. The order of s_2 and s_3 is immaterial. Figure 3 shows schedule $\sigma^* = (t_4, t_2, t_1, t_3, t_5)$ associated with a total completion time of 101, which is optimal for the given instance. While both are feasible solutions, the objective value of Figure 3 will be better than that of Figure 2.

activities	s_1	s_2	s_3	s_4	t_1	t_2	t_3	t_4	t_5
length	2	4	3	2	5	3	6	2	7

$\mathcal{R}(s_i, t_j)$	t_1	t_2	t_3	t_4	t_5
s_1	0	1	0	1	0
s_2	1	1	0	0	1
s_3	1	0	0	1	0
s_4	0	0	1	0	1

(**a**) Parameters of five jobs.



(**b**) Bipartite relation.

Figure 1. Example of 4 setups and 5 jobs.

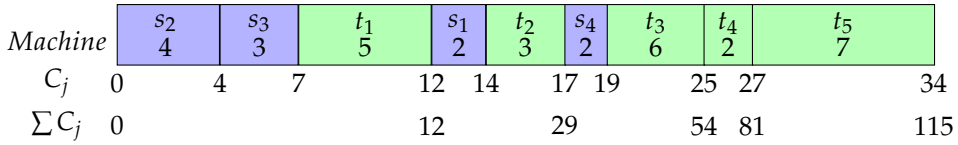


Figure 2. Feasible solution.

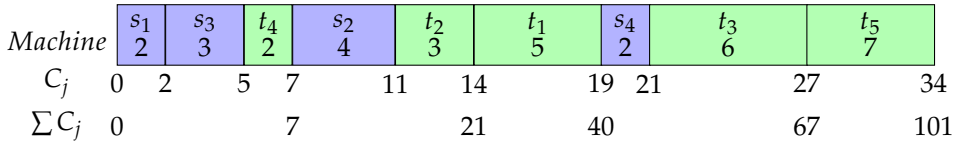


Figure 3. Optimal solution.

2.2. Literature Review

The scheduling problem studied by Kononov, Lin, and Fang [3] has a single machine that performs all setup operations and testing jobs. Referring to Brucker [5] and Leung [1], we know that precedence constraints play a crucial role in scheduling problems, especially when complexity status or categories are involved. Existing research works in the literature consider precedence relations presented in various forms. Bipartite graphs are often studied in graph theory. The graph shown in Figure 1 is bipartite because edges exist between nodes on one side and nodes on the other. Linear allocation problems can also be visualized by both supply and demand. Unfortunately, scheduling theory rarely addresses precedence constraints in bipartite graphs.

After formulation, Kononov, Lin, and Fang [3] studied two minimum sum objective functions, namely the number of late jobs and the total weighted completion time of jobs. As for the minimization of the total weighted completion time ($\sum_j w_j C_j$), Baker [6] is probably the first paper to address the existence of precedence constraints. Adolphson and Hu [7] proposed a polynomial time algorithm for the case in which rooted trees give priority. A fundamental problem for the jobs-per-unit execution time is $1|prec, p_i = 1|\sum w_j C_j$, where $w_j \in 1, 2, 3$ has been proven to be strongly NP-hard by Lawler [8]. The minimum latency set cover problem studied by Hassin and Levin [9] is the most relevant. The minimum

latency set cover problem involves a subset of several operations. A subset is complete when all of their operations are finished. The objective function is the total weighted completion time of subsets. The minimum latency set cover problem is a special case of our $1|bp - prec|\sum w_jC_j$ problem, and the correspondence is as follows: In our problem, an operation is mapped to a setup operation (s_i), and a subset is interpreted as a testing job (t_j). Furthermore, Hassin and Levin [9] showed that the minimum delay set cover problem is strongly NP-hard even if all operations require the unit execution time (UET). Subsequently, the $1|bp - prec|\sum w_jC_j$ problem is hard as well. By performing a pseudo-polynomial time reduction in Lawler's result about $1|prec, p_j = 1|\sum w_jC_j$, where $w_j \in 1, 2, 3$, Kononov, Lin, and Fang [3] proved $1|bp - prec, s_i = t_j = 1|\sum C_j$ is strongly NP-hard. In other words, it is very difficult to minimize the total weighted completion time in our model, even though all setup operations and all testing jobs require unit execution time and all weights are one.

To solve the scheduling problem, Shafransky and Strusevich [10]; Hwang, Kovalyov, and Lin [11]; and Cheng, Kravchenko, and Lin [12] studied several special cases with fixed job sequences and solved these problems in polynomial time. Moreover, the branch-and-bound algorithm is an enumeration technique that can be applied to combinatorial optimization problems. Brucker and Sievers [13] deploy branch-and-bound algorithms on the job-shop scheduling problem and Hadjar, Marcotte, and Francois [14] do the same on the multiple-depot vehicle scheduling problem. To find approximate solutions to a hard optimization problem, various meta-heuristics have been designed. Kunhare, Tiwari, and Dhar [15] used particle swarm optimization for feature selection in intrusion detection systems. Kunhare, Tiwari, and Dhar [16] further used a genetic algorithm to compose a hybrid approach to intrusion detection. For solving a worker assignment bi-level programming problem, Luo, Zhang, and Yin [17] designed a two-level algorithm, which simulated annealing as the upper level to minimize the worker idle time and the genetic algorithm as the lower level to minimize the production time. For more general coverage, the reader is referred to Ansari and Daxini [18] and Rachih, Mhada, and Chiheb [19]. Ant colony optimization (ACO) is a meta-heuristic algorithm that can be used to find approximate solutions to difficult optimization problems. Many research studies in the literature also use ACO to solve scheduling problems, such as Blum and Sampels [20] on group shop scheduling problems; Yang, Shi, and Marchese [21] on generalized TSP problems; and Xiang, Yin, and Lim [22] on operating room surgery scheduling problems. According to the above, it is known that the branch-and-bound algorithm and ACO may be effective in solving the scheduling problem in our study.

3. Integer Programming Models

In this section, to mathematically present the studied problem $1|bp - prec|\Sigma_jC_j$, we formulate two integer programming models. Since the problem's nature is set on permutations of jobs, we deploy two common approaches, sequence-based decision variables and position-based decision variables, for shaping permutation-based optimization problems. The models will be then implemented and solved by the off-the-shelf Gurobi Optimizer.

3.1. Position-Based IP

In the section, we focus on the decision that assigns $m + n$ activities at $m + n$ positions. Activities $1, 2, \ldots, m$ are setups and activities $m + 1, m + 2, \ldots,$ and $m + n$ are jobs. Therefore, an activity could be either a job or a setup operation. In the model, there are six categories of constraints; $(m + n)^2$ binary variables x; and two subsets of $m + n$ integer variables, e_l and C_k. Index $k \in \{1, 2, \ldots, m + n\}$ indicates the positions. We use the binary relation $(i, j) \in \mathcal{R}$ to indicate whether the setup i should finish before the job j starts. The variables used in the model are defined in the following:
Decision variables:

$x_{i,k} = 1$ if the activity i is in the position k; 0, otherwise.

Auxiliary variables:

$$\begin{aligned} p_l: &\quad \text{processing time of the activity } l; \\ C_k: &\quad \text{completion time of the job in the position } k. \end{aligned}$$

Note that extra variables, C'_k, are introduced for extracting the completion times of jobs. If a position k is loaded with a setup, then $C'_k \geq -M$, where M is a big number.
Position-based IP:

$$\min \quad \sum_{k=1}^{m+n} C'_k$$

$$\text{s.t.} \quad \sum_{k=1}^{m+n} x_{l,k} = 1, \qquad \text{activity } l \in \{1, \ldots, m+n\} \text{ is assigned to a position;} \quad (1)$$

$$\sum_{l=1}^{m+n} x_{l,k} = 1, \qquad \text{position } k \text{ accommodates one activity;} \quad (2)$$

$$\sum_{k=1}^{m+n} x_{i,k} \cdot k \leq \sum_{k=1}^{m+n} x_{j,k} \cdot k, \qquad (i,j) \in \mathcal{R}, 1 \leq i \leq m, m+1 \leq j \leq m+n; \quad (3)$$

$$C_1 = \sum_{l=1}^{m+n} x_{l,1} \cdot p_l, \qquad \text{completion time of the first position;} \quad (4)$$

$$C_k \geq C_{k-1} + \sum_{l=1}^{m+n} x_{l,k} \cdot p_l, \qquad \text{completion time of position } k \in \{1, \ldots, m+n\}; \quad (5)$$

$$C'_k \geq C_k - (\sum_{i=1}^{m} x_{i,k})M, \quad (6)$$

$$x_{l,k} \in \{0,1\}, \qquad 1 \leq l, k \leq m+n; \quad (7)$$

$$C_k \geq 0, C'_k \geq 0, \qquad 1 \leq k \leq m+n. \quad (8)$$

The goal is to minimize the total completion time of jobs. Constraint (1) lets each position accommodate exactly one job or one setup. Constraint (2) lets each activity be assigned to exactly one position. Constraint (3) ensures that any job j can start only after its setup operations, i, are all finished. Constraint (4) lets the completion time of the first position be greater than or equal to the processing time of the event that occupied the first position. Constraint (5) defines the completion time of the position k to be greater than or equal to the completion time of $k-1$ plus the processing time of the event that is processed in the position k. Constraint (6) defines the completion time k' if the position k contains a job. The reason we added a variable is that if the objective function computes the completion time of jobs in $\sum_{k=1}^{m+n} \sum_{j=m+1}^{m+n} C_k \cdot x_{jk}$; it becomes quadratic. Therefore, we add an extra variable, C', to make the objective function linear, i.e., $\sum_{k=1}^{m+n} C'_k$.

3.2. Sequence-Based IP

In this section, the formulation approach is to determine the relative positions between each two activities. The model consists of five categories of constraints; $(m+n)^2$ binary variables x; and two subsets of $m+n$ integer variables, p_j and C_k.

Decision variable:

$$x_{i,j} = 1 \text{ if activity } i \text{ precedes the activity } j; 0, \text{ otherwise.}$$

Auxiliary variables:

p_l : processing time of the activity l;
C_k : completion time of the activity k.

Sequence-based IP:

$$\min \quad \sum_{k=m+1}^{m+n} C_k$$

$$\text{s.t.} \quad x_{i,j} + x_{j,i} = 1, \qquad i \neq j \in \{1, \ldots, m+n\}; \tag{9}$$

$$x_{j,i} = 0, \qquad (i,j) \in \mathcal{R}, 1 \leq i \leq m, m+1 \leq j \leq m+n; \tag{10}$$

$$C_j \geq C_i + p_j + (x_{i,j} - 1)M, \qquad i \neq j \in \{1, \ldots, m+n\}; \tag{11}$$

$$C_i \geq sp_i, \qquad 1 \leq i \leq m; \tag{12}$$

$$C_j \geq p_j + \sum_{(i,j)\in\mathcal{R}} sp_i, \qquad m+1 \leq j \leq m+n; \tag{13}$$

$$x_{i,j} \in \{0,1\}, \qquad 1 \leq i,j \leq m+n; \tag{14}$$

$$C_k \geq 0, \qquad 1 \leq k \leq m+n. \tag{15}$$

The objective value is to minimize the total completion time of jobs except for setup operations. Constraint (9) limits the precedence between the two events. Constraint (10) means that if job j needs setup i, then setup i should come before job j. Constraint (11) lets the completion time of job j be greater than or equal to the completion time of job i plus the processing time of job j if job i precedes job j. Constraint (12) defines the completion time of event i if it is a setup. Constraint (13) defines the completion time of event j if it is a job.

4. Branch-and-Bound Algorithm

In this section, we explore a search tree that generates all permutations of jobs. In the branch-and-bound algorithm, there will be an upper bound representing the current best solution during the search process. In the process of searching, each node will calculate the lower bound once, and if the lower bound calculated is not better than the upper bound, the subtree of the node will be pruned to speed up the search. Therefore, we propose an upper bound as the initial solution, a lower bound for pruning non-promising nodes, and a property to check whether each node satisfies the condition when pruning the tree.

4.1. Upper Bound

First, we use an ACO algorithm coupled with local search to find an approximate solution as an upper bound, sorted by the settings of pheromone and visibility. Details about the ACO algorithm will be introduced in Section 5. Implementing a branch-and-bound algorithm with tight upper bounds helps converge the solution process faster.

4.2. Lower Bound

Lower bounds can help cut unnecessary branches that will never lead to a solution better than the incumbent one. Different approaches can be used to derive lower bounds. In our study, we compute a lower bound by sorting the remaining processing times of unscheduled jobs. We can express it as the $1|r_j|\Sigma C_j$ problem. When the setup operations of the scheduled jobs are complete, we can release these setup times. Then, we denote the unfinished setup operations as the release date of each job and implement the shortest remaining processing time (SRPT) method. The process with the least amount of time remaining before completion is selected to execute. Finally, we add the total current completion time of scheduled jobs and the result of the `SRPT` mentioned above as a lower bound. The Lower Bound algorithm is shown in Algorithm 1.

Algorithm 1: LowerBound

```
Function LowerBound(σ):
    LB = 0;
    if length(σ) == n then
        LB = sum(σ);
    else
        sort the unscheduled jobs by the shortest remaining processing time;
        LB = sum(σ)+SRPT(σ_unscheduled);
    return LB.
```

4.3. Dominance Property

In this section, after we use the lower bound to prune nodes, we also propose a property of the branch-and-bound algorithm (Algorithm 2), which can also speed up node pruning and reduce tree traversal time. The content description and proof of the property are as follows:

Lemma 1. *Let $J = \{j_1, j_2, ..., j_k\}$ be the unscheduled jobs at a node X in the branch-and-bound tree. For any unscheduled job j_a, if there is another unscheduled job j_b such that the setups of j_b are all scheduled and $p_b \leq p_a$, then the subtree $X + j_a$ by choosing j_a as the next job to schedule can be pruned off because j_b precedes j_a in some optimal solution.*

Proof. Let σ be the sequence of scheduled jobs. Assume that there is an optimal solution (σ, j_a, L, j_b), where L is a sequence of the unscheduled jobs. When the setups of job j_a are not yet completed, its completion time C_a would be $C_\sigma + \sum^{m}_{i \in unfinished} {}^{i \to a} sp_i + p_a$, and, when the setups of job j_b are completed, its completion time C_b would be $C_\sigma + C_a + C_L + p_b$, where C_L is the completion time of all jobs of L. Assuming the positions of j_b and j_a are swapped as (σ, j_b, L, j_a), we denote their completion times as C'_b,C'_L, and C'_a. At this point, C'_b will be $C_\sigma + p_b$, and C'_a will be $C_\sigma + C'_b + C'_L + \sum^{m}_{i \in unfinished} {}^{i \to a} sp_i + p_a$. Suppose that if p_b is less than p_a, it makes C'_b less than C_b, which also makes the completion time of L shorter, to the benefit of both job j_y and L. When both C'_y and C'_L move forward, the result of C'_a will also decrease accordingly. According to the assumption, we know that the total completion time of (σ, j_b, L, j_a) will be smaller than that of (σ, j_a, L, j_b). Therefore, we can prune off the branch of node j_a, which will not lead to a better solution without sacrificing the optimality. □

Algorithm 2: Check Property

```
Function CheckProperty(J):
    forall j_a ∈ J do
        forall j_b ∈ J and j_a ≠ j_b do
            if the setups of j_b have finished and p_b ≤ p_a then
                return False;
    return True.
```

4.4. Tree Traversal

In this section, we use three different tree traversal methods, depth-first search (DFS), breadth-first search (BFS), and best-first search (BestFS), to perform the branch-and-bound algorithm. Moreover, we also added the upper bound, lower bound, and property mentioned above into our branch-and-bound algorithm.

4.5. Depth-First Search (DFS)

DFS is a recursive algorithm for searching all the nodes of a tree and can generate the permutations of all the solutions. It starts at the root node and traverses along each branch before backtracking. The advantage of DFS is that the demand for memory is relatively low, but the disadvantage is that because of recursion, there will be a heavier loading in the stack operation, and it will take more time to find all the solutions. The DFS algorithm is shown in Algorithm 3.

First, the algorithm will obtain the upper bound from Line 15 and call the recursive `DFS` function. When we encounter the deepest node or have visited all of its children, we move backward along the current path to find the unvisited node to traverse. In the search process, we use `LowerBound()` and `CheckProperty()` to test whether we should continue to search down or not. If the lower bound is greater than or equal to the upper bound, or if the property is not met, we will prune the branch because it does not yield a better solution than the current one. This method can reduce the number of search nodes.

Algorithm 3: Depth-First Search

```
1  Function DFS(sequence, ub, σ, σ*):
2      if length(σ) == n then
3          if sum(σ) < ub then
4              ub = sum(σ);
5              σ* = σ;
6          return ub, σ*;
7      forall t_j ∈ sequence do
8          σ.append(t_j);
9          denote sequence_unscheduled as sequence without t_j;
10         if LowerBound(σ) < ub then
11             if CheckProperty(t_j, sequence_unscheduled) then
12                 ub, σ* = DFS(sequence_unscheduled, ub, σ, σ*);
13     return ub, σ*;
14 ub = UpperBound(sequence);
15 ub, σ* = DFS(sequence, ub, [], σ*);
```

4.6. Breadth-First Search (BFS)

BFS is a tree traversal algorithm that satisfies given properties. It starts at the root of the tree, traverses all nodes at the current level, and moves to the next depth level. Unlike DFS, which will find a solution first, it will wait until the last level is searched to find all suitable solutions. In particular, this method uses a queue to record the sequence of visited nodes. The advantage of BFS is that each node is traversed by the shortest path, but the disadvantage is that it requires more memory to store all of the traversed nodes. It thus takes more time to search deeper trees.

The BFS algorithm is shown in Algorithm 4. First, the algorithm will obtain the upper bound by `UpperBound()` from Line 25. The BFS function starts from Line 2; we create a queue that uses the First-In-First-Out strategy. Lines 3 through 5 are the initial settings that we use to set a root. From Lines 6 to 24, we enqueue the root node and then dequeue the values in order. Then, we enqueue the unvisited nodes and recalculate the lower bound until there is no value in the queue. Before each enqueue, it is necessary to use `LowerBound()` and `CheckProperty()` to check whether the lower bound is smaller than the upper bound and whether it satisfies the property. It can reduce the number of visited nodes and shorten the execution time. In Line 12, we use the `Without()` function to obtain the nodes that have not been visited yet. The loop stops when the queue is empty, indicating that all nodes

have been traversed.

Algorithm 4: Breadth-First Search

```
1  Function BFS(sequence, ub):
2      Let Q be a queue;
3      forall t_j ∈ sequence do
4          if LowerBound(t_j) < ub then
5              Q.enqueue(t_j);
6      while Q is not empty do
7          σ = Q.dequeue();
8          while LowerBound(σ) ≥ ub do
9              if Q is empty then
10                 break;
11             σ = Q.dequeue();
12         forall t_k ∈ Without(σ) do
13             σ.append(t_k);
14             if length(σ) == n then
15                 if sum(σ) < ub then
16                     ub = sum(σ);
17                     σ* = σ;
18             else
19                 if LowerBound(σ) < ub then
20                     if CheckProperty(t_k, σ) then
21                         Q.enqueue(t_j);
22         if Q is empty then
23             break;
24     return ub, σ*;
25 ub =UpperBound(sequence);
26 ub, σ* =BFS(sequence, ub);
```

4.7. Best-First Search (BestFS)

BestFS works as a combination of depth-first and breadth-first search algorithms. It is different from other search algorithms that blindly traverse to the next node, it uses the concept of a priority queue and heuristic search, using an evaluation function to determine to which neighbor node is the best to move. It is also a greedy strategy because it always chooses the best path at the time, rather than BFS using an exhaustive search. The advantage of BestFS is that it is more efficient because it always searches through the node with the smaller lower bound first. On the other hand, the disadvantage is that the structure of the heap is difficult to maintain and requires more memory resources. Since each visited node will be stored in the heap, we can directly obtain the node with the smallest lower bound bound by heapsort. Therefore, when the amount of data is large, there will be too many nodes growing at one time, which will occupy a relatively large memory space.

The concept of the BestFS algorithm is the same as Algorithm 4. The difference is that in the BestFS function, we change the queue to a priority queue by using a min-heap data structure, where the priority order is sorted using the calculated lower bound, instead of using the FIFO order. The smaller the lower bound is, the higher the priority. When we use a heap to pop or push values, we will perform the function of heapify at the same time to ensure the heap is in the form of a min-heap. Heapify is the process of creating a heap data structure from a binary tree. Similarly, before each element is pushed into the heap,

we use `LowerBound()` and `CheckProperty()` to check whether the lower bound is smaller than the upper bound and whether it satisfies the property.

5. Ant Colony Optimization (ACO)

Ant colony optimization (ACO) was proposed by Dorigo et al. [23] and Dorigo [24]. It is a meta-heuristic algorithm based on probabilistic techniques and populations. ACO is inspired by the foraging behavior of ants, where the probability of an ant choosing a path is proportional to the pheromone concentration on the path, that is, a large number of ant colonies will give positive feedback. When ants are looking for food, they constantly modify the original path through pheromones and, finally, find the best path. Initially, Ant System (AS) was used to solve the well-known traveling salesman problem (TSP). Later, many ACO variants were produced to solve different hard combinatorial optimization problems, such as assignment problems, scheduling problems, or vehicle routing problems. In recent years, some researchers have focused on applying the ACO algorithm to multi-objective problems and dynamic or stochastic problems. In ant colony optimization, each ant constructs its foraging path (solution) node by node. When determining the next node to move on, we can use dominance properties and exclusion information to rule out the nodes that are not promising. In comparison with other meta-heuristics, this feature may save the time required for handling infeasible or inferior solutions. The pseudo-code of ACO that we adopt is shown in Algorithm 5.

Algorithm 5: Ant Colony Optimization

```
Function ACO():
    initialize the ACO parameters;
    while stopping criteria is not met do
        foreach ants in population do
            generate the first job randomly;
            foreach unselected job do
                choose next job by the transition rule;
            update local pheromone;
        LocalSearch(sequence_lbest);
        update pheromone based on the best solution.
```

State transition rule: We treat each job as a node in the graph and all nodes are connected. To choose the next edge, the ant will consider the visibility of each edge available from its current location, as well as the pheromones. The formula for calculating the visibility value is given by $\eta_{ij} = \frac{1}{\sum\limits_{i \to j} sp_i + p_j}$, where η_{ij} is the visibility value from node i to node j defined as the inverse of the processing time of job j plus its unfinished setup operations. Then, we will calculate the probability of each feasible path; the probability formula is given as $p_{ij}^k = \frac{\tau_{ij}^{\alpha} * \eta_{ij}^{\beta}}{\sum\limits_{k \in unselected_i} \tau_{ik}^{\alpha} * \eta_{ik}^{\beta}}$, where τ_{ij} is the pheromone on the edge from node i to node j, $\alpha \geq 0$ is a parameter for controlling the influence of the pheromone, and $\beta \geq 0$ is a parameter for controlling the influence of invisibility. The next node is determined by a roulette wheel selection.

Pheromone update rule: When all ants have found their solutions, the pheromone trails are updated. The formula for updating the pheromones is defined as $\tau_{ij} = (1 - \rho) * \tau_{ij} + \triangle \tau_{ij}^k$, where ρ is the pheromone evaporation rate, and $\triangle \tau_{ij}^k$, the incremental of the pheromone from node i to node j by the kth ant, is $\tau_{ij}^k = \frac{Q}{\Sigma C_k}$ if the ant k traverses $edge_{i,j}$; 0, otherwise, where ΣC_k is the total completion time in the solution of the kth ant, and Q is a constant.

Stopping criterion: We set a time limit of 1800 s for the ACO execution. Once the course reaches the time limit, the ACO algorithm will stop and report the incumbent best solution.

To close the discussion of ACO features, we note that local search algorithms can improve on the ACO solution at each iteration and make the result closer to the global optimal solution. At the end of each ACO generation, we deploy a 2-OPT local search procedure to the best solution of each generation so as to probabilistically escape the incumbent solution away from the local optimum.

6. Computational Experiments

In this section, we generate test data for appraising the proposed methods. The solution algorithms were coded in Python, and the integer programming models are implemented on Gurobi 9.1.2 interfaced with Python API. The experiments were performed on a desktop computer with Intel Core(TM) i7-8700K CPU at 3.70GHz with 32.0 GB RAM. The operating system is Microsoft Windows 10. We will describe the data generation design and parameter settings in detail and discuss the experimental results.

6.1. Data Generation Scheme

In the experiments, datasets were generated according to the following rules, and all parameters are integers:

1. Six different numbers of jobs $n \in \{5, 10, 20, 30, 40, 50\}$ and different numbers of setup operations $m \in \{4, 8, 18, 25, 35, 45\}$.
2. A binary support relation array $\mathcal{R}$ of a size $n * m$ is randomly generated. If (s_i, t_j) belongs to $\mathcal{R}$, denoted by $r_{ij} = 1$, then job t_j cannot start unless setup s_i is completed. The probability for $r_{ij} = 1$ is set to be 0.5, i.e., if a generated random number ≤ 0.5, then $r_{ij} = 1$. Note that when $r_{ij} = 1$ for all i and j, the problem can be solved by simply arranging the job in the shortest processing time (SPT) order.
3. The processing times of jobs p_j were generated from the uniform distribution $[1, 10]$.
4. The processing times of setups sp_i were generated from the uniform distribution $[1, 5]$.
5. For each job number, three independent instances were generated. In total, 18 datasets will be tested, as shown in Table 1.

Table 1. Categories of datasets.

Datasets	*n*	*m*
$\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$	5	4
$\mathcal{D}_4, \mathcal{D}_5, \mathcal{D}_6$	10	8
$\mathcal{D}_7, \mathcal{D}_8, \mathcal{D}_9$	20	18
$\mathcal{D}_{10}, \mathcal{D}_{11}, \mathcal{D}_{12}$	30	25
$\mathcal{D}_{13}, \mathcal{D}_{14}, \mathcal{D}_{15}$	40	35
$\mathcal{D}_{16}, \mathcal{D}_{17}, \mathcal{D}_{18}$	50	45

6.2. Results of Integer Programming Models

In the experiments of the integer programming models, we ran two integer programming models on the dataset with a time limit of 1800 s. The results are shown in Table 2. If an IP model did not complete its execution of a dataset in 1800 s, its run time is denoted as "−". In the table, the *gap* column indicates the relative difference between the feasible solution found upon termination and the best proven lower bound. The gap values were in the output of Gurobi. The gap value is defined as: $gap(\%) = \frac{|ObjBound - ObjVal|}{|ObjVal|} \times 100\%$, where *ObjBound* and *ObjVal* are a lower bound and the incumbent solution objective, respectively. When the gap is zero we have demonstrated optimality. The column *best solution* represents the best result of all our proposed methods on the same dataset.

When n_k is 10, the sequence-based IP takes more than 1800 s, even though both methods can obtain the optimal solution. The position-based IP takes less time and ends up with a gap of 0%. When n_k is greater than or equal to 20, neither model can find the optimal solution within 1800 s, but there are still some solutions that can find the same solution as the best solution, such as $\mathcal{D}_{17}$ of the position-based IP and $\mathcal{D}_{13}$ of the sequence-based IP. As

n_k increases, the gap of the position-based IP will be greater than that of the sequence-based IP. However, when we compare it with the best solution, even if the objective value is the same as the best solution, the gap value is still very large, such as the position-based IP of $\mathcal{D}_{17}$ and the sequence-based IP of $\mathcal{D}_7$ to $\mathcal{D}_9$. It means that their lower bounds are not tight, i.e., they have a significant deviation from the final feasible solution.

Table 2. Results of different IP models.

n_k	Datasets	Position-Based			Sequence-Based			Best Solution
		obj.	Time	gap	obj.	Time	gap	
5	$\mathcal{D}_1$	101	0.09	0%	101	0.10	0%	101
	$\mathcal{D}_2$	128	0.09	0%	128	0.10	0%	128
	$\mathcal{D}_3$	122	0.08	0%	122	0.10	0%	122
10	$\mathcal{D}_4$	461	246.11	0%	461	-	19%	461
	$\mathcal{D}_5$	423	46.10	0%	423	-	21%	423
	$\mathcal{D}_6$	457	355.95	0%	457	-	25%	457
20	$\mathcal{D}_7$	2055	-	44%	2052	-	53%	2052
	$\mathcal{D}_8$	1822	-	34%	1820	-	58%	1820
	$\mathcal{D}_9$	1670	-	46%	1662	-	57%	1662
30	$\mathcal{D}_{10}$	3609	-	50%	3602	-	61%	3597
	$\mathcal{D}_{11}$	3985	-	50%	4007	-	62%	3985
	$\mathcal{D}_{12}$	4448	-	62%	4469	-	66%	4424
40	$\mathcal{D}_{13}$	8196	-	72%	8168	-	66%	8168
	$\mathcal{D}_{14}$	7395	-	68%	7402	-	67%	7390
	$\mathcal{D}_{15}$	7975	-	70%	7935	-	66%	7935
50	$\mathcal{D}_{16}$	12,882	-	76%	12,963	-	66%	12,880
	$\mathcal{D}_{17}$	12,305	-	73%	12,374	-	67%	12,305
	$\mathcal{D}_{18}$	10,891	-	72%	10,953	-	66%	10,871

6.3. Results of Branch-and-Bound Algorithm

Table 3 shows the results of the branch-and-bound algorithm with three different tree traversal methods. We set the time limit to 1800 s. In this table, the column *node_cnt* represents the number of visited nodes. The *dev* column is an abbreviation for deviation, expressed as a percentage of the difference between the objective value and the best solution. The calculation formula is as $dev(\%) = \frac{(obj - best\ solution)}{best\ solution} \times 100\%$.

When n_k is less than 20, DFS and BestFS successfully find the optimal solutions, but their execution times and the number of visited nodes of BFS are much larger than others. Even if n_k is 20, BFS cannot find the optimal solution within the time limit. In addition, we can see that the execution time of BestFS is faster than that of DFS for a small number of jobs. When n_k is greater than or equal to 30, the three methods fail to find the optimal solution within the time limit. The number of visited nodes and the deviation of DFS are clearly lower than those of BFS and BestFS. The results indicate that DFS is more efficient than BFS and BestFS because the DFS algorithm is not a layer-order traversal but will backtrack after finding the solution. Therefore, BFS and BestFS may not be able to find any feasible solution within the time limit. To sum up, the performance of DFS is better than those of BFS and BestFS, so we will analyze the experimental results of DFS in detail in the next section.

Table 3. Results of different tree traversal methods.

n_k	Datasets	DFS				BFS				BestFS			
		obj.	**node_cnt**	**Time**	**dev**	**obj.**	**node_cnt**	**Time**	**dev**	**obj.**	**node_cnt**	**Time**	**dev**
5	$\mathcal{D}_1$	101	25	0.00	0.00	101	67	0.00	0.00	101	16	0.00	0.00
	$\mathcal{D}_2$	128	22	0.00	0.00	128	44	0.00	0.00	128	12	0.00	0.00
	$\mathcal{D}_3$	122	27	0.00	0.00	122	68	0.01	0.00	122	20	0.00	0.00
10	$\mathcal{D}_4$	461	203	0.06	0.00	461	4141	0.39	0.00	461	283	0.04	0.00
	$\mathcal{D}_5$	423	246	0.04	0.00	423	2518	0.27	0.00	423	128	0.02	0.00
	$\mathcal{D}_6$	457	1055	0.25	0.00	457	34,646	2.57	0.00	457	1788	0.18	0.00
20	$\mathcal{D}_7$	2052	34,798	69.07	0.00	7839	1,940,323	-	2.82	2052	33,393	22.62	0.00
	$\mathcal{D}_8$	1820	44,844	77.49	0.00	6651	2,081,103	-	2.65	1820	44,166	22.03	0.00
	$\mathcal{D}_9$	1662	201,418	314.67	0.00	6211	2,078,269	-	2.74	1662	172,094	96.32	0.00
30	$\mathcal{D}_{10}$	3605	440,015	-	0.00	18,533	1,482,922	-	4.15	18,533	1,903,404	-	4.15
	$\mathcal{D}_{11}$	4053	385,433	-	0.02	22,005	1,083,532	-	4.52	22,055	1,064,974	-	4.53
	$\mathcal{D}_{12}$	4470	392,869	-	0.01	22,990	1,165,534	-	4.20	22,990	1,423,640	-	4.20
40	$\mathcal{D}_{13}$	8357	189,796	-	0.02	51,610	846,721	-	5.32	51,610	769,918	-	5.32
	$\mathcal{D}_{14}$	7564	176,723	-	0.02	44,779	873,638	-	5.06	44,779	1,112,645	-	5.06
	$\mathcal{D}_{15}$	8074	185,522	-	0.02	49,102	888,079	-	5.19	49,102	1,094,652	-	5.19
50	$\mathcal{D}_{16}$	13,007	74,873	-	0.01	106,541	856,905	-	7.27	106,541	954,605	-	7.27
	$\mathcal{D}_{17}$	12,527	84,980	-	0.02	97,383	933,536	-	6.91	97,383	1,047,559	-	6.91
	$\mathcal{D}_{18}$	11,255	90,908	-	0.04	87,545	1,055,329	-	7.05	87,545	1,181,533	-	7.05

6.4. Results of DFS Algorithm

In the experiment, we compare three different cases, including the original DFS algorithm, DFS with the lower bound, and DFS with the dominance property. Table 4 shows the experimental results of the different cases and also compares their objective values $(obj.)$, numbers of visited nodes $(node\_cnt)$, execution times $(time)$, and deviations (dev) from the best solution.

We can find that when the lower bound and properties are incorporated into DFS, the number of visited nodes is significantly reduced. Since this method will cut off unhelpful branches, it can also speed up the traversal, making it easier to find better solutions. Even when n_k is greater than or equal to 30, none of the three cases can find the best solution within the time limit. However, compared with the original DFS, DFS with a lower bound and DFS with a dominance property attained smaller deviations, indicating the capability of finding solutions closer to the best solution.

Table 4. Results of DFS algorithm.

n_k	Datasets	DFS				DFS + LB				DFS + LB + Property			
		obj.	**node_cnt**	**Time**	**dev**	**obj.**	**node_cnt**	**Time**	**dev**	**obj.**	**node_cnt**	**Time**	**dev**
5	$\mathcal{D}_1$	101	325	0.00	0.00	101	28	0.00	0.00	101	25	0.00	0.00
	$\mathcal{D}_2$	128	325	0.00	0.00	128	48	0.00	0.00	128	22	0.00	0.00
	$\mathcal{D}_3$	122	325	0.00	0.00	122	27	0.00	0.00	122	27	0.00	0.00
10	$\mathcal{D}_4$	461	9,864,100	173.53	0.00	461	614	0.11	0.00	461	203	0.07	0.00
	$\mathcal{D}_5$	423	9,864,100	174.60	0.00	423	626	0.09	0.00	423	246	0.05	0.00
	$\mathcal{D}_6$	457	9,864,100	182.09	0.00	457	2381	0.51	0.00	457	1055	0.25	0.00
20	$\mathcal{D}_7$	2174	27,177,572	-	0.06	2052	135,187	230.35	0.00	2052	34,798	69.07	0.00
	$\mathcal{D}_8$	2145	28,810,807	-	0.18	1820	287,087	429.71	0.00	1820	44,844	77.49	0.00
	$\mathcal{D}_9$	1904	29,472,356	-	0.15	1662	1,006,390	1405.52	0.00	1662	201,418	314.67	0.00
30	$\mathcal{D}_{10}$	4229	14,013,551	-	0.18	3679	426,340	-	0.02	3605	440,015	-	0.00
	$\mathcal{D}_{11}$	4757	13,668,237	-	0.19	4211	462,562	-	0.06	4053	385,433	-	0.02
	$\mathcal{D}_{12}$	5087	14,077,620	-	0.15	4530	443,418	-	0.02	4470	392,869	-	0.01

Table 4. *Cont.*

n_k	Datasets	DFS				DFS + LB				DFS + LB + Property			
		obj.	node_cnt	Time	dev	obj.	node_cnt	Time	dev	obj.	node_cnt	Time	dev
40	$\mathcal{D}_{13}$	9557	7,184,349	-	0.17	8405	169,550	-	0.03	8357	189,796	-	0.02
	$\mathcal{D}_{14}$	8505	7,327,087	-	0.15	7645	210,938	-	0.03	7564	176,723	-	0.02
	$\mathcal{D}_{15}$	9131	7,111,936	-	0.15	8250	210,891	-	0.04	8074	185,522	-	0.02
50	$\mathcal{D}_{16}$	14,629	4,136,372	-	0.14	13,196	104,445	-	0.02	13,007	74,873	-	0.01
	$\mathcal{D}_{17}$	14,272	4,189,689	-	0.16	12,911	119,416	-	0.05	12,527	84,980	-	0.02
	$\mathcal{D}_{18}$	12,901	4,254,099	-	0.19	11,522	114,774	-	0.06	11,255	90,908	-	0.04

6.5. Results of ACO Algorithm

In this section, we performed the ACO algorithm on the 18 datasets and set the time limit to 1800 s. Tables 5–7 summarize the results of the three branch-and-bound algorithms with ACO upper bounds. The results include objective values $(obj.)$ and deviation (dev) of the ACO. The execution time of the ACO algorithm is much shorter than that of the branch-and-bound algorithm. In addition, we will compare the objective value $(obj.)$, the number of visited nodes $(node\_cnt)$, and the execution times, $(time)$, of the original algorithm and the algorithm with ACO as the upper bound. The ACO parameters used in the experiments are shown as follows: $generation = 300$; $population = 20$; $\alpha = 3$; $\beta = 1$; and $\rho = 0.1$.

As can be seen from the experimental table, the deviation of the ACO is small and an even better solution can be found than IP models within 1800 s. Therefore, we can use the ACO as the initial value of the upper bound (ub) to speed up the tree traversal time.

When the branch-and-bound algorithm is executing with the test $lb < ub$, the ACO can make ub smaller, cutting more unnecessary branches. According to the tables, when n_k is less than or equal to 20, the algorithm with an upper bound finds the best solution in a shorter time and visits fewer nodes; especially for the ACO in BFS, this is more obvious. As the value of n_k becomes larger, it increases the probability of the algorithm finding the best solution within the same time limit. In summary, using the ACO solution as an upper bound can make the branch-and-bound algorithm perform better.

Table 5. Results of DFS with ACO upper bounds.

n_k	Datasets	DFS			ACO		DFS + ACO		
		obj.	node_cnt	Time	obj.	dev	obj.	node_cnt	Time
5	$\mathcal{D}_1$	101	25	0.00	101	0.00	101	5	0.00
	$\mathcal{D}_2$	128	22	0.00	128	0.00	128	7	0.00
	$\mathcal{D}_3$	122	27	0.00	122	0.00	122	4	0.00
10	$\mathcal{D}_4$	461	203	0.06	461	0.00	461	112	0.06
	$\mathcal{D}_5$	423	246	0.04	427	0.01	423	75	0.04
	$\mathcal{D}_6$	457	1055	0.25	464	0.02	457	954	0.25
20	$\mathcal{D}_7$	2052	34,798	69.07	2058	0.00	2052	32,861	64.83
	$\mathcal{D}_8$	1820	44,844	77.49	1868	0.03	1820	43,257	72.69
	$\mathcal{D}_9$	1662	201,418	314.67	1668	0.00	1662	157,071	243.59
30	$\mathcal{D}_{10}$	3605	440,015	-	3616	0.01	3597	433,789	-
	$\mathcal{D}_{11}$	4053	385,433	-	3986	0.00	3985	373,573	-
	$\mathcal{D}_{12}$	4470	392,869	-	4424	0.00	4424	357,690	-
40	$\mathcal{D}_{13}$	8357	189,796	-	8282	0.01	8282	159,727	-
	$\mathcal{D}_{14}$	7564	176,723	-	7390	0.00	7390	131,500	-
	$\mathcal{D}_{15}$	8074	185,522	-	7967	0.00	7967	145,476	-
50	$\mathcal{D}_{16}$	13,007	74873	-	12,880	0.00	12,880	69,077	-
	$\mathcal{D}_{17}$	12,527	84,980	-	12,587	0.02	12,527	83,440	-
	$\mathcal{D}_{18}$	11,255	90,908	-	10,871	0.00	10,871	71,311	-

Table 6. Results of BFS with ACO upper bounds.

n_k	Datasets	BFS			ACO		BFS + ACO		
		obj.	node_cnt	Time	obj.	Deviation	obj.	node_cnt	Time
5	$\mathcal{D}_1$	101	67	0.00	101	0.00	101	4	0.00
	$\mathcal{D}_2$	128	44	0.00	128	0.00	128	3	0.00
	$\mathcal{D}_3$	122	68	0.01	122	0.00	122	4	0.00
10	$\mathcal{D}_4$	461	4141	0.39	461	0.00	461	96	0.04
	$\mathcal{D}_5$	423	2518	0.27	427	0.01	423	118	0.03
	$\mathcal{D}_6$	457	34,646	2.57	464	0.02	457	2578	0.40
20	$\mathcal{D}_7$	7839	1,940,323	-	2058	0.00	2052	43,231	33.67
	$\mathcal{D}_8$	6651	2,081,103	-	1868	0.03	1868	3,185,673	-
	$\mathcal{D}_9$	6211	2,078,269	-	1668	0.00	1662	79,117	124.83
30	$\mathcal{D}_{10}$	18,533	1,482,922	-	3616	0.01	3616	782,158	-
	$\mathcal{D}_{11}$	22,005	1,083,532	-	3986	0.00	3986	394,009	-
	$\mathcal{D}_{12}$	22,990	1,165,534	-	4424	0.00	4424	485,726	-
40	$\mathcal{D}_{13}$	51,610	846,721	-	8282	0.01	8282	649,800	-
	$\mathcal{D}_{14}$	44,779	873,638	-	7390	0.00	7390	532,584	-
	$\mathcal{D}_{15}$	49,102	888,079	-	7967	0.00	7967	702,847	-
50	$\mathcal{D}_{16}$	106,541	856,905	-	12,880	0.00	12,880	682,047	-
	$\mathcal{D}_{17}$	97,383	933,536	-	12,587	0.02	12,587	927,886	-
	$\mathcal{D}_{18}$	87,545	1,055,329	-	10,871	0.00	10,871	820,528	-

Table 7. Results of BestFS with ACO upper bounds.

n_k	Datasets	BestFS			ACO		BestFS + ACO		
		obj.	node_cnt	Time	obj.	Deviation	obj.	node_cnt	Time
5	$\mathcal{D}_1$	101	16	0.00	101	0.00	101	4	0.00
	$\mathcal{D}_2$	128	12	0.00	128	0.00	128	3	0.00
	$\mathcal{D}_3$	122	20	0.00	122	0.00	122	4	0.00
10	$\mathcal{D}_4$	461	283	0.04	461	0.00	461	96	0.04
	$\mathcal{D}_5$	423	128	0.02	427	0.01	423	61	0.02
	$\mathcal{D}_6$	457	1788	0.18	464	0.02	457	1333	0.18
20	$\mathcal{D}_7$	2052	33,393	22.62	2058	0.00	2052	13,380	24.48
	$\mathcal{D}_8$	1820	44,166	22.03	1868	0.03	1820	39,031	25.69
	$\mathcal{D}_9$	1662	172,094	96.32	1668	0.00	1662	71,082	100.10
30	$\mathcal{D}_{10}$	18,533	1,903,404	-	3616	0.01	3616	1,350,098	-
	$\mathcal{D}_{11}$	22,055	1,064,974	-	3986	0.00	3986	518,619	-
	$\mathcal{D}_{12}$	22,990	1,423,640	-	4424	0.00	4424	648,245	-
40	$\mathcal{D}_{13}$	51,610	769,918	-	8282	0.01	8282	790,216	-
	$\mathcal{D}_{14}$	44,779	1,112,645	-	7390	0.00	7390	971,168	-
	$\mathcal{D}_{15}$	49,102	1,094,652	-	7967	0.00	7967	1,087,057	-
50	$\mathcal{D}_{16}$	106,541	954,605	-	12,880	0.00	12,880	888,985	-
	$\mathcal{D}_{17}$	97,383	1,047,559	-	12,587	0.02	12,587	1,078,245	-
	$\mathcal{D}_{18}$	87,545	1,181,533	-	10,871	0.00	10,871	1,181,376	-

To summarize the computational study, we note that the two proposed integer programming approaches and the branch-and-bound algorithm, aimed at solving the problem to optimality, can complete their execution courses for 20 jobs or less. For larger instances, these exact two approaches become inferior. When reaching the specified time limit, the reported solutions are not favorable. Another observation is about the three traversal strategies. DFS has its advantages in its easy implementations (by straightforward recursions) and minimum memory requirement. The BFS and BestFS strategies are known to show

their significance in maintaining acquired information about the quality of the unexplored nodes in a priority queue. On the other hand, they suffer from the memory space and heap manipulation work for the unexplored nodes. BFS and BestFS would be preferred when a larger memory is available and advanced data structure manipulations are available.

7. Conclusions and Future Works

In this paper, we studied the scheduling problem with shared common setups of the minimum total completion time. We proposed two integer programming models and the branch-and-bound algorithm, which incorporates three tree traversal strategies and the initial solutions yielded from an ACO algorithm. A computational study shows that the position-based IP outperforms the sequence-based one when the problem size is smaller. As the problem grows larger, the gap values for the sequence-based IP are smaller than those of the position-based IP. Similar to the branch-and-bound algorithm, the DFS performs best, regardless of whether lower bounds and other properties are used or not. Finally, we also observed that using ACO to provide an initial upper bound indeed speeds up the execution course of the branch-and-bound algorithm.

For future research, developing tighter lower bounds and upper bounds could lead to better performance. More properties can be found to help the branch-and-bound algorithm curtail non-promising branches. For integer programming models, tighter constraints can be proposed to reduce the execution time and optimality gaps to reflect a real-world circumstance in which multiple machines or servers are available for a software test project. In this generalized scenario, a setup could be performed on several machines if the jobs that it supports are assigned to distinct machines.

Author Contributions: Conceptualization, M.-T.C. and B.M.T.L.; methodology, M.-T.C. and B.M.T.L.; software, M.-T.C.; formal analysis, M.-T.C. and B.M.T.L.; writing—original draft preparation, M.-T.C. and B.M.T.L.; writing, M.-T.C. and B.M.T.L.; supervision, M.-T.C. and B.M.T.L.; project administration, B.M.T.L.; funding acquisition, B.M.T.L. All authors have read and agreed to the published version of the manuscript.

Funding: Chao and Lin were partially supported by the Ministry of Science and Technology of Taiwan under the grant MOST-110-2221-E-A49-118.

Data Availability Statement: The datasets analyzed in this study are be available upon request.

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

References

1. Leung, J.Y.T. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*; CRC Press: Boca Raton, FL, USA, 2004.
2. Pinedo, M. *Scheduling*; Springer: Berlin/Heidelberg, Germany, 2016.
3. Kononov, A.V.; Lin, B.M.T.; Fang, K.T. Single-machine scheduling with supporting tasks. *Discret. Optim.* **2015**, *17*, 69–79. [CrossRef]
4. Graham, R.; Lawler, E.L.; Lenstra, J.K.; Kan, A.R. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discret. Math.* **1979**, *5*, 287–326.
5. Brucker, P. *Scheduling Algorithms*; Springer: Berlin/Heidelberg, Germany, 2013.
6. Baker, K.E. Single Machine Sequencing with Weighting Factors and Precedence Constraints. Unpublished papers, 1971.
7. Adolphson, D.; Hu, T.C. Optimal linear ordering. *Siam J. Appl. Math.* **1973**, *25*, 403–423. [CrossRef]
8. Lawler, E.L. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Ann. Discret. Math.* **1978**, *2*, 75–90.
9. Hassin, R.; Levin, A. An approximation algorithm for the minimum latency set cover problem. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3669, pp. 726–733.
10. Shafransky, Y.M.; Strusevich, V.A. The open shop scheduling problem with a given sequence of jobs on one machine. *Nav. Res. Logist.* **1998**, *41*, 705–731. [CrossRef]
11. Hwang, F.J.; Kovalyov, M.Y.; Lin, B.M.T. Scheduling for fabrication and assembly in a two-machine flowshop with a fixed job sequence. *Ann. Oper. Res.* **2014**, *27*, 263–279. [CrossRef]
12. Cheng, T.C.E.; Kravchenko, S.A.; Lin, B.M.T. Server scheduling on parallel dedicated machines with fixed job sequences. *Nav. Res. Logist.* **2019**, *66*, 321–332. [CrossRef]

13. Brucker, P.; Jurisch, B.; Sievers, B. A branch and bound algorithm for the job-shop scheduling problem. *Discret. Appl. Math.* **1994**, *49*, 107–127. [CrossRef]
14. Hadjar, A.; Marcotte, O.; Soumis, F. A branch-and-cut algorithm for the multiple sepot Vehicle Scheduling Problem.*Oper. Res.* **2006**, *54*, 130–149. [CrossRef]
15. Kunhare, N.; Tiwari, R.; Dhar, J. Particle swarm optimization and feature selection for intrusion detection system. *Sādhanā* **2020**, *45*, 109. [CrossRef]
16. Kunhare, N.; Tiwari, R.; Dhar, J. Intrusion detection system using hybrid classifiers with meta-heuristic algorithms for the optimization and feature selection by genetic algorithms. *Comput. Ind. Eng.* **2022**, *103*, 108383. [CrossRef]
17. Luo, L.; Zhang, Z.; Yin, Y. Simulated annealing and genetic algorithm based method for a bi-level seru loading problem with worker assignment in seru production systems. *J. Ind. Manag. Optim.* **2021**, *17*, 779–803. [CrossRef]
18. Ansari, Z.N.; Daxini, S.D. A state-of-the-art review on meta-heuristics application in remanufacturing. *Arch. Comput. Methods Eng.* **2022**, *29*, 427–470. [CrossRef]
19. Rachih, H.; Mhada, F.Z.; Chiheb, R. Meta-heuristics for reverse logistics: A literature review and perspectives. *Comput. Ind. Eng.* **2019**, *127*, 45–62. [CrossRef]
20. Blum, C.; Sampels, M. An ant colony optimization algorithm for shop scheduling problems. *J. Math. Model. Algorithms* **2004**, *3*, 285–308. [CrossRef]
21. Yang, J.; Shi, X.; Marchese, M.; Liang, Y. An ant colony optimization method for generalized TSP problem. *Prog. Nat. Sci.* **2008**, *18*, 1417–1422. [CrossRef]
22. Xiang, W.; Yin, J.; Lim, G. An ant colony optimization approach for solving an operating room surgery scheduling problem. *Comput. Ind. Eng.* **2015**, *85*, 335–345. [CrossRef]
23. Dorigo, M.; Maniezzo, V.; Colorni, A. *Positive Feedback as a Search Strategy*; Technical Report 91–016; Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1991.
24. Dorigo, M. Optimization, Learning and Natural Algorithms. Ph.D. Thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1992. (In Italian)