# A Two-Level Metaheuristic Algorithm for the Job-Shop Scheduling Problem

Pisut Pongchairerks

Industrial Engineering Program, Faculty of Engineering, Thai-Nichi Institute of Technology, Bangkok 10250, Thailand

Correspondence should be addressed to Pisut Pongchairerks; pisut@tni.ac.th

Received 2 November 2018; Revised 10 January 2019; Accepted 11 February 2019; Published 6 March 2019

Academic Editor: Chongyang Liu

Copyright © 2019 Pisut Pongchairerks. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This paper proposes a novel two-level metaheuristic algorithm, consisting of an upper-level algorithm and a lower-level algorithm, for the job-shop scheduling problem (JSP). The upper-level algorithm is a novel population-based algorithm developed to be a parameter controller for the lower-level algorithm, while the lower-level algorithm is a local search algorithm searching for an optimal schedule in the solution space of parameterized-active schedules. The lower-level algorithm’s parameters controlled by the upper-level algorithm consist of the maximum allowed length of idle time, the scheduling direction, the perturbation method to generate an initial solution, and the neighborhood structure. The proposed two-level metaheuristic algorithm, as the combination of the upper-level algorithm and the lower-level algorithm, thus can adapt itself for every single JSP instance.

1. Introduction

Scheduling problems generally involve assigning jobs to machines at particular times. This paper focuses on the job-shop scheduling problem with minimizing the total length of the schedule (JSP), which is one of the hard-to-solve scheduling problems. The JSP is an NP-hard optimization problem [1, 2] well known in both academic and practical areas. To deal with the JSP and related problems, many approximation algorithms have been developed based on metaheuristic algorithms [3–8]. In addition, some methods such as [9] have been presented for the purpose of reducing the solution space of the JSP.

To solve the JSP, this paper aims at developing the two-level metaheuristic algorithm in which the upper-level algorithm controls the lower-level algorithm’s input parameters, and the lower-level algorithm acts as a local search algorithm to search for an optimal schedule. The purpose of the upper-level algorithm is to iteratively adapt the input-parameter values of the lower-level algorithm, so that the lower-level algorithm will be fitted well for every single instance. In general, a mechanism of controlling a lower-level algorithm’s input parameters by an upper-level algorithm is classified as adaptive parameter control based on the definition given by [10]. The concept of using a metaheuristic algorithm to control the input parameters of another metaheuristic algorithm, known as a meta-evolutionary algorithm, has been applied in many different researches such as [11–15].

In detail, this paper will hereafter call the upper-level algorithm and lower-level algorithm in the proposed two-level metaheuristic algorithm as UPLA and LOLA, respectively. LOLA is a local search algorithm exploring in a solution space of parameterized-active schedules (also known as hybrid schedules), where each parameterized-active schedule [16, 17] is decoded from an operation-based permutation [18–20]. In LOLA, there are four input parameters which need to be assigned their values before LOLA is executed, i.e., the maximum allowed length of idle time of constructing parameterized-active schedules, the scheduling direction of constructing schedules, the perturbation method to generate an initial solution, and the operator combination used to generate a neighborhood structure. The values of these four input parameters can be adjusted and inputted by a human user; however, this duty in the proposed two-level algorithm will belong to UPLA.

UPLA is a population-based metaheuristic algorithm searching in the real-number search space; from this point of view, it is similar to the particle swarm optimization [21], differential evolution [22], firefly [23], cuckoo search [24], and artificial fish-swarm [25, 26] algorithms. Unlike the others,
2 Preliminaries

The job-shop scheduling problem (JSP) comes with a given set of \( n \) jobs \( J_1, J_2, \ldots, J_n \) and a given set of \( m \) machines \( M_1, M_2, \ldots, M_m \). The \( n \) jobs arrive before or at the same time of the schedule’s start time (i.e., time 0), and the \( m \) machines are all available at the schedule’s start time as well. Each job \( J_i \) (\( i = 1, 2, \ldots, n \)) consists of an unchangeable sequence of \( m \) operations \( O_{i1}, O_{i2}, \ldots, O_{in} \). Thus, in order to process the job \( J_i \), the operation \( O_{i1} \) must be finished before the operation \( O_{i2} \) can start, the operation \( O_{i2} \) must be finished before the operation \( O_{i3} \) can start, and so on. Moreover, each operation must be processed on a preassigned machine with a predetermined processing time. Each machine can process at most one operation at a time, and it cannot be interrupted during processing an operation. The objective of the JSP in this paper is to find a feasible schedule of the \( n \) jobs on the \( m \) machines with minimizing makespan. Note that makespan means the total length of the schedule, or the total amount of time required to complete all jobs. Good reviews about the JSP are found in [1, 2, 27].

Schedules in the JSP can be constructed in forward or backward (reverse) directions. A schedule is defined as a forward schedule if every job \( J_i \) (where \( i = 1, 2, \ldots, n \)) in the schedule is done by starting from the first operation \( O_{i1} \) to the last operation \( O_{in} \). On the contrary, a schedule is defined as a backward schedule if every job \( J_i \) (where \( i = 1, 2, \ldots, n \)) in the schedule is done backward by starting from the last operation \( O_{in} \) to the first operation \( O_{i1} \). The backward scheduling is commonly used for the scheduling problem with a due-date criterion. However, it also provides a benefit to the JSP with amakespan criterion because some instances can be solved to optimality by the backward direction more simply than by the forward direction. A backward schedule can simply be constructed by reversing the directions of the precedence constraints of all operations, and then allocating jobs to machines in the same way as constructing a forward schedule; after that, the schedule must be turned back to front in order to make it satisfy the original precedence constraints. The uses of the backward scheduling direction are found in published articles, e.g., [7, 28–31].

All feasible schedules can be classified into three classes, i.e., semiactive schedules, active schedules, and nondelay schedules [16, 19, 32]. A feasible schedule is defined as a semiactive schedule if no operation can be started earlier without changing the operation sequence. A semiactive schedule is defined as an active schedule if no operation can start earlier without delaying another operation or without violating the precedence constraints. Finally, an active schedule is defined as a nondelay schedule if no machine is kept idle for more than a maximum allowed length of idle time. In [16], the maximum allowed length of idle time is controlled by a tunable parameter \( \delta \) \in [0, 1]. At extremes, the GA of [16] with \( \delta = 1 \) will explore in the solution space of active schedules, while the GA with \( \delta = 0 \) will explore in the solution space of nondelay schedules. Because \( \delta \) can be adjusted to be any real number between 0 and 1, this parameter thus controls the algorithm’s solution space between nondelay schedules and active schedules. Some variants of the hybrid scheduler of [16], which are modified for the purpose of simplification, can be found in published articles such as [15, 31, 33, 34].

Since the proper value of \( \delta \) is problem-dependent, the researchers try to control the tunable parameter \( \delta \) in many different ways. Some metaheuristic algorithms such as [16, 33] use \( \delta \) with fixed values. On the other hand, some other metaheuristic algorithms such as [34, 35] adjust the \( \delta \) values during their evolutionary processes; thus, their methods of changing the \( \delta \) values belong to the classification of self-adaptive parameter control based on the definition given by [10]. In addition, some metaheuristic algorithms adjust the \( \delta \) values based on the mathematical functions of another parameter (or other parameters); for example, the local search algorithms of [31] adjust the \( \delta \) value based on the iteration’s index. The concept of a two-level metaheuristic algorithm in which the upper-level algorithm controls the input parameters including \( \delta \) for the lower-level algorithm has been found in [15]. Hereafter, the parameter \( \delta \) will be called acceptable idle-time limit in this paper.
3. The Proposed Lower-Level Algorithm

LOLA, which is the lower-level algorithm in the proposed two-level algorithm, acts as a local search algorithm. It is similar to the other local search algorithms in its basic framework [36–39]; in addition, it generates neighbor solutions based on common swap and insert operators [31, 40, 41]. However, LOLA has three specific features different from general local search algorithms as follows:

(i) LOLA transforms an operation-based permutation [18–20] into a parameterized-active schedule (or hybrid schedule) via Algorithm 1 (see Section 3.1).

(ii) LOLA allows being adjusted in the values of its input parameters, so that LOLA can be best fitted for every single instance. These input parameters are the acceptable idle-time limit, the scheduling direction, the perturbation method, and the neighborhood structure. In this paper, the duties of adjusting the proper parameter values for LOLA belong to UPLA.

(iii) Once LOLA is combined with UPLA, it will use the perturbation method selected by UPLA to generate its initial solution. By the perturbation method supported from UPLA, LOLA is upgraded from a local search algorithm to be an iterated local search algorithm, which can escape from a local optimum. Reviews about iterated local search algorithms can be found in many articles, e.g., [42].

Section 3.1 presents Algorithm 1, the procedure of decoding an operation-based permutation into a parameterized-active schedule used by LOLA. A parameterized-active schedule generated by Algorithm 1 can be a forward parameterized-active schedule (i.e., a parameterized-active schedule constructed in the forward scheduling direction) or a backward parameterized-active schedule (i.e., a parameterized-active schedule constructed in the backward scheduling direction) depending on an UPLA’s selection. Section 3.2 then presents Algorithm 2, the procedure of LOLA.

3.1. Decoding Procedure to Generate Parameterized-Active Schedules Used by LOLA

For an n-job/m-machine JSP instance, an operation-based permutation [18–20] is a permutation with repetition of the numbers 1, 2, . . . , n, where each number occurs m times. As an example, the permutation (3, 2, 2, 1, 3, 1) is an operation-based permutation representing a schedule for a 3-job/2-machine JSP instance. The procedure of transforming an operation-based permutation into a parameterized-active schedule used by LOLA is given in Algorithm 1. As mentioned, a parameterized-active schedule decoded from an operation-based permutation by Algorithm 1 can be constructed in either the forward direction or backward direction as two options.

In brief, Algorithm 1 starts by receiving an operation-based permutation, an acceptable idle-time limit δ ∈ [0, 1), and a scheduling direction in Step 1. If the scheduling direction just received is backward, then Step 1 reverses the order of all members in the operation-based permutation, and Step 2 reverses the precedence relations of all operations of each job. Step 3 then transforms the operation-based permutation into the priority order of all operations. Based on the priority order of operations and the acceptable idle-time limit, a parameterized-active schedule will then be constructed by Steps 4 to 7. In these steps, Algorithm 1 will iteratively choose the highest-priority operation from all schedulable operations which can be started not-later-than σ∗ + δ(φ∗ − σ∗) and add the chosen operation into the partial schedule. Note that σ∗ is the minimum of the earliest possible start times of all schedulable operations, and φ∗ is the minimum of the earliest possible finish times of all schedulable operations. Steps 5 to 7 will be repeated until the schedule is completed. After that, if the scheduling direction received in Step 1 is forward, stop the algorithm; however, if it is backward, Step 8 will modify the schedule given by Step 7 to satisfy the original precedence constraints.

Algorithm 1 can be defined as a generalization of the solution-decoding procedures shown in [15, 31, 33, 34] since it can generate a parameterized-active schedule in either the forward direction or backward direction as the options. Algorithm 1 and its abovementioned variants are modified from the hybrid scheduler of [16] for the purpose of simplification (as found in Steps 5 and 6 of Algorithm 1). Hence, Algorithm 1 may construct a different schedule from the hybrid scheduler of [16], even if they both use an identical δ value on an identical operation-based permutation.

Algorithm 1. It is the procedure of decoding an operation-based permutation into a parameterized-active schedule used by LOLA.

Step 1. Receive an operation-based permutation. Then, receive an acceptable idle-time limit δ ∈ [0, 1) and a scheduling direction (forward or backward) as input-parameter values. If the scheduling direction is backward, then reverse the order of all members in the operation-based permutation. For example, if the received scheduling direction is backward, a permutation (3, 2, 2, 1, 3, 1) will then be changed to (1, 3, 1, 2, 2, 3).

Step 2. If the scheduling direction received in Step 1 is backward, then reverse the precedence relations of all operations of every job in the being-considered JSP instance by using Steps 2.1 and 2.2.

Step 2.1. For each job Ji (where i = 1, 2, . . . , n), the operations Oi1, Oi2, . . . , Oim must be renamed Oim, Oim−1, . . . , Oi1, respectively.

Step 2.2. Assign the precedence relations of all operations of the job Ji by following the operation indices taken from Step 2.1. This means that Oi1 must be finished before Oi2 can start, Oi2 must be finished before Oi3 can start, and so on.

Step 3. Transform the operation-based permutation taken from Step 1 into an order of priorities of all operations as follows: let the jth occurrence of the number i in the permutation, starting from furthest at the left, represent the operation Oij; then, let the order of all
operations in the permutation, starting from furthest at the left, represent the descending order of priorities of the operations. For example, the permutation (3, 2, 1, 3, 1) means priority of $O_{31}$ > priority of $O_{21}$ > priority of $O_{32}$ > priority of $O_{11}$ > priority of $O_{32}$ > priority of $O_{12}$

Step 4. Let $S_t$ be the set of all schedulable operations at stage $t$. Note that a schedulable operation is an as-yet-unscheduled operation whose all preceding operations in its job have already been scheduled. Let $\Phi_t$ be the partial schedule of the $t$ scheduled operations. Thus, $\Phi_0$ is empty, and $S_t$ consists of all $O_{ij}$ operations (where $i = 1, 2, \ldots, n$). In addition, let the earliest possible start times of all $O_{ij}$ operations be equal to the time 0. Now, let $t = 1$

Step 5. Let $s^*$ be the minimum of the earliest possible start times of all schedulable operations in $S_t$, and let $q^*$ be the minimum of the earliest possible finish times of all schedulable operations in $S_t$. (Note that the earliest possible start time of a specific schedulable operation is the maximum between the finished time of its immediate-preceding operation in its job and the earliest available time of its preassigned machine. Then, the earliest possible finish time of a specific schedulable operation is the sum of its earliest possible start time and its predetermined processing time)

Step 6. Let $O_h$ represent the highest-priority operation of all schedulable operations whose earliest possible start times are not greater than $s^* + \delta (q^* - s^*)$ in $S_t$. Then, create $\Phi_t$ by allocating $O_h$ into $\Phi_{t-1}$ on the machine preassigned for processing $O_h$ at the earliest possible start time of $O_h$. After that, create $S_{t+1}$ by deleting $O_h$ from $S_t$; in addition, if $O_h$ has an immediate-successive operation in its job, let the immediate-successive operation of $O_h$ in its job be added to $S_{t+1}$

Step 7. If $t < mn$, then let the value of $t$ be increased by 1, and repeat from Step 5; otherwise, let $\Phi_{mn}$ be a completed parameterized-active schedule, and go to Step 8. (Remember that $mn$ is the number of all operations)

Step 8. If the scheduling direction received in Step 1 is forward, then stop Algorithm 1, and let the schedule $\Phi_{mn}$ taken from Step 7 be the completed forward parameterized-active schedule and also the algorithm's final result; however, if it is backward, then modify $\Phi_{mn}$ to satisfy the original precedence constraints by using Steps 8.1 and 8.2

Step 8.1. Let the operations $O_{mn}, O_{mn-1}, \ldots, O_1$ of each job $J_i$ in the schedule $\Phi_{mn}$ be renamed $O_{1j}, O_{2j}, \ldots, O_{mn}$, respectively

Step 8.2. Turn the schedule modified from Step 8.1 back to front, so that the last-finished operation in the schedule will become the first-started operation, and vice versa; after that, let the schedule be started at the time 0. Then, stop Algorithm 1, and let the schedule $\Phi_{mn}$ modified in this step be the completed backward parameterized-active schedule and also the algorithm's final result

3.2. Procedure of LOLA. As mentioned earlier, the four input parameters of LOLA controlled by UPLA consist of the acceptable idle-time limit, the scheduling direction, the perturbation method for generating an initial operation-based permutation, and the method for generating a neighbor operation-based permutation. The details of these four input parameters of LOLA are given below:

(i) The acceptable idle-time limit $\delta \in [0, 1)$ is defined as a controller of the solution space; for example, the LOLA with $\delta = 0$ will search in the solution space of nondelay schedules

(ii) The scheduling direction $D \in \{\text{forward}, \text{backward}\}$ is a direction of constructing schedules. If $D$ is chosen to be forward, LOLA will generate only forward schedules. On the other hand, if $D$ is chosen to be backward, LOLA will generate only backward schedules

(iii) The perturbation method for generating an initial operation-based permutation $IP \in \{\text{full randomization, partial randomization}\}$ is the perturbation method used to generate an initial permutation for LOLA. If $IP$ is selected to be full randomization, the initial operation-based permutation will be generated by full randomization without using any part of the best-found permutation memorized by UPLA. However, if $IP$ is selected to be partial randomization, the initial operation-based permutation will be generated by using $n$ insert operators (i.e., using the insert operation $n$ times) on the best-found operation-based permutation memorized by UPLA

(iv) The method for generating a neighbor operation-based permutation $NP \in \{2\text{-insert, 1-insert/1-swap, 1-swap/1-insert, 2-swap}\}$ is the adjustable method for modifying the LOLA's current best-found operation-based permutation into a neighbor operation-based permutation. The 2-insert is to use two insert operators. The 1-insert/1-swap is to use an insert operator and then a swap operator. The 1-swap/1-insert is to use a swap operator and then an insert operator. Finally, the 2-swap is to use two swap operators

In this paper, the swap operator is done by randomly selecting two members (of all $mn$ members) from two different positions in the permutation, and then switching the positions of the two selected members. The insert operator is done by randomly selecting two members (of all $mn$ members) from two different positions in the permutation, removing the first-selected member from its old position, and then inserting it into the position in front of the second-selected member. Note that $n$ is the number of all jobs, $m$ is the number of all machines, and $mn$ is thus the number of all operations in the problem's instance. The procedure of LOLA is given in Algorithm 2.
Algorithm 2. It is the procedure of LOLA.

Step 1. Receive the best-found operation-based permutation memorized by UPLA. Then, receive the values of the LOLAs input parameters from UPLA via Steps 1.1 to 1.4

Step 1.1. Receive the acceptable idle-time limit \( \delta \in [0, 1) \)
Step 1.2. Receive the scheduling direction \( D \in \{ \text{forward}, \text{backward} \} \)
Step 1.3. Receive the perturbation method for generating an initial operation-based permutation \( IP \in \{ \text{full randomization, partial randomization} \} \)
Step 1.4. Receive the method for generating a neighbor operation-based permutation \( NP \in \{ 2\text{-insert, 1-insert/1-swap, 1-swap/1-insert, 2-swap} \} \)

Step 2. Let \( P_0 \) represent the LOLAs current best-found operation-based permutation. Generate an initial \( P_0 \) by using the \( IP \) method received in Step 1.3

Step 3. Execute Algorithm 1 by using \( \delta \) from Step 1.1 and \( D \) from Step 1.2 on the permutation \( P_0 \) from Step 2. Then, let \( S_0 \) be equal to the schedule returned from Algorithm 1 executed in this step

Step 4. Execute the local search procedure by Steps 4.1 to 4.5

Step 4.1. Let \( t_L = 0 \)
Step 4.2. Generate a neighbor operation-based permutation \( P_1 \) by modifying from the permutation \( P_0 \) via the \( NP \) method received in Step 1.4
Step 4.3. Execute Algorithm 1 by using \( \delta \) from Step 1.1 and \( D \) from Step 1.2 on the permutation \( P_1 \) from Step 4.2. Then, let \( S_1 \) be equal to the schedule returned from Algorithm 1 executed in this step
Step 4.4. If the makespan of \( S_1 \) is less than the makespan of \( S_0 \), then update \( P_0 \) to be equal to \( P_1 \), update \( S_0 \) to be equal to \( S_1 \), and let \( t_L = 0 \); otherwise, increase the value of \( t_L \) by 1
Step 4.5. If \( t_L < mn(mn - 1) \), then repeat from Step 4.2; otherwise, stop Algorithm 2, and let \( S_0 \) and \( P_0 \) be the final best-found schedule and the final best-found operation-based permutation, respectively, as the LOLAs final results

Remember that there are no parameter-value settings for LOLA required here since the four input parameters of LOLA are controlled by UPLA. As shown in Step 1 of Algorithm 2, the settings of \( \delta, D, IP, \) and \( NP \) are provided by UPLA.

4. The Proposed Upper-Level Algorithm

UPLA is a population-based metaheuristic algorithm exploring in a real-number search space; from this viewpoint, it is similar to the other population-based algorithms [21–26]. However, the procedure of improving its population is different since it is developed for being a parameter controller for LOLA. UPLA starts with a population of the \( N \) combinations of LOLAs input-parameter values, consisting of \( C_1(t), C_2(t), \ldots, C_N(t) \). Let \( C(t) = \{ \delta(t), d(t), ip(t), np(t) \} \) represent the \( i \)th combination of the LOLAs input-parameter values (where \( i = 1, 2, \ldots, N \)) at the \( t \)th iteration. \( \delta(t) \in [0, 1), d(t) \in R, ip(t) \in R, \) and \( np(t) \in R \) in the combination \( C(t) \) represent the acceptable idle-time limit \( \delta \), the scheduling direction \( D \), the perturbation method for generating an initial operation-based permutation \( IP \), and the method for generating a neighbor operation-based permutation \( NP \), respectively, in LOLA. Table 1 shows the translation from \( \delta(t), d(t), ip(t), \) and \( np(t) \) of the combination \( C(t) \) into \( \delta, D, IP, \) and \( NP \) of LOLA. In short, let a combination of LOLAs input-parameter values be called a parameter-value combination.

The performance of \( C(t) \) is simply equal to the makespan returned from the LOLA using the parameter values decoded from \( C(t) \). This means that, between two parameter-value combinations, the better combination is the combination which makes LOLA return lower makespan. Thus, the parameter-value combination will be defined as the best if it makes LOLA return lower makespan. In UPLA, let \( C_{\text{best}} = \{ \delta_{\text{best}}, d_{\text{best}}, ip_{\text{best}}, np_{\text{best}} \} \) represent the best-found parameter-value combination memorized by UPLA. Thus, \( C_{\text{best}} \) will always be updated whenever UPLA can find a parameter-value combination which performs better than the current \( C_{\text{best}} \). In other words, once UPLA finds a \( C(t) \) performing better than the current \( C_{\text{best}} \), this \( C(t) \) will become the new \( C_{\text{best}} \) by \( \{ \delta(t), d(t), ip(t), np(t) \} \) will become the new \( C_{\text{best}} \) by \( \{ \delta_{\text{best}}, d_{\text{best}}, ip_{\text{best}}, np_{\text{best}} \} \).

In the \( t \)th UPLA's iteration, \( \delta(t), d(t), ip(t), \) and \( np(t) \), as the members of \( C(t) \), will each be updated respectively into \( \delta(t + 1), d(t + 1), ip(t + 1), \) and \( np(t + 1) \) of \( C(t + 1) \) by two vectors. The two vectors, used for updating each member's current value in \( C(t) \) to its new value in \( C(t + 1) \), are described in directions and magnitudes as follows. In the vector directions, the first vector's direction is toward the current value in \( C(t) \) to the best-found value in \( C_{\text{best}} \), while the second vector's direction is opposite the first vector's direction. In the vector magnitudes, if the current value in \( C(t) \) and the best-found value in \( C_{\text{best}} \) are different, the first vector's magnitude is a random real number in \([0.00, 0.05]\) while the second vector's magnitude is a random real number in \([0.00, 0.01]\); however, if they are the same, the magnitudes of the two vectors are both random real numbers in \([0.00, 0.01]\). The process of iteratively updating the parameter-value combination given in this paragraph will be found in Steps 4.2 to 4.5 of Algorithm 3.

The mechanism of the UPLAs combined with LOLA is summarized in Figure 1, where -span, \( t \) stands for the makespan of the schedule returned from the LOLA using the input-parameter values decoded from \( C(t) \). The procedure of UPLA is fully presented in Algorithm 3. In Algorithm 3, Steps 1 and 2 initialize the UPLAs population. Although the largest possible range of \( \delta(t) \) is \([0, 1) \) as mentioned earlier, the algorithm generates \( \delta(t) \) initially within \([0.8, 0.9]\) and then restricts its value within \([0.7, 1]\) along the computational process, based on a suggestion from [31]. Step 3 transforms \( C(t) \) into \( \{ \delta(t), d(t), ip(t), np(t) \} \) into \( \delta, D, IP, \) and \( NP \) for LOLA, and this step then executes the LOLA using the input-parameter values transformed from \( C(t) \). After
Algorithm 3. It is the procedure of UPLA.

Step 1. Let \( C_i(t) \equiv \{\delta_i(t), d_i(t), i p_i(t), n p_i(t)\} \) represent the \( i \)th parameter-value combination (where \( i = 1, 2, \ldots, N \)) at the \( t \)th iteration. Let \( C_{best} \equiv \{\delta_{best}, d_{best}, i p_{best}, n p_{best}\} \) represent the best-found parameter-value combination. Set the initial performance of \( C_{best} \) to be equal to an extremely large value.

Step 2. Let \( t = 1 \). Then, initially generate \( C_i(t) \), where \( i = 1, 2, \ldots, N \), as follows: \( \delta_i(t) \sim U[0.8,0.9) \), \( d_i(t) \sim U[0,1) \), \( i p_i(t) \sim U[0,1) \), and \( n p_i(t) \sim U[0,1) \)

Step 3. Evaluate the performance of \( C_i(t) \) and update \( C_{best} \) by Steps 3.1 to 3.5

Step 3.1. Let \( i = 1 \)

Step 3.2. Translate \( \delta_i(t), d_i(t), i p_i(t), \) and \( n p_i(t) \) of \( C_i(t) \) into \( \delta, D, I P, \) and \( N P \) for LOLA by the relationships shown in Table 1. Then, execute the LOLA (Algorithm 2) which uses \( \delta, D, I P, \) and \( N P \) translated in this step.

Step 3.3. Let the performance of \( C_i(t) \) be equal to the makespan of the schedule returned from the LOLA executed in Step 3.2.

Step 3.4. If the performance of \( C_i(t) \) is better (lower) than the performance of \( C_{best} \), then update \( C_{best} \equiv \{\delta_{best}, d_{best}, i p_{best}, n p_{best}\} \) to be equal to \( C_i(t) \equiv \{\delta_i(t), d_i(t), i p_i(t), n p_i(t)\} \), update the performance of \( C_{best} \) to be equal to the performance of \( C_i(t) \), and also set the schedule and the permutation returned from the LOLA executed in Step 3.2 respectively as the best-found schedule and the best-found operation-based permutation memorized by UPLA.

Step 3.5. If \( i < N \), then increase the value of \( i \) by 1, and repeat from Step 3.3; otherwise, go to Step 4.

Step 4. If \( t \mod 25 = 0 \), then randomly generate \( C_i(t+1) \), where \( i = 1, 2, \ldots, N \), as follows: \( \delta_i(t+1) \sim U[0.8,0.9) \), \( d_i(t+1) \sim U[0,1) \), \( i p_i(t+1) \sim U[0,1) \), and \( n p_i(t+1) \sim U[0,1) \); otherwise, generate \( C_i(t+1) \) by Steps 4.1 to 4.6.

Step 4.1. Let \( i = 1 \)

Figure 1: Flowchart of the UPLA combined with LOLA.
Step 4.2. Generate \( u_1, u_2 \sim U(0, 1) \), and then generate \( \delta_i(t + 1) \) as follows:

\[
\delta_i(t + 1) = \begin{cases} 
\delta_i(t) + 0.05u_1 - 0.01u_2 & \text{if } \delta_i(t) < \delta_{\text{best}} \\
\delta_i(t) - 0.05u_1 + 0.01u_2 & \text{if } \delta_i(t) > \delta_{\text{best}} \\
\delta_i(t) + 0.01u_1 - 0.01u_2 & \text{if } \delta_i(t) = \delta_{\text{best}}.
\end{cases}
\]  

(1)

After that, if \( \delta_i(t + 1) < 0.7 \) or \( \delta_i(t + 1) \geq 1.0 \), then regenerate \( \delta_i(t + 1) \sim U(0.8, 0.9) \).

Step 4.3. Generate \( u_1, u_2 \sim U(0, 1) \), and then generate \( d_i(t + 1) \) as follows:

\[
d_i(t + 1) = \begin{cases} 
d_i(t) + 0.05u_1 - 0.01u_2 & \text{if } d_i(t) < d_{\text{best}} \\
d_i(t) - 0.05u_1 + 0.01u_2 & \text{if } d_i(t) > d_{\text{best}} \\
d_i(t) + 0.01u_1 - 0.01u_2 & \text{if } d_i(t) = d_{\text{best}}.
\end{cases}
\]  

(2)

Step 4.4. Generate \( u_1, u_2 \sim U(0, 1) \), and then generate \( i(p_i)(t + 1) \) as follows:

\[
i(p_i)(t + 1) = \begin{cases} 
i(p_i)(t) + 0.05u_1 - 0.01u_2 & \text{if } i(p_i)(t) < i(p_{\text{best}}) \\
i(p_i)(t) - 0.05u_1 + 0.01u_2 & \text{if } i(p_i)(t) > i(p_{\text{best}}) \\
i(p_i)(t) + 0.01u_1 - 0.01u_2 & \text{if } i(p_i)(t) = i(p_{\text{best}}).
\end{cases}
\]  

(3)

Step 4.5. Generate \( u_1, u_2 \sim U(0, 1) \), and then generate \( n(p_i)(t + 1) \) as follows:

\[
n(p_i)(t + 1) = \begin{cases} 
n(p_i)(t) + 0.05u_1 - 0.01u_2 & \text{if } n(p_i)(t) < n(p_{\text{best}}) \\
n(p_i)(t) - 0.05u_1 + 0.01u_2 & \text{if } n(p_i)(t) > n(p_{\text{best}}) \\
n(p_i)(t) + 0.01u_1 - 0.01u_2 & \text{if } n(p_i)(t) = n(p_{\text{best}}).
\end{cases}
\]  

(4)

Step 4.6. If \( i < N \), then increase the value of \( i \) by 1, and repeat from Step 4.2; otherwise, go to Step 5

Step 5. If the stopping criterion is not met, then increase the value of \( t \) by 1, and repeat from Step 3. Otherwise, stop Algorithm 3, and let the best-found schedule memorized by UPLA be the final result of UPLA

5. Performance Evaluation

Among JSP-solving algorithms, the two-level particle swarm optimization algorithm (or the two-level PSO) of [15] is probably the most similar algorithm to the proposed two-level metaheuristic algorithm. The similarity is that both their lower-level algorithms generate parameterized-active schedules by the similar procedures; in addition, both their upper-level algorithms control the same two parameters (i.e., the acceptable idle-time limit and the scheduling direction for constructing schedules) for their lower-level algorithms. However, the two-level PSO [15] is different from the proposed two-level metaheuristic algorithm in that it uses the GLN-PSO’s framework [43] in both its levels. Due to their similarity and difference just mentioned, this paper then selects the two-level PSO [15] to compare with the proposed two-level metaheuristic algorithm in their performances. In this section, UPLA will represent the proposed two-level metaheuristic algorithm as a whole (i.e., the UPLA combined with LOLA) since UPLA must work by having LOLA inside when executed.

This section evaluates the performance of UPLA on 53 well-known JSP benchmark instances. The 53 benchmark instances consist of the ft06, ft10, and ft20 instances from [44], the la01 to la40 instances from [45], and the orb01 to orb10 instances from [46]; these 53 instances can also be found online in [47]. This paper divides the 53 benchmark instances into two sets, i.e., the set of ft-and-la instances and the set of orb instances. Since the results of the two-level PSO on the orb instances do not exist, UPLA will be compared to the GA of [6] in their results on the set of ft-and-la instances. Since the results of the two-level PSO on the orb instances do not exist, UPLA will be compared to the GA of [6] in their results on the set of orb instances.

In the performance comparisons, the results of UPLA are received from the experiment here. The settings of UPLA for the experiment are given as follows:

(i) The population in UPLA consists of 10 combinations of the LOLA’s input-parameter values (i.e., \( N = 10 \))

(ii) The stopping criterion of UPLA is that either the 200th iteration is reached (i.e., \( t = 200 \) is the maximum iteration) or the optimal solution shown in published articles, e.g., [5], is found

(iii) UPLA is coded in C# and executed on an Intel® Core™ i5 CPU processor M580 @ 2.67 GHz with RAM of 6 GB (2.3 GB usable)

(iv) UPLA is executed for 10 runs with different random seed numbers

(v) The directions and magnitudes of the vectors used to generate \( C_i(t + 1) \) are given in Step 4 of Algorithm 3

### Table 2

| Instance | \( N \) | \( t \) | \( u \) | \( 
\)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ft06</td>
<td>10</td>
<td>200</td>
<td>0.5</td>
</tr>
<tr>
<td>ft10</td>
<td>10</td>
<td>200</td>
<td>0.5</td>
</tr>
<tr>
<td>ft20</td>
<td>10</td>
<td>200</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Tables 2 and 3 show the experiment’s results on the ft-and-la instances and the orb instances, respectively. Note that the words solution and solution value in this paper are equivalent to the words schedule and makespan, respectively. The information given by these tables contains the following:

(i) The column Instance provides the name of each instance

(ii) The column Opt provides the optimal solution value (i.e., the optimal schedule’s makespan) of each instance given by the published articles, e.g., [5]
Table 2: Experiment's results on the set of ft-and-la instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Opt</th>
<th>PSO [15]</th>
<th>UPLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ft06</td>
<td>55</td>
<td>55*.00</td>
<td>55*.00</td>
</tr>
<tr>
<td>ft10</td>
<td>930.00</td>
<td>930*.00</td>
<td>930*.00</td>
</tr>
<tr>
<td>ft20</td>
<td>1165.00</td>
<td>1165*.00</td>
<td>1172.062.188</td>
</tr>
<tr>
<td>la01</td>
<td>666.00</td>
<td>666*.00</td>
<td>666.00.1</td>
</tr>
<tr>
<td>la02</td>
<td>655.00</td>
<td>655*.00</td>
<td>655.00.2.2</td>
</tr>
<tr>
<td>la03</td>
<td>597.00</td>
<td>597*.00</td>
<td>597.00.10.13</td>
</tr>
<tr>
<td>la04</td>
<td>590.00</td>
<td>590*.00</td>
<td>590.00.3.4</td>
</tr>
<tr>
<td>la05</td>
<td>593.00</td>
<td>593*.00</td>
<td>593.00.1.1</td>
</tr>
<tr>
<td>la06</td>
<td>926.00</td>
<td>926*.00</td>
<td>926.00.1.4</td>
</tr>
<tr>
<td>la07</td>
<td>863.00</td>
<td>863*.00</td>
<td>863.00.1.5</td>
</tr>
<tr>
<td>la08</td>
<td>958.00</td>
<td>958*.00</td>
<td>958.00.1.4</td>
</tr>
<tr>
<td>la11</td>
<td>1222.00</td>
<td>1222*.00</td>
<td>1222.00.1.12</td>
</tr>
<tr>
<td>la12</td>
<td>1039.00</td>
<td>1039*.00</td>
<td>1039.00.1.13</td>
</tr>
<tr>
<td>la13</td>
<td>1150.00</td>
<td>1150*.00</td>
<td>1150.00.1.12</td>
</tr>
<tr>
<td>la14</td>
<td>1292.00</td>
<td>1292*.00</td>
<td>1292.00.1.12</td>
</tr>
<tr>
<td>la15</td>
<td>1207.00</td>
<td>1207*.00</td>
<td>1207.00.1.17</td>
</tr>
<tr>
<td>la16</td>
<td>945.00</td>
<td>945*.00</td>
<td>945.00.124.1458</td>
</tr>
<tr>
<td>la17</td>
<td>848.00</td>
<td>848*.00</td>
<td>848.00.6.78</td>
</tr>
<tr>
<td>la18</td>
<td>842.00</td>
<td>842*.00</td>
<td>843.076.90</td>
</tr>
<tr>
<td>la19</td>
<td>902.00</td>
<td>907.055.902*.00</td>
<td>903.06.107.1304</td>
</tr>
<tr>
<td>la20</td>
<td>1046.00</td>
<td>1052.057.1058.1.18.200.13505</td>
<td></td>
</tr>
<tr>
<td>la21</td>
<td>927.00</td>
<td>935.086.927*.00</td>
<td>935.083.191.12840</td>
</tr>
<tr>
<td>la22</td>
<td>1032.00</td>
<td>1032*.00</td>
<td>1032.00.2.116</td>
</tr>
<tr>
<td>la23</td>
<td>935.00</td>
<td>944.096.941*.064.943.083.200.12922</td>
<td></td>
</tr>
<tr>
<td>la24</td>
<td>977.00</td>
<td>984.072.982*.051.986.093.200.12931</td>
<td></td>
</tr>
<tr>
<td>la25</td>
<td>1218.00</td>
<td>1218*.00</td>
<td>1218.00.66.13547</td>
</tr>
<tr>
<td>la26</td>
<td>1235.00</td>
<td>1258.1.86.1256*.1.70.1266.2.51.200.41364</td>
<td></td>
</tr>
<tr>
<td>la27</td>
<td>1261.00</td>
<td>1218.016.1216*.00.1223.0.56.179.36085</td>
<td></td>
</tr>
<tr>
<td>la28</td>
<td>1152.00</td>
<td>1184*.2.78.1191.3.39.1199.4.04.200.39881</td>
<td></td>
</tr>
<tr>
<td>la29</td>
<td>1355.00</td>
<td>1355*.00.1355*.00.1355.0.00.10.1895</td>
<td></td>
</tr>
<tr>
<td>la30</td>
<td>1784.00</td>
<td>1784*.00.1784*.00.1784.0.00.1.701</td>
<td></td>
</tr>
<tr>
<td>la31</td>
<td>1850.00</td>
<td>1850*.00</td>
<td>1850.00.1.849</td>
</tr>
<tr>
<td>la32</td>
<td>1719.00</td>
<td>1719*.00.1719*.00.1719.0.00.1.725</td>
<td></td>
</tr>
<tr>
<td>la33</td>
<td>1721.00</td>
<td>1721*.00.1721*.00.1721.0.00.1.1255</td>
<td></td>
</tr>
<tr>
<td>la34</td>
<td>1888.00</td>
<td>1888*.00.1888*.00.1888.0.00.1.902</td>
<td></td>
</tr>
<tr>
<td>la35</td>
<td>1268.00</td>
<td>1278.079.1278.079.1288.1.61.200.48387</td>
<td></td>
</tr>
<tr>
<td>la36</td>
<td>1397.00</td>
<td>1410.093.1407*.072.1415.1.32.200.49836</td>
<td></td>
</tr>
<tr>
<td>la37</td>
<td>1196.00</td>
<td>1221.2.09.1215*.1.59.1232.3.02.200.50876</td>
<td></td>
</tr>
<tr>
<td>la38</td>
<td>1233.00</td>
<td>1251.1.46.1250*.1.38.1252.1.51.200.50603</td>
<td></td>
</tr>
<tr>
<td>la39</td>
<td>1222.00</td>
<td>1229.0.57.1229.0.57.1242.1.61.200.50609</td>
<td></td>
</tr>
</tbody>
</table>

(iii) The column UPLA in each table is divided into six columns, i.e., Best, %BSVD, Avg., %ASVD, Avg. No. of Iters, and Avg. CPU Time (sec). Their definitions are given as follows:

(a) Best stands for the best-found solution value over 10 runs of UPLA
(b) %BSVD stands for the deviation percentage between the best-found solution value over 10 runs of UPLA and the optimal solution value
(c) Avg. stands for the average of the best-found solution values from the 1st run to the 10th run of UPLA
(d) %ASVD stands for the deviation percentage between the average of the best-found solution values from the 1st run to the 10th run of UPLA and the optimal solution value
(e) Avg. No. of Iters shows the average number of iterations used by UPLA until it reaches the stopping criterion over 10 runs
The combination of UPLA and LOLA enhances the performance of an isolated LOLA in solution quality but worsens the performance in CPU-time consumption. One of its causes is that if the optimal solution cannot be found, UPLA will run until the 200th iteration as its predetermined maximum iteration. However, the use of the 200th iteration as the UPLA’s maximum iteration is usually higher than necessary since UPLA always finds its best-found solution before the 200th iteration (and it cannot find any better solution since then) in most instances. Thus, by a properly lower maximum iteration, UPLA can finish its computational process faster without any effect on solution quality. To determine the proper maximum iteration for UPLA, Figure 2 shows the %ASVD-over-iteration plots on the three hard-to-solve instances, i.e., la27, la29, and la38.

In Figure 2, the %ASVD-over-iteration plots show the similar patterns in their three periods of iterations, i.e., the 1st to the 60th iterations, the 60th to the 150th iterations, and the 150th to the 200th iterations. The %ASVD value of each plot is reduced rapidly in the first 60 iterations. After that, from the 60th to the 150th iterations, the %ASVD value continues to be reduced at a slow rate. Finally, the %ASVD value is almost stable after the 150th iteration. Based on the abovementioned finding, the maximum iteration is recommended to be in a range between the 60th iteration and the 150th iteration. At extreme points, the maximum iteration at the 60th iteration should be used when a short CPU time is required, while the 150th iteration should be used when the user is very concerned about the solution’s quality. The maximum iteration is not recommended to be lesser than the 60th iteration since it tends to stop UPLA prematurely, and it is also not recommended to be greater than the 150th iteration since UPLA will have a very low possibility to find a better solution after the 150th iteration.

The results of the GA [6] and UPLA on the 10 orb instances in Table 3 are also compared in the same three indicators. In counting the * signs, the GA can find the optimal solutions on 5 instances, while UPLA can find the optimal solutions on 7 instances. In counting the # signs, the GA cannot find better solutions than UPLA on any instances, while UPLA finds better solutions than the GA on 5 instances. In the %BSVD values, the average of the %BSVD values of the GA is 0.49%, while the average of the %BSVD values of UPLA is 0.06%. Hence, the conclusion is that UPLA performs better than the GA [6] in all the three indicators.

The results of the two-level PSO [15] and UPLA on the 43 ft-and-la instances are taken from Table 2 to compare in the three indicators, i.e., the number of * signs, the number of # signs, and the %BSVD values. In counting the instances where optimal solutions can be found over the total 43 instances (counting * signs), the two-level PSO can find the optimal solutions on 31 instances, while UPLA can find the optimal solutions on 33 instances. In counting the instances where each algorithm can find better solutions than the other (counting # signs), the two-level PSO finds better solutions than UPLA on only 2 instances, while UPLA finds better solutions than the two-level PSO on 9 instances. In the %BSVD values, the average of the %BSVD values of the two-level PSO is 0.32%, while the average of the %BSVD values of UPLA is 0.28%. In conclusion, UPLA performs better than the two-level PSO [15] in all the three indicators.

The results of the GA [6] and UPLA on the 10 orb instances in Table 3 are also compared in the same three indicators. In counting the * signs, the GA can find the optimal solutions on 5 instances, while UPLA can find the optimal solutions on 7 instances. In counting the # signs, the GA cannot find better solutions than UPLA on any instances, while UPLA finds better solutions than the GA on 5 instances. In the %BSVD values, the average of the %BSVD values of the GA is 0.49%, while the average of the %BSVD values of UPLA is 0.06%. Hence, the conclusion is that UPLA performs better than the GA [6] in all the three indicators.

The combination of UPLA and LOLA enhances the performance of an isolated LOLA in solution quality but worsens the performance in CPU-time consumption. One of its causes is that if the optimal solution cannot be found, UPLA will run until the 200th iteration as its predetermined maximum iteration. However, the use of the 200th iteration as the UPLA’s maximum iteration is usually higher than necessary since UPLA always finds its best-found solution before the 200th iteration (and it cannot find any better solution since then) in most instances. Thus, by a properly lower maximum iteration, UPLA can finish its computational process faster without any effect on solution quality. To determine the proper maximum iteration for UPLA, Figure 2 shows the %ASVD-over-iteration plots on the three hard-to-solve instances, i.e., la27, la29, and la38.

In Figure 2, the %ASVD-over-iteration plots show the similar patterns in their three periods of iterations, i.e., the 1st to the 60th iterations, the 60th to the 150th iterations, and the 150th to the 200th iterations. The %ASVD value of each plot is reduced rapidly in the first 60 iterations. After that, from the 60th to the 150th iterations, the %ASVD value continues to be reduced at a slow rate. Finally, the %ASVD value is almost stable after the 150th iteration. Based on the abovementioned finding, the maximum iteration is recommended to be in a range between the 60th iteration and the 150th iteration. At extreme points, the maximum iteration at the 60th iteration should be used when a short CPU time is required, while the 150th iteration should be used when the user is very concerned about the solution’s quality. The maximum iteration is not recommended to be lesser than the 60th iteration since it tends to stop UPLA prematurely, and it is also not recommended to be greater than the 150th iteration since UPLA will have a very low possibility to find a better solution after the 150th iteration.

Table 3: Experiment’s results on the set of orb instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Opt</th>
<th>GA [6] Best %BSVD</th>
<th>UPLA Best %BSVD</th>
<th>Avg. %ASVD</th>
<th>Avg. No. of Iters</th>
<th>Avg. CPU Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>orb01</td>
<td>1059</td>
<td>1077 1.70 0.00</td>
<td>1059 0.93 186</td>
<td>2312</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orb02</td>
<td>888</td>
<td>889 0.11 0.11</td>
<td>889 0.11 200</td>
<td>2393</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orb03</td>
<td>1005</td>
<td>1022 1.69 0.00</td>
<td>1021 1.58 186</td>
<td>2358</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orb04</td>
<td>1005</td>
<td>1005 0.00 0.00</td>
<td>1006 0.11 67</td>
<td>796</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orb05</td>
<td>887</td>
<td>890 0.34 0.23</td>
<td>890 0.32 200</td>
<td>2458</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orb06</td>
<td>1010</td>
<td>1021 1.09 0.30</td>
<td>1019 0.93 200</td>
<td>2525</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orb07</td>
<td>397</td>
<td>397 0.00 0.00</td>
<td>399 0.55 178</td>
<td>2096</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orb08</td>
<td>899</td>
<td>899 0.00 0.00</td>
<td>909 1.07 188</td>
<td>2338</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orb09</td>
<td>934</td>
<td>934 0.00 0.00</td>
<td>935 0.05 74</td>
<td>884</td>
<td></td>
<td></td>
</tr>
<tr>
<td>orb10</td>
<td>944</td>
<td>944 0.00 0.00</td>
<td>944 0.00 67</td>
<td>817</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(f) Avg. CPU Time (sec) shows the average computational time (in seconds) used by UPLA until it reaches the stopping criterion over 10 runs

(iv) The column PSO in Table 2 and the column GA in Table 3 are each divided into two columns, i.e., Best and %BSVD. Best in the column PSO means the two-level PSO’s best-found solution value [15], while Best in the column GA means the GA’s best-found solution value [6]. %BSVD means the deviation percentage between the specified algorithm’s best-found solution value and the optimal solution value

(v) Each best-found solution value will be marked by an asterisk (*) if its value is the same as the optimal solution value; in addition, it will be marked by a sharp sign (#) if it wins in comparison

The results of the GA [6] and UPLA on the 43 ft-and-la instances are taken from Table 2 to compare in the three indicators, i.e., the number of * signs, the number of # signs, and the %BSVD values. In counting the instances where optimal solutions can be found over the total 43 instances (counting * signs), the two-level PSO can find the optimal solutions on 31 instances, while UPLA can find the optimal solutions on 33 instances. In counting the instances where each algorithm can find better solutions than the other (counting # signs), the two-level PSO finds better solutions than UPLA on only 2 instances, while UPLA finds better solutions than the two-level PSO on 9 instances. In the %BSVD values, the average of the %BSVD values of the two-level PSO is 0.32%, while the average of the %BSVD values of UPLA is 0.28%. In conclusion, UPLA performs better than the two-level PSO [15] in all the three indicators.

The results of the GA [6] and UPLA on the 10 orb instances in Table 3 are also compared in the same three indicators. In counting the * signs, the GA can find the optimal solutions on 5 instances, while UPLA can find the optimal solutions on 7 instances. In counting the # signs, the GA cannot find better solutions than UPLA on any instances, while UPLA finds better solutions than the GA on 5 instances. In the %BSVD values, the average of the %BSVD values of the GA is 0.49%, while the average of the %BSVD values of UPLA is 0.06%. Hence, the conclusion is that UPLA performs better than the GA [6] in all the three indicators.
6. Conclusion

This paper introduced the two-level metaheuristic algorithm, consisting of LOLA as the lower-level algorithm and UPLA as the upper-level algorithm, for the JSP. LOLA serves as a local search algorithm to search for an optimal schedule, while UPLA serves as the parameter controller for LOLA. In more detail, UPLA is a new population-based search algorithm developed for adjusting the values of the LOLA’s input parameters. UPLA has an important role in evolving LOLA to perform its best for every single instance. For example, UPLA controls the LOLA’s solution space by the acceptable idle-time limit $\delta$, and it also upgrades LOLA from a local search algorithm to an iterated local search algorithm by the perturbation method $IP$. The numerical experiment in this paper showed that the proposed two-level metaheuristic algorithm outperforms the two other metaheuristic algorithms taken from the literature in terms of solution quality. A further study on this research should be generalizing UPLA into a general form that can be used for controlling the parameters of different metaheuristic algorithms.

Data Availability

The data used to support the findings of this study are available from the author upon request.

Conflicts of Interest

The author declares that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The author would like to acknowledge partial financial support from the Thai-Nichi Institute of Technology, Thailand.

References


