Hindawi
Complexity
Volume 2019, Article ID 8683472, 11 pages
https://doi.org/10.1155/2019/8683472

WILEY | Hindawi

Research Article

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 7 March 2019

Academic Editor: Chongyang Liu

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,

the UPLA's function is not to find the problem solutions; however, it is designed for being an input-parameter controller specifically for LOLA. At an initial iteration, UPLA starts with a population of the combinations of LOLA's input-parameter values. All input-parameter values in each input-parameter-value combination are represented in real numbers; most of them need to be decoded from real numbers to the other forms understandable for LOLA. At each iteration, UPLA tries to improve its population by using the feedback returned from LOLA. By the support of UPLA, LOLA will be upgraded from a local search algorithm to be an iterated local search algorithm. Moreover, the combination of UPLA and LOLA results in the two-level algorithm which can adapt itself for every single JSP instance.

The remainder of this paper is structured as follows. Section 2 provides the preliminary knowledge relevant to this research. Section 3 presents the proposed lower-level algorithm (LOLA), including its decoding procedure for generating solutions. Section 4 presents the proposed upper-level algorithm (UPLA), including its decoding procedure for generating LOLA's input-parameter values. Section 5 then evaluates the performance of the proposed two-level metaheuristic algorithm on well-known JSP instances. Finally, Section 6 provides a conclusion.

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 $O_{i1}, O_{i2}, \ldots, O_{im}$. 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_{im}. 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_{im} 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 a makespan 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 when it can start processing an operation. Thus, the solution space of active schedules is a subset of the solution space of semiactive schedules, and the solution space of nondelay schedules is a subset of the solution space of active schedules. The solution space of active schedules is surely dominant over the solution space of semiactive schedules since it is smaller and also guaranteed to contain an optimal schedule. The solution space of nondelay schedules, which is the smallest solution space, may not contain an optimal schedule.

In addition to the three solution spaces given above, the solution space of *parameterized-active schedules* or *hybrid schedules* [16, 17] is a subset of the solution space of active schedules which is not smaller than the solution space of nondelay schedules. In definition, a parameterized-active schedule is an active schedule where no machine runs 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 δ 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 δ is problem-dependent, the researchers try to control the tunable parameter δ in many different ways. Some metaheuristic algorithms such as [16, 33] use δ with fixed values. On the other hand, some other metaheuristic algorithms such as [34, 35] adjust the δ values during their evolutionary processes; thus, their methods of changing the δ values belong to the classification of *self-adaptive parameter control* based on the definition given by [10]. In addition, some metaheuristic algorithms adjust the δ values based on the mathematical functions of another parameter (or other parameters); for example, the local search algorithms of [31] adjust the δ 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 δ for the lower-level algorithm has been found in [15]. Hereafter, the parameter δ 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 $1, 2, \ldots, n$, 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 $\delta \in [0, 1)$, 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 $\sigma^* + \delta(\varphi^* - \sigma^*)$ 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 $\delta \in [0, 1)$ 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 be (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 J_i (where $i = 1, 2, \ldots, n$), the operations $O_{i1}, O_{i2}, \ldots, O_{im}$ must be renamed $O_{im}, O_{im-1}, \ldots, O_{i1}$, respectively

Step 2.2. Assign the precedence relations of all operations of the job J_i by following the operation indices taken from Step 2.1. This means that O_{i1} must be finished before O_{i2} can start, O_{i2} must be finished before O_{i3} 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 O_{ij}; 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, 2, 1, 3, 1) means priority of O_{31} > priority of O_{21} > priority of O_{22} > 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 Φ_t be the partial schedule of the t scheduled operations. Thus, Φ_0 is empty, and S_1 consists of all O_{i1} operations (where $i = 1, 2 \ldots, n$). In addition, let the earliest possible start times of all O_{i1} operations be equal to the time 0. Now, let $t = 1$

Step 5. Let σ^* be the minimum of the earliest possible start times of all schedulable operations in S_t, and let φ^* 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 $\sigma^* + \delta(\varphi^* - \sigma^*)$ in S_t. Then, create Φ_t by allocating O_h into Φ_{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 Φ_{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 Φ_{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 Φ_{mn} to satisfy the original precedence constraints by using Steps 8.1 and 8.2

Step 8.1. Let the operations $O_{im}, O_{im-1}, \ldots, O_{i1}$ of each job J_i in the schedule Φ_{mn} be renamed $O_{i1}, O_{i2}, \ldots, O_{im}$, 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 Φ_{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 $IP \in \{\text{full randomization}, \text{partial randomization}\}$ 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 operator n times) on the best-found operation-based permutation memorized by UPLA

(iv) The method for generating a neighbor operation-based permutation $NP \in \{\text{2-insert}, \text{1-insert/1-swap}, \text{1-swap/1-insert}, \text{2-swap}\}$ 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 LOLA's 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 $D \in \{\text{forward, backward}\}$

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 $NP \in \{\text{2-insert, 1-insert/1-swap, 1-swap/1-insert, 2-swap}\}$

Step 2. Let P_0 represent the LOLA's 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 δ 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 δ 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 LOLA's 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 δ, 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 LOLA's input-parameter values, consisting of $C_1(t), C_2(t), \ldots, C_N(t)$. Let $C_i(t) \equiv \{\delta_i(t), d_i(t), ip_i(t), np_i(t)\}$ represent the ith combination of the LOLA's input-parameter values (where $i = 1, 2, \ldots, N$) at the tth iteration. $\delta_i(t) \in [0, 1)$, $d_i(t) \in \mathbb{R}$, $ip_i(t) \in \mathbb{R}$, and $np_i(t) \in \mathbb{R}$ in the combination $C_i(t)$ represent the acceptable idle-time limit δ, 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 $\delta_i(t)$, $d_i(t)$, $ip_i(t)$, and $np_i(t)$ of the combination $C_i(t)$ into δ, D, IP, and NP of LOLA. In short, let a combination of LOLA's input-parameter values be called a *parameter-value combination*.

The performance of $C_i(t)$ is simply equal to the makespan returned from the LOLA using the parameter values decoded from $C_i(t)$. 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 the lowest makespan. In UPLA, let $C_{best} \equiv \{\delta_{best}, d_{best}, ip_{best}, np_{best}\}$ represent the best-found parameter-value combination memorized by UPLA. Thus, C_{best} will always be updated whenever UPLA can find a parameter-value combination which performs better than the current C_{best}. In other words, once UPLA finds a $C_i(t)$ performing better than the current C_{best}, this $C_i(t) \equiv \{\delta_i(t), d_i(t), ip_i(t), np_i(t)\}$ will become the new $C_{best} \equiv \{\delta_{best}, d_{best}, ip_{best}, np_{best}\}$.

In the tth UPLA's iteration, $\delta_i(t)$, $d_i(t)$, $ip_i(t)$, and $np_i(t)$, as the members of $C_i(t)$, will each be updated respectively into $\delta_i(t + 1)$, $d_i(t + 1)$, $ip_i(t + 1)$, and $np_i(t + 1)$ of $C_i(t + 1)$ by two vectors. The two vectors, used for updating each member's current value in $C_i(t)$ to its new value in $C_i(t + 1)$, are described in directions and magnitudes as follows. In the vector directions, the first vector's direction is toward the current value in $C_i(t)$ to the best-found value in C_{best}, while the second vector's direction is opposite the first vector's direction. In the vector magnitudes, if the current value in $C_i(t)$ and the best-found value in C_{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 UPLA combined with LOLA is summarized in Figure 1, where $\text{Makespan}_i(t)$ stands for the makespan of the schedule returned from the LOLA using the input-parameter values decoded from $C_i(t)$. The procedure of UPLA is fully presented in Algorithm 3. In Algorithm 3, Steps 1 and 2 initialize the UPLA's population. Although the largest possible range of $\delta_i(t)$ is $[0, 1)$ as mentioned earlier, the algorithm generates $\delta_i(t)$ 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 $C_i(t) \equiv \{\delta_i(t), d_i(t), ip_i(t), np_i(t)\}$ into δ, D, IP, and NP for LOLA, and this step then executes the LOLA using the input-parameter values transformed from $C_i(t)$. After

Table 1: Translation of $C_i(t)$ of UPLA into the input-parameter values of LOLA.

UPLA	LOLA	Relationship
$\delta_i(t) \in [0, 1)$	$\delta \in [0, 1)$	$\delta = \delta_i(t)$
$d_i(t) \in \mathbb{R}$	$D \in \{\text{forward}, \text{backward}\}$	$D = \begin{cases} \text{forward} & \text{if } d_i(t) < 0.5 \\ \text{backward} & \text{if } d_i(t) \geq 0.5 \end{cases}$
$ip_i(t) \in \mathbb{R}$	$IP \in \{\text{full randomization}, \text{partial randomization}\}$	$IP = \begin{cases} \text{full randomization} & \text{if } ip_i(t) < 0.75 \\ \text{partial randomization} & \text{if } ip_i(t) \geq 0.75 \end{cases}$
$np_i(t) \in \mathbb{R}$	$NP \in \{\text{2-insert}, \text{1-insert/1-swap}, \text{1-swap/1-insert}, \text{2-swap}\}$	$NP = \begin{cases} \text{2-insert} & \text{if } np_i(t) < 0.25 \\ \text{1-insert/1-swap} & \text{if } 0.25 \leq np_i(t) < 0.50 \\ \text{1-swap/1-insert} & \text{if } 0.5 \leq np_i(t) < 0.75 \\ \text{2-swap} & \text{if } np_i(t) \geq 0.75 \end{cases}$

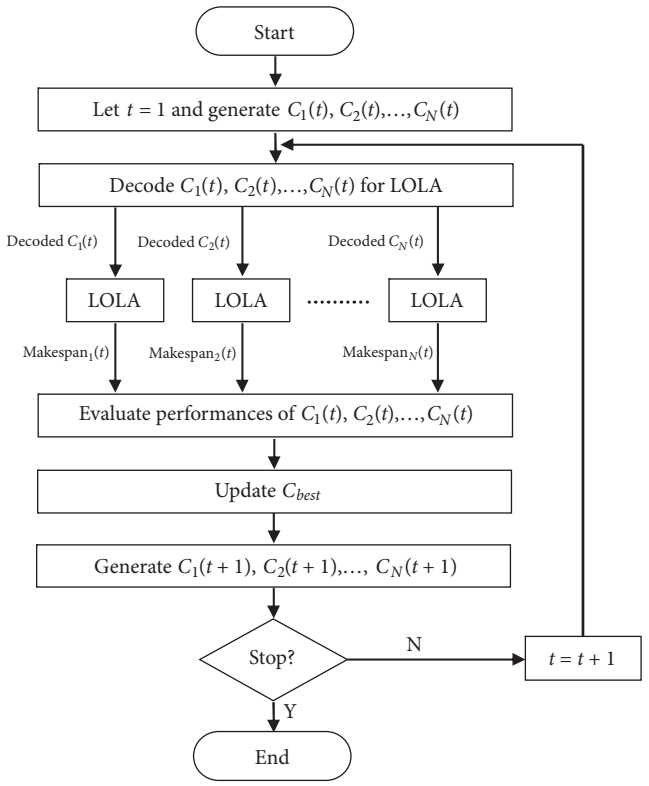

Figure 1: Flow chart of the UPLA combined with LOLA.

that, Step 3 evaluates the performance of each $C_i(t)$, where $i = 1, 2, \ldots, N$, and updates C_{best}. Step 4 then updates $C_i(t)$ to $C_i(t + 1)$; however, all values in $C_i(t + 1)$ will be randomly regenerated once for every 25 iterations for the purpose of diversifying the population. Step 5 finally checks the condition to stop or continue the next iteration.

Algorithm 3. It is the procedure of UPLA.

Step 1. Let $C_i(t) \equiv \{\delta_i(t), d_i(t), ip_i(t), np_i(t)\}$ represent the ith parameter-value combination (where $i = 1, 2, \ldots, N$) at the tth iteration. Let $C_{best} \equiv \{\delta_{best}, d_{best}, ip_{best}, np_{best}\}$ 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: $\delta_i(t) \sim U[0.8, 0.9)$, $d_i(t) \sim U[0, 1)$, $ip_i(t) \sim U[0, 1)$, and $np_i(t) \sim U[0, 1)$

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 $\delta_i(t)$, $d_i(t)$, $ip_i(t)$, and $np_i(t)$ of $C_i(t)$ into δ, D, IP, and NP for LOLA by the relationships shown in Table 1. Then, execute the LOLA (Algorithm 2) which uses δ, D, IP, and NP 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 $C_{best} \equiv \{\delta_{best}, d_{best}, ip_{best}, np_{best}\}$ to be equal to $C_i(t) \equiv \{\delta_i(t), d_i(t), ip_i(t), np_i(t)\}$, 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 3.2; otherwise, go to Step 4

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

Step 4.1. Let $i = 1$

Step 4.2. Generate $u_1, u_2 \sim U[0, 1)$, and then generate $\delta_i(t + 1)$ as follows:

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

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:

$$d_i(t+1) = \begin{cases} d_i(t) + 0.05u_1 - 0.01u_2 & \text{if } d_i(t) < d_{best} \\ d_i(t) - 0.05u_1 + 0.01u_2 & \text{if } d_i(t) > d_{best} \\ d_i(t) + 0.01u_1 - 0.01u_2 & \text{if } d_i(t) = d_{best} \end{cases} \quad (2)$$

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

$$ip_i(t+1) = \begin{cases} ip_i(t) + 0.05u_1 - 0.01u_2 & \text{if } ip_i(t) < ip_{best} \\ ip_i(t) - 0.05u_1 + 0.01u_2 & \text{if } ip_i(t) > ip_{best} \\ ip_i(t) + 0.01u_1 - 0.01u_2 & \text{if } ip_i(t) = ip_{best} \end{cases} \quad (3)$$

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

$$np_i(t+1) = \begin{cases} np_i(t) + 0.05u_1 - 0.01u_2 & \text{if } np_i(t) < np_{best} \\ np_i(t) - 0.05u_1 + 0.01u_2 & \text{if } np_i(t) > np_{best} \\ np_i(t) + 0.01u_1 - 0.01u_2 & \text{if } np_i(t) = np_{best} \end{cases} \quad (4)$$

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 composed of ft06 to la40 and the set of orb instances composed of orb01 to orb10. To evaluate UPLA's performance, UPLA will be compared to the two-level PSO of [15] 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

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.

Instance	Opt	PSO [15]		UPLA					
		Best	%BSVD	Best	%BSVD	Avg.	%ASVD	Avg. No. of Iters	Avg. CPU Time (sec)
ft06	55	55*	0.00	55*	0.00	55	0.00	1	0.3
ft10	930	930*	0.00	930*	0.00	931	0.13	92	1208
ft20	1165	1165*	0.00	1165*	0.00	1172	0.62	188	4114
la01	666	666*	0.00	666*	0.00	666	0.00	1	1
la02	655	655*	0.00	655*	0.00	655	0.00	2	2
la03	597	597*	0.00	597*	0.00	597	0.00	10	13
la04	590	590*	0.00	590*	0.00	590	0.00	3	4
la05	593	593*	0.00	593*	0.00	593	0.00	1	1
la06	926	926*	0.00	926*	0.00	926	0.00	1	4
la07	890	890*	0.00	890*	0.00	890	0.00	1	5
la08	863	863*	0.00	863*	0.00	863	0.00	1	5
la09	951	951*	0.00	951*	0.00	951	0.00	1	4
la10	958	958*	0.00	958*	0.00	958	0.00	1	4
la11	1222	1222*	0.00	1222*	0.00	1222	0.00	1	12
la12	1039	1039*	0.00	1039*	0.00	1039	0.00	1	13
la13	1150	1150*	0.00	1150*	0.00	1150	0.00	1	12
la14	1292	1292*	0.00	1292*	0.00	1292	0.00	1	12
la15	1207	1207*	0.00	1207*	0.00	1207	0.00	1	17
la16	945	945*	0.00	945*	0.00	945	0.03	124	1458
la17	784	784*	0.00	784*	0.00	784	0.00	6	78
la18	848	848*	0.00	848*	0.00	848	0.00	6	76
la19	842	842*	0.00	842*	0.00	843	0.17	90	1130
la20	902	907	0.55	902*#	0.00	903	0.06	107	1304
la21	1046	1046*#	0.00	1052	0.57	1058	1.18	200	13505
la22	927	935	0.86	927*#	0.00	935	0.85	191	12840
la23	1032	1032*	0.00	1032*	0.00	1032	0.00	2	116
la24	935	944	0.96	941#	0.64	943	0.83	200	12922
la25	977	984	0.72	982#	0.51	986	0.93	200	12931
la26	1218	1218*	0.00	1218*	0.00	1218	0.00	66	13547
la27	1235	1258	1.86	1256#	1.70	1266	2.51	200	41364
la28	1216	1218	0.16	1216*#	0.00	1223	0.56	179	36085
la29	1152	1184#	2.78	1191	3.39	1199	4.04	200	39881
la30	1355	1355*	0.00	1355*	0.00	1355	0.00	10	1895
la31	1784	1784*	0.00	1784*	0.00	1784	0.00	1	701
la32	1850	1850*	0.00	1850*	0.00	1850	0.00	1	849
la33	1719	1719*	0.00	1719*	0.00	1719	0.00	1	725
la34	1721	1721*	0.00	1721*	0.00	1721	0.00	1	1255
la35	1888	1888*	0.00	1888*	0.00	1888	0.00	1	902
la36	1268	1278	0.79	1278	0.79	1288	1.61	200	48387
la37	1397	1410	0.93	1407#	0.72	1415	1.32	200	49836
la38	1196	1221	2.09	1215#	1.59	1232	3.02	200	50876
la39	1233	1251	1.46	1250#	1.38	1252	1.51	200	50603
la40	1222	1229	0.57	1229	0.57	1242	1.61	200	50609

(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

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

Instance	Opt	GA [6]		UPLA					
		Best	%BSVD	Best	%BSVD	Avg.	%ASVD	Avg. No. of Iters	Avg. CPU Time (sec)
orb01	1059	1077	1.70	1059*#	0.00	1069	0.93	186	2312
orb02	888	889	0.11	889	0.11	889	0.11	200	2393
orb03	1005	1022	1.69	1005*#	0.00	1021	1.58	186	2358
orb04	1005	1005*	0.00	1005*#	0.00	1006	0.11	67	796
orb05	887	890	0.34	889#	0.23	890	0.32	200	2458
orb06	1010	1021	1.09	1013#	0.30	1019	0.93	200	2525
orb07	397	397*	0.00	397*	0.00	399	0.55	178	2096
orb08	899	899*	0.00	899*	0.00	909	1.07	188	2338
orb09	934	934*	0.00	934*	0.00	935	0.05	74	884
orb10	944	944*	0.00	944*	0.00	944	0.00	67	817

(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 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.

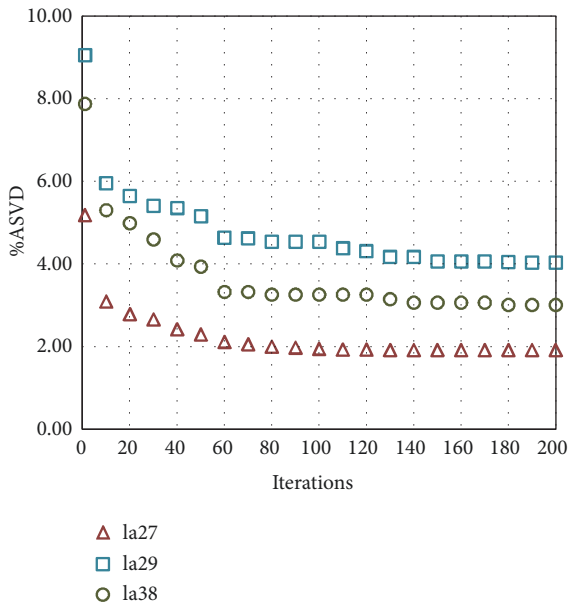

Figure 2: %ASVD-over-iteration plots of UPLA on la27, la29, and la38 instances.

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 δ, 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

[1] A. S. Jain and S. Meeran, "Deterministic job-shop scheduling: past, present and future," *European Journal of Operational Research*, vol. 113, no. 2, pp. 390–434, 1999.

[2] J. Błazewicz, W. Domschke, and E. Pesch, "The job shop scheduling problem: conventional and new solution techniques," *European Journal of Operational Research*, vol. 93, no. 1, pp. 1–33, 1996.

[3] U. Dorndorf and E. Pesch, "Evolution based learning in a job shop scheduling environment," *Computers & Operations Research*, vol. 22, no. 1, pp. 25–40, 1995.

[4] B. Peng, Z. Lü, and T. C. E. Cheng, "A tabu search/path relinking algorithm to solve the job shop scheduling problem," *Computers & Operations Research*, vol. 53, pp. 154–164, 2015.

[5] J. F. Gonçalves and M. G. C. Resende, "An extended Akers graphical method with a biased random-key genetic algorithm for job-shop scheduling," *International Transactions in Operational Research*, vol. 21, no. 2, pp. 215–246, 2014.

[6] N. H. Moin, O. C. Sin, and M. Omar, "Hybrid genetic algorithm with multiparents crossover for job shop scheduling problems," *Mathematical Problems in Engineering*, vol. 2015, Article ID 210680, 12 pages, 2015.

[7] T. Yamada and R. Nakano, "A fusion of crossover and local search," in *Proceedings of the IEEE International Conference on Industrial Technology (ICIT'96)*, pp. 426–430, IEEE, Shanghai, China, 1996.

[8] J.-Q. Li, H.-Y. Sang, Y.-Y. Han, C.-G. Wang, and K.-Z. Gao, "Efficient multi-objective optimization algorithm for hybrid flow shop scheduling problems with setup energy consumptions," *Journal of Cleaner Production*, vol. 181, pp. 584–598, 2018.

[9] U. Dorndorf, E. Pesch, and T. Phan-Huy, "Constraint propagation and problem decomposition: a preprocessing procedure for the job shop problem," *Annals of Operations Research*, vol. 115, no. 1-4, pp. 125–145, 2002.

[10] Á. E. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 2, pp. 124–141, 1999.

[11] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 16, no. 1, pp. 122–128, 1986.

[12] S.-J. Wu and P.-T. Chow, "Genetic algorithms for nonlinear mixed discrete-integer optimization problems via meta-genetic parameter optimization," *Engineering Optimization*, vol. 24, no. 2, pp. 137–159, 1995.

[13] T. Brys, M. M. Drugan, and A. Nowé, "Meta-evolutionary algorithms and recombination operators for satisfiability solving in fuzzy logics," in *Proceedings of the 2013 IEEE Congress on Evolutionary Computation*, pp. 1060–1067, IEEE, Cancun, Mexico, June 2013.

[14] P. Cortez, M. Rocha, and J. Neves, "A meta-genetic algorithm for time series forecasting," in *Proceedings of Workshop on Artificial Intelligence Techniques for Financial Time Series Analysis, 10th Portuguese Conference on Artificial Intelligence (EPIA 2001)*, pp. 21–31, Porto, Portugal, Dec 2001.

[15] P. Pongchairerks and V. Kachitvichyanukul, "A two-level particle swarm optimisation algorithm on job-shop scheduling problems," *International Journal of Operational Research*, vol. 4, no. 4, pp. 390–411, 2009.

[16] C. Bierwirth and D. C. Mattfeld, "Production scheduling and rescheduling with genetic algorithms," *Evolutionary Computation*, vol. 7, no. 1, pp. 1–17, 1999.

[17] J. F. Gonçalves, J. J. Mendes, and M. G. C. Resende, "A hybrid genetic algorithm for the job shop scheduling problem," *European Journal of Operational Research*, vol. 167, no. 1, pp. 77–95, 2005.

[18] R. Cheng, M. Gen, and Y. Tsujimura, "A tutorial survey of job-shop scheduling problems using genetic algorithms – I. representation," *Computers & Industrial Engineering*, vol. 30, no. 4, pp. 983–997, 1996.

[19] M. Gen and R. Cheng, *Genetic Algorithms and Engineering Design*, John Wiley & Sons, New York, NY, USA, 1997.

[20] C. Bierwirth, "A generalized permutation approach to job shop scheduling with genetic algorithms," *OR Spectrum*, vol. 17, no. 2-3, pp. 87–92, 1995.

[21] J. Kennedy, R. C. Eberhart, and Y. Shi, *Swarm Intelligence*, Morgan Kaufmann, San Francisco, Calif, USA, 2001.

[22] R. Storn and K. Price, "Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[23] X.-S. Yang and X. He, "Firefly algorithm: recent advances and applications," *International Journal of Swarm Intelligence*, vol. 1, no. 1, pp. 36–50, 2013.

[24] X.-S. Yang and S. Deb, "Engineering optimisation by Cuckoo search," *International Journal of Mathematical Modelling and Numerical Optimisation*, vol. 1, no. 4, pp. 330–343, 2010.

[25] M. Neshat, G. Sepidnam, M. Sargolzaei, and A. N. Toosi, "Artificial fish swarm algorithm: a survey of the state-of-the-art, hybridization, combinatorial and indicative applications," *Artificial Intelligence Review*, vol. 42, no. 4, pp. 965–997, 2014.

[26] Z.-X. Zheng, J.-Q. Li, and P.-Y. Duan, "Optimal chiller loading by improved artificial fish swarm algorithm for energy saving," *Mathematics and Computers in Simulation*, vol. 155, pp. 227–243, 2019.

[27] H. R. Lourenço, "Job-shop scheduling: Computational study of local search and large-step optimization methods," *European Journal of Operational Research*, vol. 83, no. 2, pp. 347–364, 1995.

[28] D. Sun and L. Lin, "A dynamic job shop scheduling framework: A backward approach," *International Journal of Production Research*, vol. 32, no. 4, pp. 967–985, 1994.

[29] L. Özdamar, "A genetic algorithm approach to a general category project scheduling problem," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 29, no. 1, pp. 44–59, 1999.

[30] V. K. Ganesan, A. I. Sivakumar, and G. Srinivasan, "Hierarchical minimization of completion time variance and makespan in jobshops," *Computers & Operations Research*, vol. 33, no. 5, pp. 1345–1367, 2006.

[31] P. Pongchairerks, "Efficient local search algorithms for job-shop scheduling problems," *International Journal of Mathematics in Operational Research*, vol. 9, no. 2, pp. 258–277, 2016.

[32] M. Moonen and G. K. Janssens, "Giffler-thompson focused genetic algorithm for the static job-shop scheduling problem," *Journal of Information and Computational Science*, vol. 4, no. 2, pp. 629–642, 2007.

[33] P. Pongchairerks, "Particle swarm optimization algorithm applied to scheduling problems," *ScienceAsia*, vol. 35, no. 1, pp. 89–94, 2009.

[34] P. Pongchairerks, "A self-tuning PSO for job-shop scheduling problems," *International Journal of Operational Research*, vol. 19, no. 1, pp. 96–113, 2014.

[35] D. Petrovic, E. Castro, S. Petrovic, and T. Kapamara, "Radiotherapy scheduling," in *Automated Scheduling and Planning*, vol. 505 of *Studies in Computational Intelligence*, pp. 155–189, Springer, Berlin, Germany, 2013.

[36] Y. Crama, A. W. J. Kolen, and E. J. Pesch, "Local search in combinatorial optimization," in *Artificial Neural Networks*, vol. 931 of *Lecture Notes in Computer Science*, pp. 157–174, Springer, Berlin, Germany, 1995.

[37] J. B. Orlin, A. P. Punnen, and A. S. Schulz, "Approximate local search in combinatorial optimization," *SIAM Journal on Computing*, vol. 33, no. 5, pp. 1201–1214, 2004.

[38] W. Michiels, E. Aarts, and J. Korst, *Theoretical Aspects of Local Search*, Springer, Berlin, Germany, 2007.

[39] E. Pesch, *Learning in Automated Manufacturing: A Local Search Approach*, Physica-Verlag, Heidelberg, Germany, 1994.

[40] M. den Besten, T. Stützle, and M. Dorigo, "Design of iterated local search algorithms: an example application to the single machine total weighted tardiness problem," in *Applications of Evolutionary Computing*, vol. 2037 of *Lecture Notes in Computer Science*, pp. 441–451, Springer, Berlin, Germany, 2001.

[41] M. F. Tasgetiren, O. Buyukdagli, Q.-K. Pan, and P. N. Suganthan, "A general variable neighborhood search algorithm for the no-idle permutation flowshop scheduling problem," in *Swarm, Evolutionary, and Memetic Computing*, vol. 8297 of *Lecture Notes in Computer Science*, pp. 24–34, Springer, Cham, Germany, 2013.

[42] H. R. Lourenço, O. C. Martin, and T. Stützle, "A beginner's introduction to iterated local search," in *Proceedings of the 4th Metaheuristics International Conference*, pp. 1–6, Porto, Portugal, 2001.

[43] P. Pongchairerks and V. Kachitvichyanukul, "A non-homogenous particle swarm optimization with multiple social structures," in *Proceedings of the 2005 International Conference on Simulation and Modeling*, pp. 132–136, Nakornpathom, Thailand, Jan 2005.

[44] H. Fisher and G. L. Thompson, "Probabilistic learning combinations of local job-shop scheduling rules," in *Industrial Scheduling*, J. F. Muth and G. L. Thompson, Eds., pp. 225–251, Prentice-Hall, Englewood, NJ, USA, 1963.

[45] S. Lawrence, "Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement)," Graduate School of Industrial Administration ORNL/Sub-7654/1, Carnegie Mellon University, Pittsburgh, PA, USA, 1984.

[46] D. Applegate and W. Cook, "A computational study of the job-shop scheduling problem," *ORSA Journal on Computing*, vol. 3, no. 2, pp. 149–156, 1991.

[47] J. E. Beasley, "Job shop scheduling," OR-Library, 2004, http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/jobshop1.txt.