Hindawi
Complexity
Volume 2020, Article ID 3489209, 15 pages
https://doi.org/10.1155/2020/3489209

Research Article

An Enhanced Two-Level Metaheuristic Algorithm with Adaptive Hybrid Neighborhood Structures 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 25 January 2020; Revised 17 April 2020; Accepted 27 April 2020; Published 28 June 2020

Academic Editor: José Manuel Galán

For solving the job-shop scheduling problem (JSP), this paper proposes a novel two-level metaheuristic algorithm, where its upper-level algorithm controls the input parameters of its lower-level algorithm. The lower-level algorithm is a local search algorithm searching for an optimal JSP solution within a hybrid neighborhood structure. To generate each neighbor solution, the lower-level algorithm randomly uses one of two neighbor operators by a given probability. The upper-level algorithm is a population-based search algorithm developed for controlling the five input parameters of the lower-level algorithm, i.e., a perturbation operator, a scheduling direction, an ordered pair of two neighbor operators, a probability of selecting a neighbor operator, and a start solution-representing permutation. Many operators are proposed in this paper as options for the perturbation and neighbor operators. Under the control of the upper-level algorithm, the lower-level algorithm can be evolved in its input-parameter values and neighborhood structure. Moreover, with the perturbation operator and the start solution-representing permutation controlled, the two-level metaheuristic algorithm performs like a multistart iterated local search algorithm. The experiment's results indicated that the two-level metaheuristic algorithm outperformed its previous variant and the two other high-performing algorithms in terms of solution quality.

1. Introduction

Production scheduling is an important tool for controlling and optimizing workloads in an industrial production system. It is a decision-making process which involves assigning jobs to machines on a timetable. The job-shop scheduling problem (JSP) is one of well-known production scheduling problems. Such a problem is also defined as a much complex optimization problem both in theoretical and practical aspects. The objective of JSP is commonly to find a feasible schedule which completes all jobs by the shortest makespan (note that *makespan* stands for the length of the schedule). Approximation algorithms, such as [1–6], have been developed for JSP since the problem cannot be optimally solved in a reasonable (polynomial) amount of time [7, 8]. The two-level metaheuristic algorithm of [9] is one of the high-performing approximation algorithms for JSP. This algorithm is, as its name implies, a combination between its upper-level algorithm (named UPLA) and its lower-level algorithm (named LOLA). Its mechanism is that UPLA controls the input parameters of LOLA, and LOLA then searches for an optimal schedule. Due to successful results of [9], this paper aims at developing an enhanced two-level metaheuristic algorithm for JSP. To do so, the two-level metaheuristic algorithm proposed in this paper has been changed from its original variant [9] in both levels.

The lower-level algorithm proposed in this paper, named LOSAP, is a *lo*cal *s*earch *a*lgorithm exploring in a *p*robabilistic-based hybrid neighborhood structure. To generate each neighbor solution, LOSAP randomly uses one from the two predetermined neighbor operators (i.e., the first and second operators) by a preassigned probability of selecting the first operator. A high value of this probability leads the

hybrid neighborhood structure to be more likely similar to the first operator's neighborhood structure, and vice versa. Previous successful applications of randomly using one from two different operators have been found in [10, 11]. Major differences between LOSAP and LOLA [9] are briefly presented as follows. The LOLA's search ability is mainly based on its special solution space, which is a solution space of parameterized-active schedules. The LOSAP's search ability is, however, mainly based on its hybrid neighborhood structure since it searches in an ordinary solution space of semiactive schedules. LOSAP also has many proposed operators as the options for its perturbation and neighbor operators; most of them were not used in LOLA. In addition, LOSAP uses a different criterion from LOLA on accepting a new best-found solution.

Although LOLA and LOSAP have many differences from each other as above-mentioned, they still share a common weakness. The weakness is that no single combination of input-parameter values performs best for all instances. In other words, a combination of input-parameter values performing best for an instance may not perform as best for another instance. For each instance, each algorithm has a specific best combination of input-parameter values; however, it cannot be foreknown without doing an experiment on the being-considered instance. This weakness is, in fact, a common weakness of most metaheuristic algorithms. To overcome such a weakness, past researches, e.g., [4, 9, 12–15], developed an upper-level metaheuristic algorithm to control input parameters of a problem-solving metaheuristic algorithm in a lower level.

The upper-level algorithm proposed in this paper, named MUPLA, is a *m*odification of UPLA (i.e., the upper-level algorithm of [9]). MUPLA is a population-based metaheuristic algorithm designed to be a parameter controller for LOSAP. Thus, its population consists of a number of combinations of the LOSAP's input-parameter values which are evolved (updated) over iterations. For short, let a combination of LOSAP's input-parameter values be called a *parameter-value combination*. Each parameter-value combination contains specific values of the perturbation operator, the scheduling direction, the ordered pair of two neighbor operators, the probability of selecting a neighbor operator, and the start solution-representing permutation. A major change of MUPLA from UPLA [9] is that each parameter-value combination has a different start solution-representing permutation from those of the others. As a consequence, the MUPLA combined with LOSAP acts as a multistart iterated local search algorithm. This is a large upgrade because the UPLA combined with LOLA [9] is just an iterated local search algorithm.

The remainder of this paper is divided into four main sections. Section 2 provides an overview of the relevant publications of the research topic. Section 3 describes the proposed two-level metaheuristic algorithm in both levels. Thus, the lower-level algorithm (LOSAP) and the upper-level algorithm (MUPLA) are described in Sections 3.1 and 3.2, respectively. Then, Section 4 presents the results and discussions on the evaluation of the two-level metaheuristic algorithm's performance. Section 5 finally summarizes the findings and recommendations.

2. Literature Review

The job-shop scheduling problem (JSP) comes with n given jobs $J_1, J_2, \ldots, J_n$ and m given machines $M_1, M_2, \ldots, M_m$. Each job J_i is composed of a sequence of m given operations $O_{i1}, O_{i2}, \ldots, O_{im}$ as a chain of precedence constraints. To process each job J_i, O_{ij} (where $j = 1, 2, \ldots, m-1$) is defined as an immediate-preceding operation of O_{ik} (where $k = j+1$); thus, O_{ij} must be finished before O_{ik} can start. In addition, each operation must be processed on a preassigned machine with a predetermined processing time. Each machine cannot process more than one operation at a time, and it cannot be stopped or paused during processing an operation. All n given jobs arrive at the time 0, and all m given machines are also available at the time 0. A schedule is feasible if it completely allocates all n jobs under all the given constraints. An optimal schedule is a feasible schedule which minimizes the makespan, i.e., the total amount of time required to complete all jobs. Excellent reviews about JSP are available in [7, 8, 16, 17].

In JSP, feasible schedules can be alternatively constructed in forward or backward (reverse) directions. A forward schedule is a schedule constructed in the forward direction, while a backward (reverse) schedule is a schedule constructed in the backward direction. In other words, a forward schedule is a schedule in which all jobs J_i (where $i = 1, 2, \ldots, n$) are constructed forward from O_{i1} to O_{im}, while a backward schedule is a schedule in which all jobs J_i are constructed backward from O_{im} to O_{i1}. Although the forward scheduling is commonly used for the makespan criterion, the backward scheduling as an alternative has been applied in many researches, e.g., [18–20]. Besides the schedule's classification based on the scheduling directions, the feasible schedules can be classified based on their allowable delay times, e.g., semiactive schedules and active schedules [17, 21]. A feasible schedule is defined as a semiactive schedule if no operation can be started earlier without altering an operation sequence on any machine. A semiactive schedule is then defined as an active schedule if no operation can be started earlier without delaying any other operation or violating any precedence constraint.

Many approximation algorithms have been developed based on metaheuristic algorithms for solving JSP. Iterated local search, a well-known type of metaheuristic algorithms, has also been applied for JSP [22, 23]. In general, an iterated local search algorithm is a single-solution-based local search technique which can search for a global optimal solution. During an exploration, it uses a neighbor operator repeatedly to find a local optimum and then uses a perturbation operator to escape the just-found local optimum (note that a *perturbation operator* stands for an operator that generates a new start solution by largely modifying a found local optimal solution [24, 25]). It has also been found that some researches such as [26–28] enhanced their iterated local search algorithms by adding multistart properties.

In the iterated local search and related algorithms, there are three operators commonly used as a neighbor operator and a perturbation operator. These three operators are the common swap, insert, and inverse operators [29]. Some iterated local search algorithms, such as [30, 31], use the

common swap operator or the common insert operator multiple times as their perturbation operators. The definitions of the three common operators are given as follows (let u and v are random integers from 1 to the number of all members in the permutation, and $v \neq u$):

(i) The common swap operator is to swap between the two members in the u^{th} and v^{th} positions of a permutation.

(ii) The common insert operator is to remove a member from the u^{th} position of a permutation and then insert it back at the v^{th} position.

(iii) The common inverse operator is to inverse the sequence of all members from the u^{th} to v^{th} positions of a permutation.

The two-level metaheuristic algorithm of [9] can be classified as an adaptive iterated local search algorithm for JSP. Its upper-level algorithm (named UPLA) controls the input parameters of its lower-level algorithm, and its lower-level algorithm (named LOLA) then searches for an optimal schedule. Thus, the two-level metaheuristic algorithm can adapt itself for every single JSP instance. The development of the two-level metaheuristic algorithm of [9] followed in the successes of the previous researches of [4, 12–15] in using a metaheuristic algorithm to control parameters of another metaheuristic algorithm.

In [9], LOLA is a local search algorithm exploring in a solution space of parameterized-active schedules. Its input parameters (i.e., an acceptable idle-time limit, a scheduling direction, a perturbation operator, and a neighbor operator) are controlled by UPLA. UPLA is a population-based metaheuristic algorithm searching in a real-number search space. In this view's point, UPLA is similar to the other population-based algorithms, such as particle swarm optimization [32], differential evolution [33], fish swarm [34], and cuckoo search [35]. However, the evolving procedure of the UPLA's population is different from those of the others mentioned. The UPLA's population consists of the combinations of input-parameter values of LOLA. For a parameter-value combination, each parameter's value is iteratively changed by a sum of two changeable opposite-direction vectors. The first vector's direction is toward the memorized best-found value, whereas the second vector's direction is away from. The magnitudes of these two vectors are generated randomly between zeros to their given maximum values. The first vector's maximum magnitude (0.05) is usually larger than the second vector's maximum magnitude (0.01). However, if the parameter's value equals the memorized best-found value, the maximum magnitudes of both vectors then equal the same value (0.01).

3. Method

Section 3 describes the procedure of the proposed two-level metaheuristic algorithm in both levels. In this section, the lower-level algorithm (LOSAP) is described in Section 3.1, and the upper-level algorithm (MUPLA) is described in Section 3.2.

3.1. Proposed Lower-Level Algorithm. The lower-level algorithm proposed in this paper, named LOSAP, is a local search algorithm searching for an optimal solution in a probabilistic-based hybrid neighborhood structure. Its framework is similar to those of the other local search algorithms [36–39]. However, its neighborhood structure is generated based on the two predetermined operators, i.e., the first and second neighbor operators. By a given probability, LOSAP randomly uses one from the two predetermined operators in order to generate a neighbor solution-representing permutation. This means that based on the given probability, LOSAP can switch between the two given operators anytime during its exploration. In this paper, many operators are proposed as the options for being the LOSAP's neighbor operators (note that the successes of randomly using one from two neighbor operators have been found in different algorithms, e.g., [10, 11]).

LOSAP generates a hybrid neighborhood structure between the first operator's neighborhood structure and the second operator's neighborhood structure. The hybridization is controlled by the probability of selecting the first neighbor operator as a LOSAP's input parameter (the probability of selecting the second neighbor operator, as a complement, is the unity minus the probability of selecting the first operator). Note that the higher the probability of selecting the first operator, the more likely the hybrid neighborhood structure is like the first operator's neighborhood structure. It equally means that the lower the probability of selecting the first operator, the more likely the hybrid neighborhood structure is like the second operator's neighborhood structure. At boundaries, the mentioned probability's values of 1.00 and 0.00 make the hybrid neighborhood structure be identical to the first operator's neighborhood structure and the second operator's neighborhood structure, respectively.

The probability of selecting the first neighbor operator is not the only LOSAP's input parameter. LOSAP has total five input parameters consisting of the perturbation operator, the scheduling direction, the ordered pair of the first and second neighbor operators, the probability of selecting the first neighbor, and the start operation-based permutation. LOSAP provides many options for setting each input parameter in order that LOSAP with proper parameter values can perform well for every single instance. Below, Section 3.1.1 describes Algorithm 1 as the decoding procedure used by LOSAP. In detail, Algorithm 1 is the procedure of transforming each solution-representing permutation generated by LOSAP into a semiactive schedule. Section 3.1.2 then describes Algorithm 2 as the procedure of LOSAP.

3.1.1. Decoding Procedure. LOSAP searches in a solution space of semiactive schedules, and it uses an operation-based permutation [40–43] to represent a semiactive schedule. An operation-based permutation has been used to represent a semiactive schedule in many researches, such as [20, 43, 44]. For an n-job/m-machine instance, an operation-based

Step 1. Receive an operation-based permutation and a scheduling direction (forward or backward) from LOSAP (Algorithm 2).
Step 2. If the scheduling direction received in Step 1 is forward, then the precedence relations of the operations of each job are unchanged. However, if it is backward, then the precedence relations of the operations must be reversed by using Steps 2.1 and 2.2.
Step 2.1. For each job J_i (where $i = 1, 2, \ldots, n$), let the operations $O_{i1}, O_{i2}, \ldots, O_{im}$ be renamed $O_{im}, O_{im-1}, \ldots, O_{i1}$, respectively.
Step 2.2. Assign the precedence relations of all operations O_{ij} (where $j = 1, 2, \ldots, m$) of each job J_i in ascending order of j values. Thus, 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. If the scheduling direction received in Step 1 is forward, then the operation-based permutation received in Step 1 is unchanged. However, if it is backward, then the order of all members in the operation-based permutation must be reversed. For example, the permutation (3, 2, 3, 1, 1, 2) with a backward scheduling direction must be changed into (2, 1, 1, 3, 2, 3).
Step 4. Transform the permutation taken from Step 3 by changing the number i in its j^{th} occurrence (from left to right) into the operation O_{ij} ($i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$). For example, the permutation (3, 2, 3, 1, 1, 2) must be transformed into (O_{31}, O_{21}, O_{32}, O_{11}, O_{12}, O_{22}).
Step 5. Transform the permutation taken from Step 4 into a semiactive schedule by using Steps 5.1 to 5.3.
Step 5.1. Let Φ_t be the partial schedule of the t scheduled operations. Thus, Φ_0 is empty. Now, let $t \longleftarrow 1$.
Step 5.2. Let $O_L \longleftarrow$ the leftmost as-yet-unscheduled operation in the permutation. Then, create Φ_t by scheduling O_L into Φ_{t-1} at its earliest possible start time on its preassigned machine (the earliest possible start time of O_L is the maximum between the finished time of its immediate-preceding operation in its job and the finished time of the current last-scheduled operation on its machine).
Step 5.3. If $t < mn$, then let $t \longleftarrow t + 1$ and repeat from Step 5.2. Otherwise, go to Step 6.
Step 6. If the scheduling direction received in Step 1 is forward, then return Φ_{mn} (which is a completed forward semiactive schedule) as the final result. However, if it is backward, then modify Φ_{mn} to satisfy the original precedence relations by using Steps 6.1 and 6.2.
Step 6.1. 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 6.2. Turn the schedule modified from Step 6.1 back to front in order that the last-finished operation in the schedule becomes the first-started operation, and so on. After that, let the schedule be started at the time 0. Then, return the schedule modified in this step (which is a completed backward semiactive schedule) as the final result.

Algorithm 1: The procedure of decoding an operation-based permutation into a semiactive schedule.

Step 1. Receive the input-parameter values from MUPLA (Algorithm 3) via Steps 1.1 to 1.5.
Step 1.1. Receive $PTBT \in \{n$-medium swap, n-large swap, n-medium inverse, n-large insert, n-medium insert$\}$.
Step 1.2. Receive $SD \in$ {forward, backward}.
Step 1.3. Receive $PNO \equiv (NO1, NO2) \in$ {(1-small inverse, 1-medium insert), (1-large swap, 1-large insert), (1-medium swap, 1-medium insert), (1-small swap, 1-small insert)}.
Step 1.4. Receive $PROB \in$ all real numbers within [0, 1].
Step 1.5. Receive $P \in$ all possible operation-based permutations (i.e., all permutations with m repetitions of the numbers 1, 2, ..., n).
Step 2. Generate an initial P_0 by using $PTBT$ on P.
Step 3. Execute Algorithm 1 by inputting SD and P_0 in order to receive S_0.
Step 4. Find a local optimal schedule by using Steps 4.1 to 4.4.
Step 4.1. Let $t_L \longleftarrow 0$.
Step 4.2. Randomly generate u from $U[0, 1)$. If $u \leq PROB$, then generate P_1 by using $NO1$ on P_0; otherwise, generate P_1 by using $NO2$ on P_0.
Step 4.3. Execute Algorithm 1 by inputting SD and P_1 in order to receive S_1.
Step 4.4. Update P_0, S_0, and t_L by using Steps 4.4.1 to 4.4.3.
Step 4.4.1. If $Makespan\ (S_1) < Makespan\ (S_0)$, then let $P_0 \longleftarrow P_1$ and $S_0 \longleftarrow S_1$, and repeat from Step 4.1.
Step 4.4.2. If $Makespan\ (S_1) = Makespan\ (S_0)$, then let $P_0 \longleftarrow P_1$ and $S_0 \longleftarrow S_1$, and repeat from Step 4.2.
Step 4.4.3. If $Makespan\ (S_1) > Makespan\ (S_0)$, then let $t_L \longleftarrow t_L + 1$. After that, if $t_L < (mn)^2$, then repeat from Step 4.2; otherwise, go to Step 5.
Step 5. Return P_0 and S_0 as the final (best-found) operation-based permutation and the final (best-found) schedule, respectively.

Algorithm 2: The procedure of LOSAP.

permutation is a permutation with m repetitions of the numbers 1, 2, ..., n. In the permutation, the number i (where $i = 1, 2, \ldots, n$) in its j^{th} occurrence from left to right (where $j = 1, 2, \ldots, m$) represents the operation O_{ij}. Then, a semiactive schedule is constructed by scheduling all operations in the order given by the permutation; in addition, each operation must be scheduled at its earliest possible start time on its preassigned machine. For example, the permutation (3, 2, 3, 1, 1, 2) represents the schedule in which the operations O_{31}, O_{21}, O_{32}, O_{11}, O_{12}, and O_{22} are sequentially scheduled at their earliest possible start times on their preassigned machines. The earliest possible start time of a specific operation is the

maximum between the finished time of its immediate-preceding operation in its job and the finished time of the current last-scheduled operation on its machine.

Algorithm 1 is the solution-decoding procedure used by LOSAP. As the options, it can transform an operation-based permutation into a forward semiactive schedule (i.e., a semiactive schedule constructed by the forward direction) or a backward semiactive schedule (i.e., a semiactive schedule constructed by the backward direction). Thus, Algorithm 1 requires assigning values of the two input parameters, i.e., an operation-based permutation and a scheduling direction. Remind that m and n represent the number of all machines and the number of all jobs, respectively, in the being-considered JSP instance. Thus, mn (i.e., m multiplied by n) represents the number of all operations.

As mentioned above, Algorithm 1 is the solution-decoding method used by LOSAP, and Algorithm 2 in Section 3.1.2 is the LOSAP's procedure. Thus, it can be said that Algorithm 1 is a component of Algorithm 2.

3.1.2. LOSAP Procedure. LOSAP, as shown in Algorithm 2, has the five input parameters whose values need to be assigned, i.e., *PTBT*, *SD*, $PNO \equiv (NO1, NO2)$, *PROB*, and *P*. The definitions of these LOSAP's input parameters are given below:

(i) *PTBT* and *P* stand for the perturbation operator and the start operation-based permutation, respectively. LOSAP uses *PTBT* on *P* in order to generate its initial best-found operation-based permutation.

(ii) *SD* stands for the scheduling direction (forward or backward) of all solutions generated by LOSAP. If *SD* is selected to be forward, LOSAP transforms each generated operation-based permutation into a forward schedule. Otherwise, LOSAP transforms each permutation into a backward schedule.

(iii) $PNO \equiv (NO1, NO2)$ is the ordered pair of the first neighbor operator (called $NO1$) and the second neighbor operator (called $NO2$).

(iv) *PROB* is the probability of selecting the first neighbor operator ($NO1$). Consequently, the probability of selecting the second neighbor operator ($NO2$) is the unity minus *PROB*.

Besides the definitions of the input parameters mentioned above, the other abbreviations used in LOSAP (i.e., Algorithm 2) are defined below:

(i) P_0 stands for the current best-found operation-based permutation. As mentioned above, the initial P_0 is generated by using *PTBT* on *P*.

(ii) S_0, which is decoded from P_0, stands for the current best-found schedule. In addition, *Makespan* (S_0) stands for the makespan of S_0.

(iii) P_1, which is generated by using $NO1$ or $NO2$ on P_0, stands for the current neighbor operation-based permutation.

(iv) S_1, which is decoded from P_1, stands for the current neighbor schedule. In addition, *Makespan* (S_1) stands for the makespan of S_1.

(v) m and n are the number of machines and the number of jobs, respectively, in the being-considered JSP instance. Thus, mn is the number of operations.

The LOSAP's procedure given in Algorithm 2 is efficient but simple. In brief, LOSAP starts its procedure by using *PTBT* on *P* to generate an initial P_0; after that, LOSAP transforms P_0 into S_0. Then, LOSAP starts its repeated loop by using *PROB* to randomly select either $NO1$ or $NO2$. LOSAP uses the selected operator (i.e., either $NO1$ or $NO2$) on P_0 to generate P_1, and LOSAP then transforms P_1 into S_1. If the criterion of accepting a new best-found permutation is satisfied, then LOSAP updates $P_0 \longleftarrow P_1$ and $S_0 \longleftarrow S_1$. Finally, LOSAP determines whether to continue the repeated loop's next round or stop its procedure. Note that S_0 and S_1 are always generated in the forward direction if *SD* is selected as forward at the beginning; otherwise, they are always generated in the backward direction.

In LOSAP, there are many optional operators for *PTBT* and *PNO*. These optional operators are modified from the common swap, insert, and inverse operators [29] (the definitions of the common operators are given in Section 2). In detail, the five LOSAP's options for *PTBT* consist of the n-medium swap operator, the n-large swap operator, the n-medium inverse operator, the n-large insert operator, and the n-medium insert operator. LOSAP also provides the four options for $PNO \equiv (NO1, NO2)$, consisting of (1-small inverse, 1-medium insert), (1-large swap, 1-large insert), (1-medium swap, 1-medium insert), and (1-small swap, 1-small insert). These optional operators are defined as follows. The number in front of the hyphen sign (-) indicates the number of repeated uses of the operator mentioned in back of the hyphen sign. For example, the 1-small swap operator is to use the small swap operator once on a permutation, and the n-medium inverse operator is to use the medium inverse operator n times on a permutation.

In addition to the above paragraph, the words *small*, *medium*, and *large* in the names of the optional operators are used to restrict the value of v in its distance from u as explained below (remind that u is a random integer within $[1, mn]$):

(i) For all operators with *small*, v is a random integer within $[u-4, u+4]$ (note that the small swap in [45] means the swap of two adjacent members, and it thus differs from the small swap in LOSAP).

(ii) For all operators with *medium*, v is a random integer within $[u-(mn/5), u+(mn/5)]$.

(iii) For all operators with *large*, v is a random integer within $[1, mn]$. It means that the operators with *large* are identical to the common operators.

After generating v successfully, its value must be verified whether it can be used or not. If the generated value of v is outside of $[1, mn]$ or equal to u, then it must be randomly regenerated within the given same range. This procedure must be repeated until receiving the value of v within $[1, mn]$ and unequal to u.

Remind that an addition of more optional operators into LOSAP may not always give a benefit for the two-level metaheuristic algorithm. Moreover, it sometimes makes the two-level metaheuristic algorithm harder to find a better solution. As shown in Algorithm 2, LOSAP does not include the n-small swap, n-small insert, and n-large inverse operators as the options for *PTBT*. The reason of excluding the n-small swap and n-small insert operators is that they make a too-small change into P; by using them, the two-level metaheuristic algorithm can hardly escape a local optimum. In contrast, a change from the n-large inverse operator is almost as large as a change from a re-initialization. For *PNO*, the 1-medium inverse and 1-large inverse operators are not used as the options because, as neighbor operators, they make a too-large change into P_0.

LOLA [9] and LOSAP both are lower-level algorithms of their own two-level metaheuristic algorithms; however, there are many differences between them. The LOLA's search ability is mainly based on its solution space of parameterized-active schedules. By contrast, LOSAP uses an ordinary solution space of semiactive schedules; thus, its search ability is mainly based on its probabilistic-based hybrid neighborhood structure. Most optional perturbation and neighbor operators of LOSAP are different from those of LOLA; the n-large insert operator is the only optional perturbation operator found in both LOLA and LOSAP. Another difference between LOLA and LOSAP is in their criteria of accepting a new best-found solution. LOLA accepts only a better neighbor solution, while LOSAP accepts a nonworsening neighbor solution. LOSAP uses this acceptance criterion to escape from a shoulder (i.e., a flat area of search space adjacent to a downhill edge [46]). In addition, LOSAP will not reset t_L to 0 for any sideways move in order to avoid an endless loop when finding a flat local minimum.

3.2. Proposed Upper-Level Algorithm. MUPLA is the upper-level algorithm of the proposed two-level algorithm. The purpose of MUPLA is to evolve the LOSAP's input-parameter values so that LOSAP can return its best performance on every single JSP instance. MUPLA is a population-based search algorithm specifically developed for being a parameter tuner. The MUPLA's population contains N combinations of the LOSAP's input-parameter values, i.e., $C_1(t)$, $C_2(t)$,..., $C_N(t)$. In short, let a *parameter-value combination* stand for a combination of the LOSAP's input-parameter values. In the population, each parameter-value combination is adjusted over iterations by the MUPLA's specific evolutionary process.

Let $C_i(t) \equiv (c_{1i}(t), c_{2i}(t), c_{3i}(t), c_{4i}(t), p_i(t))$ represent the i^{th} parameter-value combination (where $i = 1, 2, \ldots, N$) in the MUPLA's population at the t^{th} iteration. These $c_{1i}(t)$, $c_{2i}(t)$, $c_{3i}(t)$, and $c_{4i}(t)$ are real numbers representing the values of the perturbation operator (*PTBT*), the scheduling direction (*SD*), the ordered pair of the first and second neighbor operators (*PNO*), and the probability of selecting the first neighbor operator (*PROB*) of LOSAP, respectively. In addition, $p_i(t)$ represents the start operation-based permutation of LOSAP. Note that $p_i(t)$ is an operation-based permutation, not a real number like others. Table 1 presents the transformation from $c_{1i}(t)$, $c_{2i}(t)$, $c_{3i}(t)$, $c_{4i}(t)$, and $p_i(t)$ into the values of *PTBT*, *SD*, *PNO*, *PROB*, and P of LOSAP, respectively.

The abbreviations used in MUPLA (i.e., Algorithm 3) are defined below:

(i) *Score* $(C_i(t))$ stands for the performance score of $C_i(t)$. Note that the lower the performance score, the better the performance. Between any two parameter-value combinations, the combination with a lower performance score is the better one.

(ii) $P_{fi}(t)$ stands for the final (best-found) operation-based permutation of the LOSAP using the input-parameter values decoded from $C_i(t)$.

(iii) $S_{fi}(t)$ stands for the final (best-found) schedule of the LOSAP using the input-parameter values decoded from $C_i(t)$. In addition, *Makespan* $(S_{fi}(t))$ stands for the makespan of $S_{fi}(t)$.

(iv) $C_{best} \equiv (c_{1best}, c_{2best}, c_{3best}, c_{4best}, p_{best})$ stands for the best parameter-value combination ever-found by the population. In addition, *Score* (C_{best}) stands for the performance score of C_{best} (note that C_{best} in definition is similar to the global best position of PSO [32]).

(v) S_{best} stands for the best schedule ever-found by the population.

The procedure of MUPLA is presented in detail in Algorithm 3. In addition, it is also presented in a form of flow chart in Figure 1. In brief, MUPLA starts its procedure by assigning $t \longleftarrow 1$ and *Score* $(C_{best}) \longleftarrow +\infty$. Then, MUPLA randomly generates $C_i(t)$, where $i = 1, 2, \ldots, N$. After that, MUPLA processes its repeated loop as follows. For each $C_i(t)$, MUPLA decodes it into the LOSAP input-parameter values and then runs the LOSAP using these input-parameter values to receive $P_{fi}(t)$ and $S_{fi}(t)$; then, MUPLA assigns *Score* $(C_i(t)) \longleftarrow$ *Makespan* $(S_{fi}(t))$. If MUPLA finds any $C_i(t)$ whose *Score* $(C_i(t))$ is less than or equal to *Score* (C_{best}), then it updates $C_{best} \longleftarrow C_i(t)$, *Score* $(C_{best}) \longleftarrow$ *Score* $(C_i(t))$, and $S_{best} \longleftarrow S_{fi}(t)$. After that, MUPLA generates $C_i(t+1)$, where $i = 1, 2, \ldots, N$, by using its specific evolutionary process (as shown in Step 3 of Algorithm 3), and it then assigns $t \longleftarrow t+1$. Finally, MUPLA determines whether to continue its repeated loop's next round or stop its procedure.

A main difference of MUPLA from the other population-based algorithms such as [32–35] is in its specific evolutionary process (i.e., the procedure of adjusting its population) given by the equation in Step 3.3. In the equation in Step 3.3, each $c_{ji}(t+1)$ is updated from $c_{ji}(t)$ by the sum of two opposite-direction vectors. The first vector is toward the best-found value, whereas the second vector is away from. The equation in Step 3.3 used in MUPLA is slightly modified from that in UPLA [9]. The modification is to reduce the first vector's maximum magnitude by a half (from 0.05 to 0.025) in cases that $c_{ji}(t)$ differs from c_{jbest}. The purpose of this modification is to delay the population from getting stuck in a local optimum. In a preliminary experiment, the proposed two-level metaheuristic algorithm performed better on average after reducing the first vector's maximum magnitude as mentioned.

Table 1: Transformation from $C_i(t)$ into the LOSAP's input-parameter values.

MUPLA's $C_i(t)$	LOSAP's input-parameter values
$c_{1i}(t) \in \mathbb{R}$	$PTBT = \begin{cases} n\text{-medium swap} & c_{1i}(t) < 0.20 \\ n\text{-large swap} & 0.20 \le c_{1i}(t) < 0.40 \\ n\text{-medium inverse} & 0.40 \le c_{1i}(t) < 0.60 \\ n\text{-large insert} & 0.60 \le c_{1i}(t) < 0.80 \\ n\text{-medium insert} & c_{1i}(t) \ge 0.80 \end{cases}$
$c_{2i}(t) \in \mathbb{R}$	$SD = \begin{cases} \text{forward} & c_{2i}(t) < 0.50 \\ \text{backward} & c_{2i}(t) \ge 0.50 \end{cases}$
$c_{3i}(t) \in \mathbb{R}$	$PNO \equiv (NO1, NO2) = \begin{cases} (1\text{-small inverse}, 1\text{-medium insert}) & c_{3i}(t) < 0.25 \\ (1\text{-large swap}, 1\text{-large insert}) & 0.25 \le c_{3i}(t) < 0.50 \\ (1\text{-medium swap}, 1\text{-medium insert}) & 0.50 \le c_{3i}(t) < 0.75 \\ (1\text{-small swap}, 1\text{-small insert}) & c_{3i}(t) \ge 0.75 \end{cases}$
$c_{4i}(t) \in \mathbb{R}$	$PROB = \begin{cases} 0.00 & c_{4i}(t) < 0.00 \\ c_{4i}(t) & 0.00 \le c_{4i}(t) < 1.00 \\ 1.00 & c_{4i}(t) \ge 1.00 \end{cases}$
$p_i(t) \in$ all possible operation-based permutations	$P = p_i(t)$

Step 1. Let $t \longleftarrow 1$ and $Score(C_{best}) \longleftarrow +\infty$. Generate each $C_i(t) \equiv (c_{1i}(t), c_{2i}(t), c_{3i}(t), c_{4i}(t), p_i(t))$ (where $i = 1, 2, \ldots, N$) by randomly generating $c_{ji}(t)$ from $U[0, 1)$ (where $j = 1, 2, 3, 4$) and randomly generating $p_i(t)$ from any possible operation-based permutation.
Step 2. Evaluate $Score(C_i(t))$, and update C_{best} and S_{best} by using Steps 2.1 to 2.5.
Step 2.1. Let $i \longleftarrow 1$.
Step 2.2. Transform $C_i(t)$ into the values of $PTBT$, SD, PNO, $PROB$, and P of LOSAP by the relationships shown in Table 1.
Step 2.3. Execute Algorithm 2 (LOSAP) by inputting the $PTBT$, SD, PNO, $PROB$, and P taken from Step 2.2 in order to receive $P_{fi}(t)$ and $S_{fi}(t)$. Then, let $Score(C_i(t)) \longleftarrow Makespan(S_{fi}(t))$.
Step 2.4. If $Score(C_i(t)) \le Score(C_{best})$, then let $C_{best} \longleftarrow C_i(t)$, $Score(C_{best}) \longleftarrow Score(C_i(t))$, and $S_{best} \longleftarrow S_{fi}(t)$.
Step 2.5. If $i < N$, then let $i \longleftarrow i + 1$ and repeat from Step 2.2; otherwise, go to Step 3.
Step 3. Update $C_i(t+1)$, where $i = 1, 2, \ldots, N$, by using Steps 3.1 to 3.4.
Step 3.1. Let $i \longleftarrow 1$.
Step 3.2. If $t \bmod 1000 = 0$, then randomly generate $p_i(t+1)$ from any possible operation-based permutation; otherwise, let $p_i(t+1) \longleftarrow P_{fi}(t)$.
Step 3.3. If $t \bmod 25 = 0$, then randomly generate $c_{ji}(t+1)$ from $U[0, 1)$ (where $j = 1, 2, 3, 4$); otherwise, generate $c_{ji}(t+1)$ by the following equation. Let u_1 and u_2 be randomly generated from $U[0, 1)$. $c_{ji}(t+1) = \begin{cases} c_{ji}(t) + 0.025u_1 - 0.01u_2 & c_{ji}(t) < c_{jbest} \\ c_{ji}(t) - 0.025u_1 + 0.01u_2 & c_{ji}(t) > c_{jbest} \\ c_{ji}(t) + 0.01u_1 - 0.01u_2 & c_{ji}(t) = c_{jbest}. \end{cases}$
Step 3.4. If $i < N$, then let $i \longleftarrow i + 1$ and repeat from Step 3.2; otherwise, go to Step 4.
Step 4. If the stopping criterion is not met, then let $t \longleftarrow t + 1$ and repeat from Step 2. Otherwise, return S_{best} as the final result.

Algorithm 3: The procedure of MUPLA.

Besides the modification in the equation in Step 3.3, there are larger other changes of MUPLA from UPLA [9]. One change is that, unlike UPLA, each $C_i(t)$ of the MUPLA's population includes a start operation-based permutation $p_i(t)$. By this change mentioned, MUPLA can add a multistart property into LOSAP. Consequently, the combination of MUPLA and LOSAP becomes a multistart iterated local search algorithm. This is a large upgrade because the combination of UPLA and LOLA in [9] is just an iterated local search algorithm. Another change is in its criterion of accepting a new best-found parameter-value combination. UPLA accepts only a better parameter-value combination, while MUPLA accepts a nonworsening parameter-value combination.

4. Results and Discussions

The performance of the proposed two-level metaheuristic algorithm was evaluated via the experiment conducted in this research. Section 4 then presents results of the

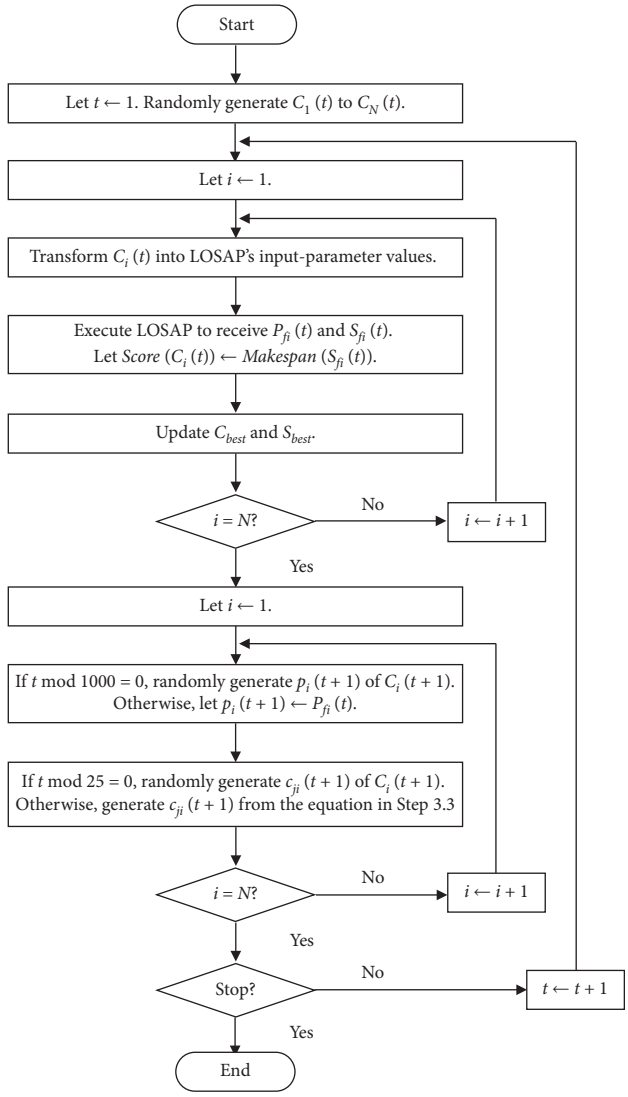

Figure 1: Flow chart of MUPLA.

experiment. In this section, let MUPLA stand for the whole two-level metaheuristic algorithm (i.e., the MUPLA combined with LOSAP). The reason is that MUPLA uses LOSAP as its component when it solves JSP. For a comparison purpose, the MUPLA's results were compared to those of TS, GA, and UPLA taken from [5, 6], and [9], respectively. Let TS stand for the taboo search algorithm developed by Nowicki and Smutnicki [5], and let GA stand for the genetic algorithm developed by Piroozfard et al. [6]. In addition, let UPLA in this section stand for the UPLA combined with LOLA.

The performance of MUPLA was evaluated on 53 benchmark instances. These 53 instances consisted of the ft01, ft10, and ft20 instances from [47], the la01 to la40 instances from [48], and the orb01 to orb10 instances from [49]. The 53 mentioned instances were chosen because they covered all instances used by [5, 6, 9] in their experiments. In detail, the TS's performance was evaluated in [5] on the 43 first-mentioned instances, i.e., ft06, ft10, ft20, and all 40 la instances. The GA's performance was evaluated in [6] on the 42 instances, i.e., ft06, ft20, and all 40 la instances. In addition, the UPLA's performance was evaluated in [9] on all the 53 instances.

There were two main objectives of the experiment. The first objective was to evaluate the best performance of which MUPLA was capable on its solution quality without a computational time's limitation. To achieve this objective, MUPLA was run on an extremely long computational time (i.e., 5,000 iterations in this paper) for each trial unless it could find the known optimal solution. The reason was to ensure that MUPLA already reached the best solution as possible as it could. The second objective of the experiment was to evaluate the MUPLA's search performance over iterations. To do so, the solution's convergence rates of MUPLA were plotted. A finding from the solution's convergence rates was to identify a proper maximum iteration for MUPLA. It was very important because a proper maximum iteration could balance the solution quality and the consumed computational time. Remind that the higher the number of iterations used, the higher the computational time consumed.

The parameter settings of MUPLA used in the experiment are shown below:

(i) The population of MUPLA consisted of three parameter-value combinations (i.e., $N = 3$ in Algorithm 3).

(ii) The stopping criterion of MUPLA was that either the 5,000th iteration (i.e., $t = 5{,}000$ in Algorithm 3) was reached or the known optimal solution value was found (note that *solution* and *solution value* in this paper represent *schedule* and *makespan*, respectively).

(iii) MUPLA was coded in C# and executed on an Intel® CoreTM i5 CPU processor M580 @ 2.67 GHz with RAM of 6 GB (2.3 GB usable).

(iv) For each instance, MUPLA was executed five trials with different random-seed numbers.

The results of the experiment using the above settings are presented in Table 2. In the table, the MUPLA's results on each instance consisted of the best-found solution value over five trials, the average of the first trial's best-found solution value to the fifth trial's best-found solution value, the average number of used iterations, and the average computational time consumed. In the purpose of comparison, Table 2 presents the best-found solution values of TS, GA, and UPLA taken from [5, 6], and [9], respectively. Remind that in the table, the TS's results are shown on only the ft06, ft10, ft20, and all 40 la instances, and the GA's results are shown on only the ft06, ft20, and all 40 la instances.

The abbreviations used in Table 2 are defined as follows:

(i) *Instance* and *Opt* represent the name of each instance and its known optimal solution value given by literature (e.g., [2]), respectively.

(ii) In each instance, *Best* represents the best-found solution value of each algorithm. *Best* of MUPLA was taken from the five trials in this experiment.

Table 2: Experiment's results on benchmark instances.

Instance	Opt	TS [5]		GA [6]		UPLA [9]		MUPLA					
		Best	%BD	Best	%BD	Best	%BD	Best	%BD	Avg	%AD	Avg iterations	Avg CPU time
ft06	55	55	0.00	55	0.00	55	0.00	55	0.00	55	0.00	1	0.1
ft10	930	930	0.00	n/a	n/a	930	0.00	930	0.00	930	0.00	97	315
ft20	1165	1165	0.00	1165	0.00	1165	0.00	1165	0.00	1165	0.00	335	1,890
la01	666	666	0.00	666	0.00	666	0.00	666	0.00	666	0.00	1	1
la02	655	655	0.00	655	0.00	655	0.00	655	0.00	655	0.00	5	2
la03	597	597	0.00	597	0.00	597	0.00	597	0.00	597	0.00	27	10
la04	590	590	0.00	590	0.00	590	0.00	590	0.00	590	0.00	6	2
la05	593	593	0.00	593	0.00	593	0.00	593	0.00	593	0.00	1	0.4
la06	926	926	0.00	926	0.00	926	0.00	926	0.00	926	0.00	1	2
la07	890	890	0.00	890	0.00	890	0.00	890	0.00	890	0.00	1	2
la08	863	863	0.00	863	0.00	863	0.00	863	0.00	863	0.00	1	2
la09	951	951	0.00	951	0.00	951	0.00	951	0.00	951	0.00	1	2
la10	958	958	0.00	958	0.00	958	0.00	958	0.00	958	0.00	1	2
la11	1222	1222	0.00	1222	0.00	1222	0.00	1222	0.00	1222	0.00	1	4
la12	1039	1039	0.00	1039	0.00	1039	0.00	1039	0.00	1039	0.00	1	8
la13	1150	1150	0.00	1150	0.00	1150	0.00	1150	0.00	1150	0.00	1	6
la14	1292	1292	0.00	1292	0.00	1292	0.00	1292	0.00	1292	0.00	1	6
la15	1207	1207	0.00	1207	0.00	1207	0.00	1207	0.00	1207	0.00	1	7
la16	945	945	0.00	946	0.11	945	0.00	945	0.00	945	0.00	89	294
la17	784	784	0.00	784	0.00	784	0.00	784	0.00	784	0.00	10	33
la18	848	848	0.00	848	0.00	848	0.00	848	0.00	848	0.00	8	24
la19	842	842	0.00	842	0.00	842	0.00	842	0.00	842	0.00	45	149
la20	902	902	0.00	907	0.55	902	0.00	902	0.00	902	0.00	312	1,073
la21	1046	1047	0.10	1050	0.38	1052	0.57	1046	0.00	1046.6	0.06	3,478 (1,644)	64,880 (30,668)
la22	927	927	0.00	927	0.00	927	0.00	927	0.00	927	0.00	72	1,439
la23	1032	1032	0.00	1032	0.00	1032	0.00	1032	0.00	1032	0.00	2	25
la24	935	939	0.43	943	0.86	941	0.64	935	0.00	937.4	0.26	4,673 (1,186)	84,121 (21,350)
la25	977	977	0.00	984	0.72	982	0.51	977	0.00	977	0.00	954	15,827
la26	1218	1218	0.00	1218	0.00	1218	0.00	1218	0.00	1218	0.00	1	82
la27	1235	1236	0.08	1255	1.62	1256	1.70	1235	0.00	1235.4	0.03	3,702 (3,266)	220,382 (194,427)
la28	1216	1216	0.00	1217	0.08	1216	0.00	1216	0.00	1216	0.00	28	1,972
la29	1152	1160	0.69	1179	2.34	1191	3.39	1163	0.95	1163.8	1.02	5,000 (2,041)	318,615 (130,059)
la30	1355	1355	0.00	1355	0.00	1355	0.00	1355	0.00	1355	0.00	1	123
la31	1784	1784	0.00	1784	0.00	1784	0.00	1784	0.00	1784	0.00	1	306
la32	1850	1850	0.00	1850	0.00	1850	0.00	1850	0.00	1850	0.00	1	172
la33	1719	1719	0.00	1719	0.00	1719	0.00	1719	0.00	1719	0.00	1	313
la34	1721	1721	0.00	1721	0.00	1721	0.00	1721	0.00	1721	0.00	1	448
la35	1888	1888	0.00	1888	0.00	1888	0.00	1888	0.00	1888	0.00	1	393
la36	1268	1268	0.00	1294	2.05	1278	0.79	1268	0.00	1268	0.00	1,104	85,418
la37	1397	1407	0.72	1418	1.50	1407	0.72	1397	0.00	1397	0.00	823	60,481
la38	1196	1196	0.00	1222	2.17	1215	1.59	1196	0.00	1199	0.25	3,985 (2,282)	296,821 (169,974)
la39	1233	1233	0.00	1249	1.30	1250	1.38	1233	0.00	1233	0.00	265	18,057
la40	1222	1229	0.57	1233	0.90	1229	0.57	1224	0.16	1224	0.16	5,000 (1,678)	355,968 (119,463)
orb01	1059	n/a	n/a	n/a	n/a	1059	0.00	1059	0.00	1059	0.00	307	965
orb02	888	n/a	n/a	n/a	n/a	889	0.11	888	0.00	888.2	0.02	2,438 (1,443)	7,622 (4,511)
orb03	1005	n/a	n/a	n/a	n/a	1005	0.00	1005	0.00	1005	0.00	903	2,500
orb04	1005	n/a	n/a	n/a	n/a	1005	0.00	1005	0.00	1005	0.00	257	840
orb05	887	n/a	n/a	n/a	n/a	889	0.23	887	0.00	887	0.00	2,064	6,237
orb06	1010	n/a	n/a	n/a	n/a	1013	0.30	1010	0.00	1010.4	0.04	2,912 (2,033)	9,610 (6,709)
orb07	397	n/a	n/a	n/a	n/a	397	0.00	397	0.00	397	0.00	61	216
orb08	899	n/a	n/a	n/a	n/a	899	0.00	899	0.00	899	0.00	184	594
orb09	934	n/a	n/a	n/a	n/a	934	0.00	934	0.00	934	0.00	127	515
orb10	944	n/a	n/a	n/a	n/a	944	0.00	944	0.00	944	0.00	58	183

Bests of TS, GA, and UPLA were taken from [5, 6], and [9], respectively. For each algorithm, *%BD* represents the percent deviation of *Best* from *Opt*.

(iii) In each instance, *Avg* represents the average of the first trial's best-found solution value to the fifth trial's best-found solution value of MUPLA. Then, *%AD* represents the percent deviation of *Avg* from *Opt*.

(iv) *Avg Iterations* and *Avg CPU Time* stand for the average number of iterations and the average

computational time (in second), respectively, consumed by MUPLA until the stopping criterion is met.

(v) In parentheses, *Avg Iterations* and *Avg CPU Time* provide the average number of iterations and the approximate average computational time (in second), respectively, consumed by MUPLA until no further improvement. The information occurs only when at least one trial of MUPLA could not find the known optimal solution before reaching the 5,000th iteration.

Based on the results in Table 2, Section 4.1 evaluates the best search performance of which MUPLA was capable without a computational time's limitation. Section 4.2 then evaluates the MUPLA's search performance over iterations.

4.1. Evaluation on Search Performance without Computational Time's Limitation. As just mentioned, Section 4.1 aims at revealing the best search performance of which MUPLA was capable on its solution quality without a computational time's restriction. To achieve the aim, MUPLA was executed on an extremely long computational time (i.e., 5,000 iterations) unless MUPLA could find the known optimal solution. The purpose of this setting was to ensure that MUPLA already reached its best solution as possible as it could. In the performance evaluation, the %BDs of MUPLA were compared to the %BDs of TS [5], GA [6], and UPLA [9] shown in Table 2. Remind that the %BDs of TS, GA, and UPLA were calculated from the best-found solution values given by their original articles. The use of these %BDs of TS, GA, and UPLA can be defined as a limitation of this experiment because TS, GA, and UPLA might find improving solutions if they were run on more computational times than those used in their articles. On the other hand, they might not find any improving solution although much more computational times were provided; it usually happens to any metaheuristic algorithm when its search gets stuck in a local optimum.

Discussions on the %BDs shown in Table 2 are given hereafter along with using one-sided paired t-tests to compare the mean %BDs. Let the significance level (α) be equal to 0.1. As shown in Table 2, MUPLA performed better than TS [5] in terms of %BD (remind that the performances were compared on only the 43 instances). Of the total 43 instances, the average %BD of MUPLA was 0.03%, while the average %BD of TS was 0.06%. MUPLA returned better %BDs than TS on five instances (i.e., la21, la24, la27, la37, and la40), while TS returned a better %BD than MUPLA on only one instance (i.e., la29). On the 37 remaining instances, they both returned equal %BDs. In addition, there were 41 instances where MUPLA returned %BDs of 0.00%, while there were only 37 instances where TS returned %BDs of 0.00%. It means that from all 43 instances, MUPLA found optimal solutions on 41 instances, while TS found optimal solutions on only 37 instances. With $\alpha = 0.1$, an one-sided paired t-test indicated that the mean %BD of MUPLA was significantly better than the mean %BD of TS (p value = 0.066).

MUPLA also outperformed GA [6] in terms of %BD (remind that the performances were compared on only the 42 instances). Over the total 42 instances, the average %BD of MUPLA was 0.03%, while the average %BD of GA was 0.35%. MUPLA returned better %BDs than GA on 13 instances, and they both returned equal %BDs on the 29 remaining instances. This means that GA could not return a better %BD than MUPLA on any instance. Moreover, MUPLA returned %BDs of 0.00% on 40 instances, while GA returned %BDs of 0.00% on only 29 instances. This means that from the 42 instances, MUPLA found the optimal solutions on 40 instances, while GA found the optimal solutions on only 29 instances. In detail, MUPLA failed to find optimal solutions on la29 and la40, while GA failed on la29, la40, and the 11 other instances. Although they both failed on la29 and la40, MUPLA performed much better on these two instances. On la29 and la40, respectively, the MUPLA's %BDs were 0.95% and 0.16%, while the GA's %BDs were 2.34% and 0.90%. An one-sided paired t-test with $\alpha = 0.1$ indicated that the mean %BD of MUPLA was significantly better than the mean %BD of GA (p value = 0.001).

Based on %BDs in Table 2, MUPLA outperformed UPLA [9]. From the total 53 instances, the average %BD of MUPLA was 0.02%, while the average %BD of UPLA was 0.24%. MUPLA returned better %BDs than UPLA on 13 instances, and they both returned the same %BDs on 40 instances. It means that there were no instances where UPLA returned better %BDs than MUPLA. In addition, MUPLA returned the %BDs of 0.00% on 51 instances, while UPLA returned the %BDs of 0.00% on only 40 instances. It means that MUPLA could find optimal solutions on 51 instances, while UPLA could find optimal solutions on only 40 instances. In detail, MUPLA failed to find optimal solutions on only la29 and la40, while UPLA failed on la29, la40, and the 11 other instances. Based on the results from Table 2, an one-sided paired t-test with $\alpha = 0.1$ indicated that the mean %BD of MUPLA was significantly better than the mean %BD of UPLA (p value = 0.002).

A summary of the above discussions on %BDs is given below:

(i) When compared with TS [5] on the 43 instances, MUPLA returned better %BDs on five instances, while TS returned a better %BD on only one instance. The average %BD of MUPLA was 0.03%, while the average %BD of TS was 0.06%.

(ii) When compared with GA [6] on the 42 instances, MUPLA returned better %BDs on 13 instances, while GA could not return a better %BD on any instance. The average %BD of MUPLA was 0.03%, while the average %BD of GA was 0.35%.

(iii) When compared with UPLA [9] on the total 53 instances, MUPLA returned better %BDs on 13 instances, while UPLA could not return a better %BD on any instance. The average %BD of MUPLA was 0.02%, while the average %BD of UPLA was 0.24%.

(iv) Based on the one-sided paired t-tests with $\alpha = 0.1$, the mean %BD of MUPLA was significantly better than the mean %BDs of the other algorithms (with p

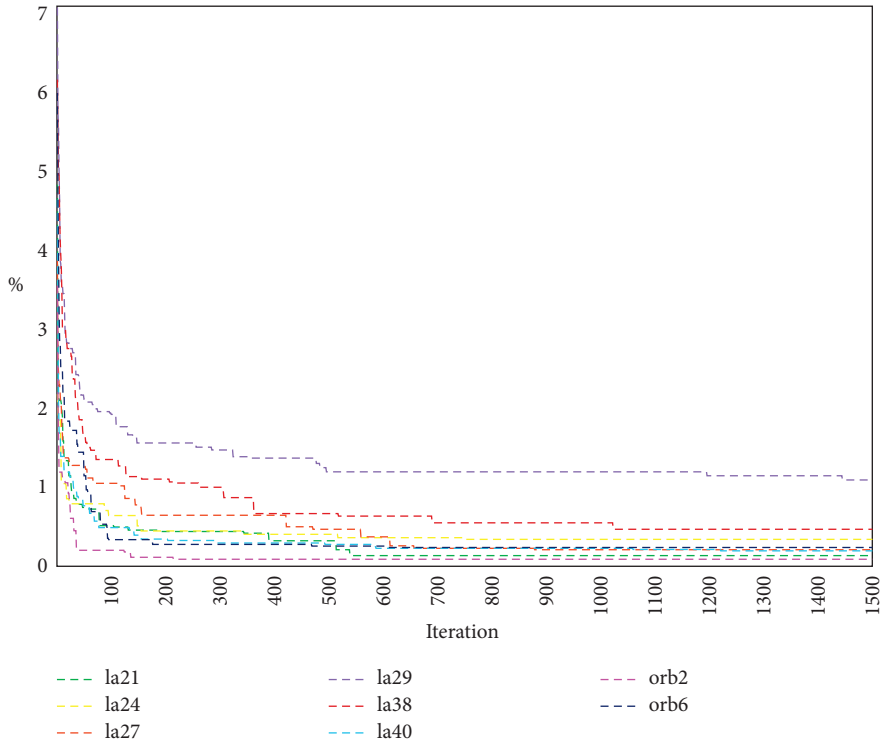

Figure 2: %AD-over-iteration plots.

values of 0.066 for TS, 0.001 for GA, and 0.002 for UPLA).

In discussion on %ADs, Table 2 shows that over all 53 instances, MUPLA returned %ADs of 0.00% on 45 instances. It means that MUPLA found the optimal solution by every single trial for the 45 instances. On the eight remaining instances, %ADs of MUPLA were also very low. Notice that the highest %AD, belonging to la29, was only 1.02%. This can be interpreted that MUPLA could perform very well in overall, not only in its best trial. When compared with %BDs, %ADs were equal or almost equal to %BDs for all instances. This emphasizes that MUPLA kept its high performance with good consistency from one trial to another.

4.2. Evaluation on Search Performance over Iterations. In this section, the solution's convergence rates of MUPLA were plotted in order to evaluate the MUPLA's search performance over iterations (remind that a solution's convergence rate is usually presented by a scatter plot showing the relationship between the solution quality and the number of iterations used). As shown in Figures 2 and 3 respectively, *%AD-over-iteration plots* (i.e., the plots of %ADs over iterations) and *%BD-over-iteration plots* (i.e., the plots of %BDs over iterations) were drawn for the eight instances, i.e., la21, la24, la27, la29, la38, la40, orb2, and orb6. These eight instances were selected because their %ADs in Table 2 were greater than 0% (it means that for each mentioned instance, at least one trial of MUPLA could not find the known optimal solution). The total number of iterations in each figure was only 1,500 iterations (not 5,000 iterations) because %ADs and %BDs, in general, were changed hardly after the 1,500th iteration.

To simplify the analysis, the pieces of information in Figures 2 and 3 were combined altogether and then put into Figure 4. At each iteration, the %ADs and %BDs on eight instances from Figures 2 and 3 were averaged into the avg %AD and avg %BD, respectively, in Figure 4. They resulted in the *avg-%AD-over-iteration plot* and the *avg-%BD-over-iteration plot* in Figure 4. For a comparison purpose, Figure 4 also presents the plots of *TS's final avg %BD*, *GA's final avg %BD*, and *UPLA's final avg %BD*. These plots represent the average %BDs of the final solutions on eight instances of TS, GA, and UPLA (i.e., 0.31%, 1.38%, and 1.41%, respectively). They were used to identify the lowest numbers of iterations of which MUPLA returned better solutions than the final solutions of TS, GA, and UPLA.

In Figure 4, the avg-%AD-over-iteration and avg-%BD-over-iteration plots could be divided into three periods based on their reduction rates. The first period roughly started from the first iteration to the 160th iteration, where the avg %AD and avg %BD were reduced sharply. The second period roughly started from the 160th iteration to the 700th iteration, where the avg %AD and avg %BD were reduced relatively quickly but slower than the first period's rate. The third period roughly started from the 700th iteration onwards, where the avg %AD and avg %BD were reduced very slowly. Since the higher number of iterations results in the higher computational time, the proper maximum iteration can help MUPLA to balance the solution quality and the computational time. Based on the two given plots, the MUPLA's maximum

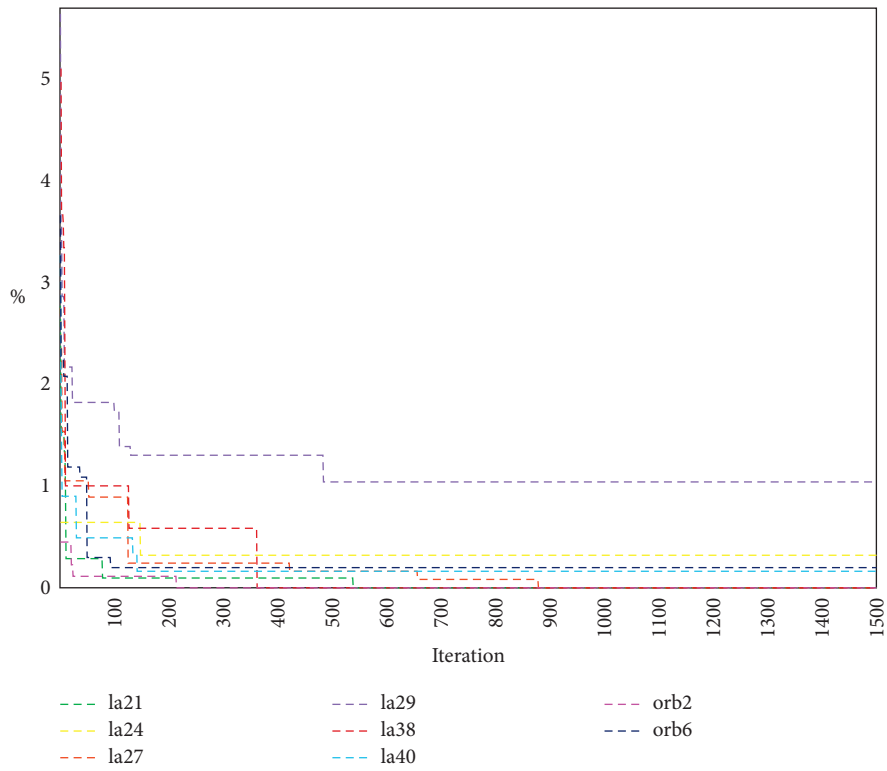

Figure 3: %BD-over-iteration plots.

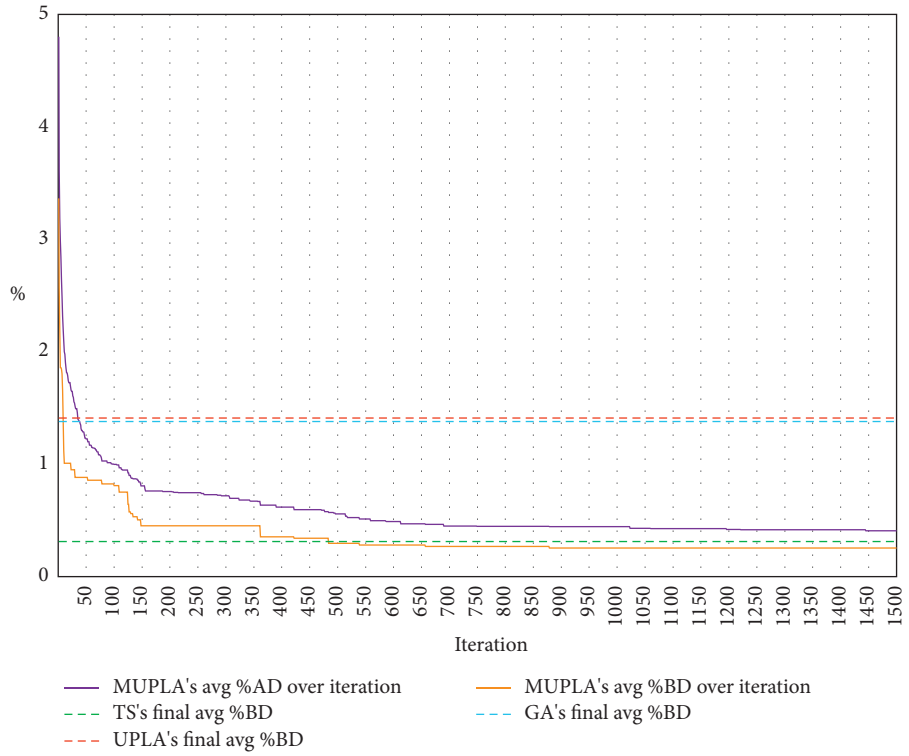

Figure 4: Avg-%AD-over-iteration plot and avg-%BD-over-iteration plot of MUPLA against final avg %BDs of TS, GA, and UPLA.

iteration should be set within the 160^{th} to 700^{th} iterations. At extremes, the 160^{th} iteration should be used when highly concerned on the computational time, while the 700^{th} iteration should be used when highly concerned on the solution quality. For any other point within the 160^{th} to 700^{th} iterations, the higher possibility to find a better

solution must be traded off for the greater number of iterations used.

For more analysis on Figure 4, the avg-%AD-over-iteration and avg-%BD-over-iteration plots were compared to the final avg %BDs of TS, GA, and UPLA. For each algorithm, the final avg %BD is the average of the %BDs of the final solutions on eight instances taken from Table 2. In Figure 4, the avg-%BD-over-iteration plot of MUPLA was lower than the final avg %BDs of UPLA, GA, and TS at the eighth, ninth, and 484^{th} iterations, respectively. It means that the MUPLA's best trial (among the total five trials) at its eighth, ninth, and 484^{th} iterations provided better solutions than the final solutions of UPLA, GA, and TS, respectively. Moreover, the avg-%AD-over-iteration plot was lower than the final avg %BDs of UPLA and GA at the 35^{th} and 39^{th} iterations, respectively. It can be interpreted roughly that MUPLA could usually return the better solutions than the final solutions of UPLA and GA within the first 40 iterations. The findings mentioned in this paragraph fulfilled those of the previous paragraph. One finding was that MUPLA could return acceptable-quality solutions within its first 10 iterations and then high-quality solutions within its first 500 iterations. Moreover, MUPLA was still able to find improving solutions after the 500^{th} iteration.

In addition, Figure 4 also reveals an enhancement percentage of the LOSAP's search performance received from MUPLA. Remind that at the MUPLA's first iteration (as the start), the LOSAP's input-parameter values were generated full-randomly. It means that at the MUPLA's first iteration, LOSAP performed its own search performance without any support from the upper-level algorithm yet. In Figure 4, the avg %AD at the first iteration was 4.80%. After passing through the MUPLA's evolutionary process to the 100^{th} iteration, the avg %AD became 1.00%. At the 500^{th} iteration, the avg %AD was then reduced to 0.56%. The improvement of the avg %AD has been a strong evidence of the enhancement in the LOSAP's search performance received from MUPLA. This can be concluded in the same way when explaining by the avg-%BD-over-iteration plot.

5. Conclusion

The proposed two-level metaheuristic algorithm is composed of the upper-level algorithm and the lower-level algorithm named MUPLA and LOSAP, respectively. In the upper level, MUPLA is a population-based search algorithm developed for controlling the LOSAP's input parameters. In the lower level, LOSAP is a local search algorithm with a probabilistic-based hybrid neighborhood structure. The work relation between MUPLA and LOSAP is given in brief as follows: MUPLA starts a repeated loop by generating the LOSAP's input-parameter values; then, LOSAP uses these input-parameter values to return its best-found JSP solution. The best-found JSP solution becomes a feedback sent back to MUPLA for improving the LOSAP's input-parameter values. MUPLA then starts the next round of the repeated loop. The MUPLA combined with LOSAP performs like an adaptive multistart iterated local search algorithm. The experiment's results indicated that the proposed two-level metaheuristic algorithm (i.e., the MUPLA combined with LOSAP) outperformed its original variant (i.e., the UPLA combined with LOLA) and the two other high-performing algorithms in terms of solution quality. Based on the convergence rates, the maximum iteration of the two-level metaheuristic algorithm was recommended to be set within the 160^{th} to the 700^{th} iterations. A future study should be conducted to enhance the performance of the two-level metaheuristic algorithm for JSP and related other problems. Another interesting future study is to modify MUPLA (uncombined with LOSAP) for being a lower-level algorithm. Consequently, the two-level metaheuristic algorithm using MUPLA in both levels will be possibly developed.

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 acknowledges partial financial support for publication from the Thai-Nichi Institute of Technology, Thailand.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] E. Nowicki and C. Smutnicki, "A fast taboo search algorithm for the job shop problem," *Management Science*, vol. 42, no. 6, pp. 797–813, 1996.

[6] H. Piroozfard, K. Y. Wong, and A. Hassan, "A hybrid genetic algorithm with a knowledge-based operator for solving the job shop scheduling problems," *Journal of Optimization*, vol. 2016, Article ID 7319036, 13 pages, 2016.

[7] 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.

[8] 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.

[9] P. Pongchairerks, "A two-level metaheuristic algorithm for the job-shop scheduling problem," *Complexity*, vol. 2019, Article ID 8683472, 11 pages, 2019.

[10] Q. Luo, Y. Zhou, J. Xie, M. Ma, and L. Li, "Discrete bat algorithm for optimal problem of permutation flow shop scheduling," *The Scientific World Journal*, vol. 2014, Article ID 630280, 15 pages, 2014.

[11] M. N. Janardhanan, Z. Li, P. Nielsen, and Q. Tang, "Artificial bee colony algorithms for two-sided assembly line worker assignment and balancing problem," in *Advances in Intelligent Systems and Computing*, vol. 620, pp. 11–18, Springer, Cham, Switzerland, 2018.

[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] 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*, pp. 21–31, Porto, Portugal, December 2001.

[14] S. Luke and A. K. A. Talukder, "Is the meta-EA a viable optimization method?" in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, pp. 1533–1540, Amsterdam, Netherlands, July 2013.

[15] S. Wink, T. Bäck, and M. Emmerich, "A meta-genetic algorithm for solving the capacitated vehicle routing problem," in *Proceedings of the 2012 IEEE Congress on Evolutionary Computation*, pp. 1–8, Brisbane, Australia, June 2012.

[16] 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.

[17] T. Yamada and R. Nakano, "Job-shop scheduling," in *Genetic Algorithms in Engineering Systems*, A. M. S. Zalzala and P. J. Fleming, Eds., pp. 134–160, Institution of Electrical Engineers, London, UK, 1997.

[18] T. Yamada and R. Nakano, "A fusion of crossover and local search," in *Proceedings of the IEEE International Conference on Industrial Technology*, pp. 426–430, Shanghai, China, December 1996.

[19] 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.

[20] P. Pongchairerks, "Forward VNS, reverse VNS, and multi-VNS algorithms for job-shop scheduling problem," *Modelling and Simulation in Engineering*, vol. 2016, Article ID 5071654, 15 pages, 2016.

[21] M. Sakawa, *Genetic Algorithms and Fuzzy Multiobjective Optimization*, Kluwer Academic Publishers, Boston, MA, USA, 2001.

[22] E. Balas and A. Vazacopoulos, "Guided local search with shifting bottleneck for job shop scheduling," *Management Science*, vol. 44, no. 2, pp. 262–275, 1998.

[23] H. R. Lourenço and M. Zwijnenburg, "Combining the large-step optimization with tabu-search: application to the job-shop scheduling problem," in *Meta-Heuristics: Theory & Applications*, I. H. Osman and J. P. Kelly, Eds., pp. 219–236, Springer, Boston, MA, USA, 1996.

[24] 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, July 2001.

[25] H. R. Lourenço, O. C. Martin, and T. Stützle, "Iterated local search," in *International Series in Operations Research and Management Science*, vol. 57, pp. 321–354, Springer, Boston, MA, USA, 2003.

[26] J. Michallet, C. Prins, L. Amodeo, F. Yalaoui, and G. Vitry, "Multi-start iterated local search for the periodic vehicle routing problem with time windows and time spread constraints on services," *Computers & Operations Research*, vol. 41, pp. 196–207, 2014.

[27] S. Kande, C. Prins, L. Belgacem, and B. Redon, "Multi-start iterated local search for two-echelon distribution network for perishable products," in *Proceedings of the International Conference on Operations Research and Enterprise Systems*, pp. 294–303, Lisbon, Portugal, January 2015.

[28] M. Avci and S. Topaloglu, "A multi-start iterated local search algorithm for the generalized quadratic multiple knapsack problem," *Computers & Operations Research*, vol. 83, pp. 54–65, 2017.

[29] C.-W. Chiou and M.-C. Wu, "A GA-Tabu algorithm for scheduling in-line steppers in low-yield scenarios," *Expert Systems with Applications*, vol. 36, no. 9, pp. 11925–11933, 2009.

[30] T. Davidović, P. Hansen, and N. Mladenović, "Scheduling by VNS: experimental analysis," in *Proceedings of the Yugoslav Symposium on Operations Research*, pp. 319–322, Belgrade, Serbia, October 2001.

[31] M. Marmion, F. Mascia, M. López-Ibáñez, and T. Stützle, "Automatic design of hybrid stochastic local search algorithms," in *Lecture Notes in Computer Science*, vol. 7919, pp. 144–158, Springer, Berlin, Germany, 2013.

[32] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *Proceedings of the IEEE International Conference on Evolutionary Computation*, pp. 69–73, Anchorage, AK, USA, May 1998.

[33] 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.

[34] 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.

[35] 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.

[36] Y. Crama, A. W. J. Kolen, and E. J. Pesch, "Local search in combinatorial optimization," in *Lecture Notes in Computer Science*, vol. 931, 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] 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.

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

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

[43] M. Gen, Y. Tsujimura, and E. Kubota, "Solving job-shop scheduling problems by genetic algorithm," in *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, pp. 1577–1582, San Antonio, TX, USA, October 1994.

[44] Y. Tsujimura, Y. Mafune, and M. Gen, "Introducing co-evolution and sub-evolution processes into genetic algorithm for job-shop scheduling," in *Proceedings of the 26th Annual Conference of the IEEE Industrial Electronics Society*, pp. 2827–2830, Nagoya, Japan, October 2000.

[45] W. Bożejko and M. Makuchowski, "Solving the no-wait job-shop problem by using genetic algorithm with automatic adjustment," *International Journal of Advanced Manufacturing Technology*, vol. 57, no. 5–8, pp. 735–752, 2011.

[46] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[47] 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.

[48] S. Lawrence, *Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*, Carnegie Mellon University, Pittsburgh, PA, USA, 1984.

[49] 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.