2-2013

A Multi-Restart Iterated Local Search Algorithm for the Permutation Flow Shop Problem Minimizing Total Flow Time

Xingye Dong
School of Computer and IT, Beijing Jiatong University, xydong@bjtu.edu.cn

Ping Chen
TEDA College, NanKai University, chenpingteda@nankai.edu.cn

Houkuan Huang
School of Computer and IT, Beijing Jiaotong University, hkhuang@bjtu.edu.cn

Maciek Nowak
Quinlan School of Business, Loyola University Chicago, mnowak4@luc.edu

A Multi-restart Iterated Local Search Algorithm for the Permutation Flow Shop Problem Minimizing Total Flow Time

Xingye Dong[a,c,*], Ping Chen[b], Houkuan Huang[a], Maciek Nowak[c]

[a] *School of Computer and IT, Beijing Jiaotong University, 100044 Beijing, China*
[b] *TEDA College, NanKai University, 300457 Tianjin, China*
[c] *Quinlan School of Business, Loyola University, 60611 Chicago, IL, USA*

Abstract

A variety of metaheuristics have been developed to solve the permutation flow shop problem minimizing total flow time. Iterated local search (ILS) is a simple but powerful metaheuristic used to solve this problem. Fundamentally, ILS is a procedure that needs to be restarted from another solution when it is trapped in a local optimum. A new solution is often generated by only slightly perturbing the best known solution, narrowing the search space and leading to a stagnant state. In this paper, a strategy is proposed to allow the restart solution to be generated from a group of solutions drawn from local optima. This allows an extension of the search space, while maintaining the quality of the restart solution. A multi-restart ILS (MRSILS) is proposed, with the performance evaluated on a set of benchmark instances and compared with six state of the art metaheuristics. The results show that the easily implementable MRSILS is significantly better than five of the other metaheuristics and comparable to or slightly better than the remaining one.

Keywords: Scheduling, Metaheuristic, Iterated local search, Permutation flow shop, Total flow time

*Corresponding author.
Email addresses: xydong@bjtu.edu.cn (Xingye Dong), chenpingteda@nankai.edu.cn (Ping Chen), hkhuang@bjtu.edu.cn (Houkuan Huang), mnowak4@luc.edu (Maciek Nowak)

1. Introduction

Flow shop scheduling is an important and well-known combinatorial optimization problem in operations research, first gaining attention with Johnson's pioneering work [1]. In this problem, n jobs must be processed on m machines in the fixed order $1, \ldots, m$. Though the most commonly studied objective is to minimize the maximum job completion time, or makespan, research has also been devoted to minimizing the total flow time, as it is considered more relevant in today's dynamic production environment [2]. It also tends to stabilize the use of resources and minimize the work-in-process inventory [3]. In this paper, the objective is to minimize the total flow time.

The general flow shop problem is a notoriously hard problem. Among various flow shop problems, the permutation flow shop problem (PFSP), where each machine is required to process the set of all jobs in the same order, has been extensively studied. The PFSP is a simplified version of the flow shop problem; however, it is still a hard problem. Garey et al. [4] prove that the PFSP with makespan criterion is strongly NP-complete with more than two machines, and the PFSP with total flow time criterion is even harder, strongly NP-complete even with two machines.

Many methods have been proposed for total flow time minimization in the literature, including mathematical methods [5, 6, 7, 8, 9, 10] and simple heuristics [11, 12, 13, 14]. However, the former methods are only feasible to solve relatively small-sized problems, and the solution constructed by the latter methods is often poor. A desire to find better solutions more quickly led to the development of many metaheuristics.

Rajendran et al. [15, 16] propose several ant-colony algorithms that lead to better solutions than those found by some constructive heuristics [2]. Though the particle swarm optimization (PSO) algorithm is introduced to optimize continuous functions, Tasgetiren et al. [17] apply this algorithm to the flow shop problem by using the smallest position value rule, and their hybrid algorithm with Variable Neighborhood Search (VNS), PSO_{VNS}, resulted in better solutions than those found by Liu et al. [2], or those using a max-min ant system (M-MMAS) and a renewal ant-colony optimization (PACO) [15] for 57 of 90 of Taillard's benchmark instances [18]. Researchers have developed several genetic algorithms [3, 19, 20], in which different crossover operators and local search procedures are incorporated. Jarboui et al. [21] propose an estimation of distribution algorithm by using VNS to improve performance. Pan et al. [22] develop a differential evolution algorithm hybridized with a referenced local search procedure. Dong et al. show the effectiveness of an iterated local search algorithm [23]. Most

recently, Tasgetiren et al. [24] illustrate an artificial bee colony algorithm and a discrete differential evolution algorithm. In both algorithms, a well designed local search procedure is used. In the local search procedure, the advantages of both an iterated local search algorithm [23] and an iterated greedy algorithm [25] are combined.

Past research shows that ILS is a metaheuristic providing a simple but powerful framework for improving the performance of local search. It has attracted a great deal of attention due to its simplicity, effectiveness and efficiency, and is applied successfully to the travelling salesman problem [26, 27], job shop scheduling [28], and PFSP with makespan criterion [29]. Recently, Ruiz and Stützle propose a very effective iterated greedy algorithm, which is quite similar to the ILS, for the PFSP with makespan criterion [25]. Dong et al. also apply the ILS to the PFSP minimizing total flow time [23], and show that its performance is better than several other metaheuristics, including PACO and M-MMAS by Rajendran and Zeigler [15], ACO2 by Rajendran and Zeigler [16], and PSO_{VNS} by Tasgetiren et al. [17]. Chen et al. also apply a variant of this algorithm to the capacitated vehicle routing problem and find that it performs very effectively [30].

In an ILS algorithm, a restart point is selected when the local search is trapped in a local optimum, and this restart point has a significant effect on the performance of the ILS. Presently, the method to generate the restart point is quite simple: perturbation of the best known solution [23]. This restart method potentially narrows the search space and the search may be lead to a stagnant state. In order to improve this heuristic, it is necessary to develop a method that extends the explorative capability while maintaining the exploitative capability.

In this paper, a scheme is designed in which the restart point is randomly generated from several selective solutions and a new multi-restart ILS (MRSILS) algorithm is proposed. Considerable testing of a variety of alternative techniques developed by the authors revealed that this methodology is the most efficient and effective at solving the PFSP. Comparing MRSILS with other published algorithms confirms this. MRSILS is tested against six recently developed, state of the art metaheuristics here, outperforming five of them and performing equal to or better than the other. In extending the ILS algorithm developed by Dong et al. [23], this paper shows that a simple, yet nontrivial, strategy can provide the best solutions with the least effort. While a great deal of research now focuses on more intricate, sophisticated metaheuristics, this work may encourage others to revisit established methods to determine if adjustments can lead to the improvements found here.

The remainder of this paper is organized as follows. In Section 2, the formulation of the PFSP with total flow time criterion is presented. In Section 3, the proposed MRSILS algorithm is discussed in detail. The computational results are illustrated and analyzed in Section 4, and the paper is concluded in Section 5.

2. Problem formulation

The PFSP minimizing total flow time can be formally defined as follows. A set of jobs $J = \{1, 2, \ldots, n\}$ available at time zero must be processed on m machines, where $n \geq 1$ and $m \geq 1$. Each job has m operations, each of which has an uninterrupted processing time. The processing time of the ith operation of job j is denoted by p_{ij}, where $p_{ij} \geq 0$. The ith operation of a job is processed on the ith machine. An operation of a job is processed only if the previous operation of the job is completed and the requested machine is available. Each machine processes these jobs in the same order and at most one operation of each job can be processed at a time. This problem is usually denoted by $F_m|prmu|\sum C_j$ [31], where C_j denotes the completion time of job j. Let π denote a permutation, which represents a job processing order, on the set J. Let $\pi(k)$, $k = 1, \ldots, n$, denote the kth job in π, then the completion time of job $\pi(k)$ on each machine i can be computed through a set of recursive equations:

$$C_{i,\pi(1)} = \sum_{r=1}^{i} p_{r,\pi(1)} \qquad i = 1, \ldots, m \tag{1}$$

$$C_{1,\pi(k)} = \sum_{r=1}^{k} p_{1,\pi(r)} \qquad k = 1, \ldots, n \tag{2}$$

$$C_{i,\pi(k)} = max\{C_{i-1,\pi(k)}, C_{i,\pi(k-1)}\} + p_{i,\pi(k)} \qquad i = 2, \ldots, m; k = 2, \ldots, n \tag{3}$$

Then $C_{\pi(k)} = C_{m,\pi(k)}$, $k = 1, \ldots, n$. The total flow time is $\sum C_{\pi(k)}$, or the sum of the completion time on machine m for all the jobs. The objective of the PFSP when minimizing total flow time is to minimize $\sum C_{\pi(k)}$.

3. Algorithm description

The framework of ILS is very simple. The pseudo code is presented in Figure 1, where s^* denotes the best solution found in the search history. In the ILS framework, several methodologies must be specified: the method to generate the initial solution in step 1; the local search procedure in step 2; the acceptance criterion in step 3; and, the termination criterion and method to perturb solution s in step 4.

1. Generate an initial solution s; let $s' = s$; let $s^* = s$;
2. Generate $s'' = LocalSearch(s')$; update s^* if better solution is found in the process;
3. Let $s = Accept(s, s'')$ (this may be a simulated-annealing or related function);
4. If the termination criterion is not satisfied, then generate $s' = Perturb(s)$ and go to step 2; otherwise, output s^*.

Figure 1: Pseudo code of an ILS framework

The MRSILS algorithm is presented in Figure 2. Dong et al. [23] use the H(2) method developed by Liu and Reeves [2] to generate an initial solution as it constructs a solution in negligible time and the corresponding ILS performs well. For this reason, the same initial solution method is used in this paper.

Similarly, the local search procedure and acceptance criterion are taken from Dong et al. [23]. The local search procedure corresponds to steps 5 and 6, and it is imbedded in the loop in step 4. The goal of this procedure is to improve the solution by inserting a job into another position in the sequence. The acceptance criterion is that only improved solutions may be accepted. This corresponds to step 8 in Figure 2.

In the literature, the termination criterion can be set as the maximum number of iterations for the local search procedure or the maximum allowable CPU time. In order to make fair comparisons with other metaheuristics, both criteria are used in this paper. This is implemented in step 3.

It is important that the solution s should not be excessively modified during perturbation [23, 24]. In Dong et al. [23], s is set as the best known solution and the perturbation method swaps six random pairs of adjacent jobs. This allows the perturbed solution to still have some characteristics of the best known solution, while potentially moving the algorithm away from a local optimum. However, this method results in a small search space as the restart solution is always relatively close to the best known solution, and so it is difficult to escape from local optima in many cases.

In this work, the search space is extended by generating the restart solution from a set of selected solutions, which are drawn from local optima. If the algorithm finds no improvement by generating the restart point from the best known solution after several iterations, the local optimum is likely difficult to escape from and the search should make a jump larger than that

```
1. Set cnt = 0, flag = false, pool = φ, pool_size, i;
2. Generate an initial permutation π, set π* = π.
3. While(termination criterion is not satisfied)
   {
4.     for(i = 1; i ≤ n; i++)
       {
5.         Find k, which satisfies π(k) = π*(i);
6.         Insert job π(k) into other n − 1 positions in π,
           respectively, and let π' be the best one among
           the n − 1 generated permutations;
7.         if(π' is better than π)
             Set π = π', cnt = 0;
           else
             Set cnt = cnt + 1;
8.         if(π is better than π*)
             Set π* = π, flag = true;
9.         if(cnt == n)
           {
10.          if(flag == true)
               Set pool = φ, flag = false;
11.          if(π is not in pool)
               Add π into pool;
12.          if(|pool| > pool_size)
               Delete the worst schedule from pool;
13.          if(|pool| < pool_size)
               Perturb π* to generate a new π;
             else
               Select one schedule randomly from pool
               and perturb it to generate a new π;
14.          if(π is better than π*)
               Set π* = π;
15.          Set cnt = 0;
           }
       }
   }
   Output π* and stop.
```

Figure 2: Pseudo code of MRSILS

allowed by simple perturbations. This is implemented by generating the restart point from a set of selected solutions.

This set of solutions, denoted by *pool*, is initialized as empty and has a maximum size $pool\_size$. When the search has found no improvements after n iterations, the local optimum is added to *pool*, if it is not already in the set. In order to maintain the best solutions, the worst solution in *pool* is deleted if the size limit $pool\_size$ is exceeded. When *pool* is not full, the restart solution is generated from the best known solution π^*; otherwise, a solution is randomly selected from *pool* and the selected solution is perturbed to generate the restart solution.

Past research shows that the method used for perturbation has a significant impact on performance. These methods include: swapping several pairs of randomly selected adjacent jobs [23]; inserting a job to another position [24]; and, removing several jobs and then inserting them based on a constructive heuristic [24, 25]. Tasgetiren et al. [24] found that the performance is quite good by inserting one or two randomly chosen jobs into randomly selected positions. Testing has shown that the performance of the perturbation method of Tasgetiren et al. [24] is almost the same as that used by Dong et al. [23] when running 1000 iterations, but the former is slightly better for long run times, e.g., 5000 iterations. In MRSILS, solutions are perturbed by inserting one randomly chosen job into another randomly chosen position.

The time complexity of MRSILS is determined by the local search procedure used in the algorithm. The time complexity for computing a solution is $O(mn)$, and moving a job to all other positions will generate $n-1$ solutions, so the time complexity of step 6 is $O(mn^2)$. Step 6 is embedded in the loop in step 4, resulting in a time complexity of $O(mn^3)$. In one iteration, the loop will be computed once, so the time complexity for the entire algorithm is $O(iter \times mn^3)$, where $iter$ is the number of iterations. Given that $iter$ is often a constant, the time complexity for the algorithm is $O(mn^3)$.

4. Computational results

In this section, the procedure for evaluating the parameter $pool\_size$ is described and MRSILS is compared with six state of the art metaheuristics. The benchmark instances used for the analysis are taken from Taillard [18], with the 120 instances evenly distributed among 12 different sizes. The scale of these problems varies from 20 jobs and 5 machines to 500 jobs and 20 machines. In the literature, most metaheuristics are tested on the first 90 benchmark instances [3, 15, 16, 19, 20, 21, 22, 23, 24] as the largest instances

with 200 and 500 jobs are too time consuming. MRSILS is also only tested on the first 90 instances.

MRSILS is implemented in C++, running on a PC with an Intel Core2 Duo processor (2.99 GHz) with 2G main memory. Though the computer has two processors, only one is used in the experiments, as no parallel programming technique has been applied.

4.1. Evaluation of parameter pool_size

MRSILS has only one parameter, *pool_size*, aside from the parameters for the termination criterion which are primarily dictated by the metaheuristics used for comparison. This parameter is tested with values ranging from 1 to 20. For each case, five independent runs with 1000 iterations are performed on each benchmark instance, and the best solution among these five runs is recorded. The relative percentage deviation (RPD) is calculated as:

$$RPD = (F - F_{best})/F_{best} \tag{4}$$

where F is the result found by MRSILS and F_{best} is the best result provided by BEST(LR) [2], M-MMAS and PACO [15] for Taillard's benchmark instances [18]. Here, BEST(LR) denotes the best performing heuristic among several heuristics provided by Liu et al. [2]. Note that a negative RPD value indicates that MRSILS performs better than the best solution found by the other heuristics. The performance for these parameter settings is compared by average RPD (ARPD). The results are shown in Figure 3.

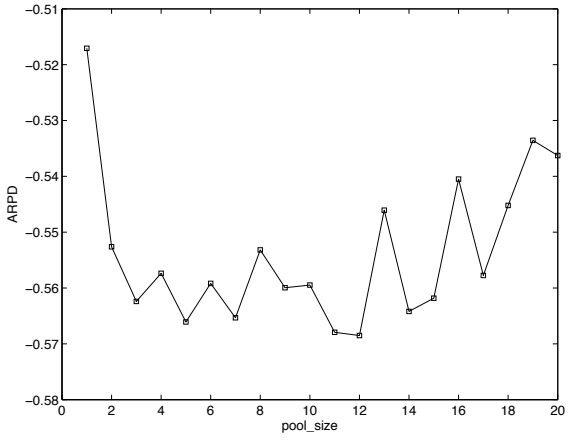

Figure 3: Effects of different *pool_size* (1000 iterations)

Figure 3 shows that *pool_size* is a robust parameter that may take a large range of values, with the best performance between 3 and 12. As the parameter increases above 12, performance becomes somewhat unstable. It should not be surprising that the worst performance occurs with *pool_size* = 1. With that value, there is only one option when selecting a solution to perturb, increasing the likelihood that the algorithm can not escape from the local optimum. However, there are limitations to increasing this parameter. As *pool_size* increases, lower quality solutions may end up in the set *pool* and these may be selected for perturbation. While this may drive the algorithm out of a local optimum, it can also potentially degrade solution quality, resulting in the instability that was evident in the results. In order to allow for a wider range of available solutions, but to limit the potential for instability, *pool_size* is set at 5.

4.2. Performance against previous metaheuristics

In this section, MRSILS is first compared with the ILS algorithm developed by Dong et al. [23], then compared with five other recently published metaheuristics, including the discrete differential evolution algorithm and the iterated greedy algorithm by Pan et al. [22], the hybrid genetic algorithm by Zhang et al. [19], and the discrete artificial bee colony and discrete differential evolution algorithms by Tasgetiren et al. [24]. In the literature, it has been noted that some complex metaheuristics are not easily re-implemented as effectively and efficiently as by the original authors [25], so the results reported here are taken from the original papers. The hardware used for each algorithm is reported as indicated within the respective papers. The hardware used for the MRSILS test runs is very similar in computing power.

In order to make a comparison with the ILS algorithm [23], both the ILS and MRSILS are run for 1000, 3000, and 5000 iterations five independent times, and the best solution among the five runs is recorded. The average RPD for ILS and MRSILS are listed in Table 1.

From Table 1, it can be seen that the average RPD for MRSILS is less than that for ILS, indicating that MRSILS performs better than ILS. Note that for the two smallest data sets, the solutions do not improve on average as the number of iterations increases. With smaller problem instances, the state space is inherently smaller and the benefit of more iterations for exploring this state space is diminished.

To provide more rigorous comparisons with respect to performance, one-sided, paired-samples t-tests are carried out between the ILS and MRSILS with 1000, 3000, and 5000 iterations. Suppose a problem instance is solved

Table 1: Comparison of RPD for ILS and MRSILS

$n\|m$	1000		3000		5000	
	ILS	MRSILS	ILS	MRSILS	ILS	MRSILS
20|5	-0.168	-0.168	-0.175	-0.175	-0.168	-0.168
20|10	-0.035	-0.039	-0.031	-0.039	-0.039	-0.039
20|20	-0.058	-0.068	-0.068	-0.068	-0.068	-0.068
50|5	-0.451	-0.593	-0.592	-0.721	-0.691	-0.775
50|10	-0.955	-1.026	-1.014	-1.169	-1.180	-1.224
50|20	-0.780	-0.863	-0.932	-0.999	-1.024	-1.104
100|5	-0.458	-0.430	-0.549	-0.609	-0.609	-0.707
100|10	-0.814	-0.837	-1.035	-1.141	-1.077	-1.250
100|20	-1.063	-1.072	-1.241	-1.313	-1.353	-1.483
avg.	-0.531	-0.566	-0.626	-0.693	-0.690	-0.757

Table 2: One-sided paired-samples t-tests with respect to the performance between the ILS and MRSILS

iterations	MRSILS > ILS
1000	$p = 0.026$
3000	$p = 0.000$
5000	$p = 0.000$

by two algorithms A and B, and the RPDs are denoted by r_A and r_B, respectively, where the current best known solution and the solutions obtained by A and B are quite good, such that it is assumed that they are close to the unknown global best solution. Then, both r_A and r_B are quite small. If there is no difference between algorithms A and B, then the values of r_A and r_B should be virtually identical. For a group of problem instances, the differences between r_A and r_B may be a result of random error, obeying a normal distribution with mean zero. If the hypothesis of a mean of zero does not hold, then there exists a statistically significant difference between algorithms A and B.

The tests are done for all the instances such that the degrees of freedom value is 89. The results are arranged in Table 2, where the p value is given as the minimum level of significance to accept the hypothesis. The hypothesis is denoted as "MRSILS > ILS", indicating that MRSILS performs better than ILS. From this table, it can be seen that as $\alpha = 0.05 > p$, the hypothesis can be accepted for all the cases. This confirms that MRSILS is significantly better than ILS.

Table 3: Comparison of CPU time (in seconds) for the ILS and MRSILS

$n\|m$	1000		3000		5000	
	ILS	MRSILS	ILS	MRSILS	ILS	MRSILS
20|5	0.13	0.13	0.40	0.40	0.65	0.65
20|10	0.31	0.29	0.92	0.92	1.48	1.46
20|20	0.60	0.58	1.75	1.74	2.96	2.90
50|5	1.79	1.74	5.25	5.21	8.77	8.67
50|10	4.24	4.15	12.65	12.57	21.11	20.80
50|20	8.87	8.73	26.24	26.33	44.25	43.67
100|5	13.01	12.68	38.30	38.18	63.97	63.39
100|10	31.99	31.13	93.16	93.44	156.17	155.79
100|20	69.36	67.15	200.81	201.22	339.31	335.70
avg.	14.48	14.06	42.16	42.22	70.96	70.33

The CPU times for both algorithms are listed in Table 3. The CPU time is almost the same for each pair, with similar increases in time as instance size grows, as the complexity of both algorithms is the same.

Pan et al. [22] propose a discrete differential evolution algorithm (DDE) for the flow shop scheduling problem minimizing total flow time. The DDE with the referenced local search algorithm (DDE_{RLS}) performs quite well on Taillard's benchmark instances. The authors also design an iterated greedy algorithm with referenced local search (IG_{RLS}) and it performs well on the same benchmark instances. Both algorithms are implemented in Visual C++ and run on an Intel Pentium IV 3.0GHz PC with 512MB memory. These two algorithms are run for $n \times m/2 \times 90$ milliseconds CPU time for 5 replications and the best results are reported. In order to make a fair comparison, MRSILS is run with identical settings, including the same CPU stopping time criterion. Note that the speed of the computer used by Pan et al. is comparable to that of the computer used for MRSILS testing. Table 4 presents a comparison of the RPD for all three algorithms. MRSILS performs similarly to or better than IG_{RIS} and DDE_{RLS} on all nine group instances. Two groups of paired-samples t-test are also carried out and the results show that MRSILS is significantly better than IG_{RIS} and DDE_{RLS}, with $p = 0.000$ for all cases.

Zhang et al. [19] propose a hybrid genetic algorithm (denoted by HGA-Z here), which is implemented in Java and tested on a 2.93GHz P4 PC with 512M RAM. The best solution in five runs and the average CPU time per run for each instance are also reported. The RPD for HGA-Z is computed

Table 4: Comparison in RPD between MRSILS, IG_{RIS}, and DDE_{RLS}

$n\|m$	IG_{RIS}	DDE_{RLS}	MRSILS
20|5	-0.168	-0.168	-0.168
20|10	-0.039	-0.039	-0.039
20|20	-0.068	-0.068	-0.068
50|5	-0.708	-0.683	-0.747
50|10	-1.089	-1.118	-1.212
50|20	-0.833	-0.876	-0.978
100|5	-0.422	-0.438	-0.512
100|10	-0.820	-0.879	-1.029
100|20	-1.011	-0.959	-1.147
avg.	-0.573	-0.581	-0.656

using the best solutions reported by Zhang et al. and compared to MRSILS results with 5000 iterations. The results are shown in Table 5, where t denotes the average CPU time. The overall performance of MRSILS is much better than HGA-Z, with a considerably lower average RPD for MRSILS. A paired-samples t-test is also carried out, and the result shows that the p value is 0.000, which indicates that MRSILS is significantly better than HGA-Z. As computer speed and programming languages differ, it is difficult to compare the CPU times directly. However, these results show that the improved performance of MRSILS is not at a cost of increased CPU time.

Table 5: Comparison in RPD and computational time (in seconds) between MRSILS and HGA-Z

$n\|m$	HGA-Z		MRSILS	
	RPD	t	RPD	t
20|5	-0.175	2.18	-0.168	0.65
20|10	-0.039	4.14	-0.039	1.46
20|20	-0.068	7.57	-0.068	2.90
50|5	-0.707	40.85	-0.775	8.67
50|10	-1.041	111.74	-1.224	20.80
50|20	-0.941	189.98	-1.104	43.67
100|5	-0.739	461.33	-0.707	63.39
100|10	-1.045	826.76	-1.250	155.79
100|20	-1.295	2014.96	-1.483	335.70
avg.	-0.672	406.61	-0.757	70.33

Tasgetiren et al. [24] propose a discrete artificial bee colony algorithm

(DABC) and apply the artificial bee strategy to the discrete differential evolution algorithm developed by Pan et al. [22], forming a new algorithm, hDDE. Both algorithms perform better than the estimation of distribution algorithm by Jarboui et al. [21], the genetic local search algorithms by Tseng and Lin [3, 20], and the traditional iterated greedy algorithm [25]. Both the DABC and the hDDE are implemented in Visual C++ and run independently 10 times on an Intel Pentium IV 3.0 GHz PC with 512MB memory. The best results are reported for both algorithms with a short-term search, for which the CPU time is limited to $0.4 \times n \times m$ seconds for each instance, and a long-term search, for which the CPU time is limited to $3 \times n \times m$ seconds for each instance. The long-term search is too time consuming, requiring more than 20 days for one algorithm and these results are not presented here.

Table 6 presents a comparison of RPD for MRSILS, DABC, and hDDE, showing that all three algorithms perform well on those instances with 20 jobs. For the 50 and 100 job instances, MRSILS generally performs better on larger instances and slightly worse on smaller instances. Paired-samples t-tests show that MRSILS is significantly better than DABC with $p = 0.001$, but this is not the case for hDDE with $p = 0.162$. However, MRSILS is comparable to or even better than hDDE when average RPD is considered. Additionally, hDDE is quite complex, while MRSILS is rather simple. The best solutions obtained in this experiment are shown in Table 7, where 27 improved solutions are listed in boldface. The results on the instances with 20 jobs are identical for the three algorithms, so they are not presented here. Compared with DABC, MRSILS finds a better solution for 37 instances and is equivalent for one instance. MRSILS finds a better solution than hDDE for 32 instances and is equivalent for 4 instances.

5. Conclusions

This paper presents a strategy to improve the performance of the ILS algorithm on the permutation flow shop problem minimizing total flow time. The strategy extends the explorative capability of the local search on which ILS is based, while maintaining a strong exploitative capability. Experimental results on benchmark instances show that MRSILS performs better than five state of the art metaheuristics and comparable to or slightly better than another state of the art metaheuristic.

In addition providing good solutions, MRSILS is easily implementable. Compared with other quite complex metaheuristics, this work shows that the ILS algorithm is worthy of additional research. For example, the generation

Table 6: Comparisons results for the MRSILS with the DABC and the hDDE

$n\|m$	DABC	hDDE	MRSILS
20|5	-0.175	-0.175	-0.175
20|10	-0.039	-0.039	-0.039
20|20	-0.068	-0.068	-0.068
50|5	-0.961	-0.984	-0.956
50|10	-1.426	-1.433	-1.436
50|20	-1.252	-1.191	-1.194
100|5	-0.704	-0.918	-0.874
100|10	-1.132	-1.313	-1.367
100|20	-1.508	-1.477	-1.616
avg.	-0.807	-0.844	-0.858

of the restart solution in MRSILS still requires tuning, as the best known solution is not improved after many iterations in some instances. Finding a heuristic or designing a metaheuristic that can help generate more meaningful restart solutions deserves attention in future research.

Acknowledgements

The authors would like to give thanks to the two anonymous reviewers for their valuable suggestions and comments. This work is supported by The Fundamental Research Funds for the Central Universities of China (Project Ref. 2009JBM018, Beijing Jiaotong University).

References

[1] S. Johnson, Optimal two and three-stage production schedule with setup times included, Naval Research Logistics Quarterly 1 (1) (1954) 61–68.

[2] J. Liu, C. Reeves, Constructive and composite heuristic solutions to the $p//\sum c_i$ scheduling problem, European Journal of Operational Research 132 (2001) 439–452.

[3] L.-Y. Tseng, Y.-T. Lin, A hybrid genetic local search algorithm for the permutation flowshop scheduling problem, European Journal of Operational Research 198 (2009) 84–92.

[4] M. Garey, D. Johnson, R. Sethi, The complexity of flowshop and jobshop scheduling, Mathematics of Operations Research 1 (1976) 117–129.

Table 7: Best solutions obtained by the hDDE, DABC and MRSILS on Taillard's benchmarks

$n\|m$	hDDE	DABC	MRSILS	$n\|m$	hDDE	DABC	MRSILS
50|5	64803	64803	64838	100|5	254319	254738	**253973**
	68051	68086	68077		243410	243834	243963
	63226	63162	63241		238772	239242	**238654**
	68345	68242	68298		228518	228925	**228412**
	69360	69448	69449		241243	241959	241649
	66841	66878	66841		233696	234017	**233651**
	66271	66271	**66253**		241013	241727	241357
	64365	64381	64578		231716	232238	232167
	63015	63081	**62981**		249180	249884	**248999**
	68906	68989	**68811**		243838	244335	243927
50|10	87143	87340	87541	100|10	300201	301204	**299957**
	82949	83068	83080		275920	276470	**275868**
	80105	80139	**80094**		289366	289400	289545
	86547	86525	**86469**		303403	303062	**302322**
	86511	86453	**86377**		285950	286742	**285466**
	86730	86687	**86645**		271601	272282	272082
	89024	88996	**88778**		280921	281716	281221
	86886	86883	86907		292664	293071	**292255**
	85646	85637	85646		303742	304457	304026
	88139	87998	88108		293138	293775	**292505**
50|20	125877	125842	125955	100|20	368702	369297	**367580**
	119270	119270	119270		374894	374321	**374156**
	116628	116712	116779		372057	373210	**372013**
	120983	120897	**120839**		375540	374205	374521
	118767	118457	**118366**		370646	371334	370699
	120703	120850	121083		373826	373689	374343
	123084	123043	123084		376807	375118	375813
	122672	122529	122791		386803	387582	**386173**
	122018	121872	122111		377730	377113	377186
	124327	124097	**124005**		380773	380725	**379983**

[5] E. Ignall, L. Schrage, Application of the branch and bound technique to some flow-shop scheduling problems, Operations Research 13 (1965) 400–412.

[6] E. Stafford, On the development of a mixed integer linear programming model for the flowshop sequencing problem, Journal of the Operational Research Society 39 (1988) 1163–1174.

[7] S. L. van de Velde, Minimizing the sum of the job completion times in the two-machine flow shop by lagrangian relaxation, Annals of Operations Research 26 (1990) 257–268.

[8] F. Croce, V. Narayan, R. Tadei, The two-machine total completion time flow shop problem, European Journal of Operational Research 90 (1996) 227–237.

[9] F. Croce, M. Ghirardi, R. Tadei, An improved branch-and-bound algorithm for the two machine total completion time flow shop problem, European Journal of Operational Research 139 (2002) 293–301.

[10] C. Chung, J. Flynn, O. Kirca, A branch and bound algorithm to minimize the total flow time for m-machine permutation flowshop problems, International Journal of Production Economics 79 (2002) 185–196.

[11] J. Framinan, R. Leisten, R. Ruiz-Usano, Comparison of heuristic for flowtime minimisation in permutation flowshops, Computers and Operations Research 32 (2005) 1237–1254.

[12] X. Li, C. Wu, An efficient constructive heuristic for permutation flow shops to minimize total flowtime, Chinese Journal of Electronics 14 (2) (2005) 203–208.

[13] X. Li, Q. Wang, C. Wu, Efficient composite heuristics for total flowtime minimization in permutation flowshops, OMEGA 37 (2009) 155–164.

[14] D. Laha, S. C. Sarin, A heuristic to minimize total flowtime in permutation flowshop, Omega 37 (2009) 734–739.

[15] H. Z. C. Rajendran, Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs, European Journal of Operational Research 155 (2004) 426–438.

[16] C. Rajendran, H. Ziegler, Two ant-colony algorithms for minimizing total flowtime in permutation flowshops, Computers and Industrial Engineering 48 (2005) 789–797.

[17] M. Tasgetiren, Y.-C. Liang, M. Sevkli, G. Gencyilmaz, A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem, European Journal of Operational Research 177 (2007) 1930–1947.

[18] E. Taillard, Benchmarks for basic scheduling problems, European Journal of Operational Research 64 (1993) 278–285.

[19] Y. Zhang, X. Li, Q. Wang, Hybrid genetic algorithm for permutation flowshop scheduling problems with total flowtime minimization, European Journal of Operational Research 196 (3) (2009) 869–876.

[20] L.-Y. Tseng, Y.-T. Lin, A genetic local search algorithm for minimizing total flowtime in the permutation flowshop scheduling problem, International Journal of Production Economics 127 (2010) 121–128.

[21] B. Jarboui, M. Eddaly, P. Siarry, An estimation of distributation algorithm for minimizing the total flowtime in permutation flowshop scheduling problems, Computers and Operations Research 36 (2009) 2638–2646.

[22] Q.-K. Pan, M. Tasgetiren, Y.-C. Liang, A discrete differential evolution algorithm for the permutation flowshop scheduling problem, Computers and Industrial Engineering 55 (2008) 795–816.

[23] X. Dong, H. Huang, P. Chen, An iterated local search algorithm for the permutation flowshop problem with total flowtime criterion, Computers and Operations Research 36 (2009) 1664–1669.

[24] M. Tasgetiren, Q.-K. Pan, P. Suganthan, A. H.-L. Chen, A discrete artificial bee colony algorithm for the total flowtime minimization in permutation flow shops, Information Sciences 181 (2011) 3459–3475.

[25] R. Ruiz, T. Stützle, A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem, European Journal of Operational Research 177 (2007) 2033–2049.

[26] O. Martin, S. Otto, E. Felten, Large-step markov chains for the traveling salesman problem, Complex Systems 5 (3) (1991) 299–326.

[27] O. Martin, S. Otto, Combining simulated annealing with local search heuristics, Annals of Operations Research 63 (1996) 57–75.

[28] H. Lourenco, Job-shop scheduling: computational study of local search and large-step optimization methods, European Journal of Operational Research 83 (1995) 347–364.

[29] T. Stützle, Applying iterated local search to the permutation flow shop problem, Technical Report, AIDA-98-04, FG Intellektik, TU Darmstadt.

[30] P. Chen, H. Huang, X. Dong, Iterated variable neighborhood descent algorithm for the capacitated vehicle routing problem, Expert Systems with Applications 37 (2010) 1620–1627.

[31] M. Pinedo, Scheduling: theory, algorithms, and systems, 2nd Edition, Prentice Hall, 2001.