J Sched (2006) 9: 515–543
DOI 10.1007/s10951-006-8787-z

Two-machine flow shop problems with a single server

Andrew Lim · Brian Rodrigues · Caixia Wang

Abstract Scheduling models that allow the handling of pre-operational setup have been a source of major interests because of their practical relevance and theoretical impacts. Two-stage flow-lines have drawn much attention to researchers as they are simple, yet practical and can be easily extended to represent more complex situations. In this paper, two-machine flow-shop problems with a single setup server are surveyed. These problems have been shown to be NP-complete with special cases that are polynomial-time solvable. Several heuristics are proposed to solve the problems in general case, including simulated annealing, Tabu search, genetic algorithms, GRASP, and other hybrids. The results on small inputs are compared with the optimal solutions and results on large inputs are compared to a lower bound. Experiments show that the heuristics developed, obtain nearly optimal solutions.

Keywords Machine scheduling · Flow shop · Heuristics

1. Introduction

Scheduling models that allow the handling of pre-operational work, i.e., setup, have been a source of major interests for some decades because of their practical relevance and theoretical impacts. Modern computing and manufacturing processes provide a continuous supply of new models with various types of setups. (Glass et al., 2000) For example, in just in time (JIT) manufacturing systems, the one-worker-multiple-machine (OWMM) concept is widely applied. This concept basically means that one worker tends several machines simultaneously, where all machines perform different operations and thus constituting a flow-line. The machine layout is organized

A. Lim (✉)
Department of IEEM, Hong Kong University of Science and Technology, Clearwater Bay, Hong Kong
e-mail: jealim@ust.hk

B. Rodrigues
School of Business, Singapore Management University, 50 Stamford Road, Singapore 178899

C. Wang
School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543

in a logical sequence, according to the order of the operations to be performed on the jobs, and the worker setups the corresponding machines before the starting of each operation. Krajeswski and Ritzman (1987) two-stage flow-lines usually draw more attention to researchers, as they are simple, yet practical and can be easily extended to more complex situations.

Take a CD manufactory as an example. The process of producing CDs includes two steps, namely, replication and printing. Before the replication, the stamper is installed into the injection-molding machine. Then the CD can start performing injection, metallization, and inspection. After finishing replication, the CD is transferred to the printing section. A setup is required on the printer before starting printing. In the simplest case, there is only one machine for each of the two steps and one worker is assigned in the flow-line to do the setup works, thus forming the model of a two-machine flow-shop problem with a single server.

In this paper, we study the two-machine flow-shop problems with a single server. Firstly, we assume that all processing times are constant. We call this kind of problem—TMFSP problem, which has been shown to be NP-hard (Brucker et al., 2001). AI approaches including metaheuristics are developed. A lower bound is proposed to compare the heuristics and some special cases are proven to be polynomial time solvable. Next, this problem is generalized by removing the constant processing times constraint, which changes to TMFS problem.

2. Problem definition

The two-machine flow-shop problem with a single server and constant processing time (TMFSP) is formally designated as $F2, S1\,|P_{ij} = p|C_{\max}$.

Given two machines M_1, M_2, and N jobs $i = 1, 2, \ldots, N$, each job i consists of two operations O_{ij} $(j = 1, 2)$ which have to be processed in the order, first O_{i1} then $O_{i2}(O_{i1} \to O_{i2})$. Operation O_{ij} has to be processed on machine M_i without preemption for $P_{ij} > 0$ time units. In TMFSP problems, all P_{ij}'s are constant and equal to p. Each machine can only process one operation at a time. Immediately before processing an operation, O_{ij} has to be prepared for processing O_{ij} the corresponding machine, which takes a setup time of S_{ij} time units. During such setup, the machine is also occupied for S_{ij} time units, i.e., no other job can be processed on it. The setup times are assumed to be separable from processing times, i.e., a setup on a subsequent machine may be performed while the job is still in process on the preceding machine. All setups have to be done by a single server that can perform at most one setup at a time. The goal is to determine a feasible schedule that minimizes the objective function, makespan.

For all $i = 1, 2, \ldots, N$ and $j = 1, 2$, let $T(S_{ij})$ and $T(P_{ij})$ be the starting times of setup and processing operation O_{ij} separately and let

$$y_{ii'j} = \begin{cases} 1, & \text{if } O_{ij} \textit{ precedes } O_{i'j} \\ 0, & \text{if } O_{i'j} \textit{ precedes } O_{ij} \end{cases},$$

$$z_{iji'j} = \begin{cases} 1, & \text{if } O_{ij} \textit{ precedes } O_{i'j} \\ 0, & \text{if } O_{i'j'} \textit{ precedes } O_{ij} \end{cases},$$

and

M be a large integer.

Then, a Integer Programming model for the TMFSP problem can be given as follows: (note $P_{ij} = p$ for $\forall i = 1, 2, \ldots, N$ and $j = 1, 2$)

$$\begin{aligned}
&\text{Minimize } C\\
&\text{s.t.} \quad \forall i, i' = 1, 2, \ldots, N \text{ and } \forall j, j' = 1, 2 :\\
&T(S_{i1}) \geq 0\\
&T(S_{i1}) < T(S_{i2})\\
&T(P_{ij}) = T(S_{ij}) + S_{ij}\\
&T(P_{i1}) + P_{i1} \leq T(P_{i2})\\
&My_{ii'j} + (T(S_{ij}) - T(S_{i'j})) \geq S_{i'j} + P_{i'j}(i \neq i')\\
&M(1 - y_{ii'j}) + (T(S_{i'j}) - T(S_{ij})) \geq S_{ij} + P_{ij}(i \neq i')\\
&Mz_{iji'j'} + (T(S_{ij}) - T(S_{i'j'})) \geq S_{i'j'}(i \neq i' \; or \; j \neq j')\\
&M(1 - z_{iji'j'}) + (T(S_{i'j'}) - T(S_{ij})) \geq S_{ij}(i \neq i' \text{ or } j \neq j'\\
&T(P_{i2}) + P_{i2} \leq C
\end{aligned}$$

3. Related work

In 1954, Johnson proved that the two-stage flow-shop problem with makespan as the criterion can be solved in polynomial time (Johnson, 1954). He showed that the optimal solution of the flow-shop problem is not better than that of the corresponding permutation flow-shop, where solutions are restricted to the same job sequences on all machines. Following Johnson's work, many researchers have focused on solving m-machine ($m > 2$) flow-shop problems with the same criterion. As these are in the class of NP-hard problems (Garey et al., 1974), focus has been kept on developing heuristics, such as simulated annealing (Osman and Potts, 1989; Ogbu and Smith, 1990, 1990b); Tabu search (Widmer and Hertz, 1989; Taillard, 1990); genetic algorithms (Chen et al., 1995; Murata et al., 1996; Yamada and Reeves, 1998b). In these models, the setup times for the operations are sequence-independent and included the in processing times.

Another set of related models are sequence-dependent setup time flow-shop problems (SDST flow-shop) where setup times are strongly dependent on the job order. The two-stage case is proven to be NP-complete in the strong sense (Gupta and Darrow, 1986). Flow shop with multiple processors (FSMP) has been extensively studied in hybrid flow-shop literature. Narasimhan and Panwalkar (1984) considered a real-life FSMP with one machine at stage 1 and two machines at stage 2. A cumulative minimum deviation rule was suggested for reducing the sum of machine idle time and in-process job waiting time. Gupta (1988) showed that the two-stage FSMP problem is NP-hard, and developed a heuristic in finding a minimum makespan schedule of a special case when there is only one machine at stage 2.

However, if the setup is sequence-independent and separated from the processing, two-stage flow-shop problems can be solved in polynomial time when infinite servers are available (Yoshida and Hitomi, 1979). In 2001, Brucker proved that if there is only one server available, the two-stage flow-shop problem is NP-complete in the strong sense (Brucker et al., 2001). He also proved that the special case where both setup times and processing times are constant is polynomial-time solvable. Cyclic movement of server was assumed by Cheng, which requires the setup server to move between the two machines, according to some cyclic pattern. This new model was proved to be NP-complete in the strong sense, and their worst-case error bounds were analyzed (Cheng et al., 1999). With the cyclic constraint, the size of solution space is reduced to N!. However, in Section 5, we show that such constraint will exclude optimal solutions in the general case. Glass added a no-wait constraint into the two-stage flow-shop problems with a setup server, which is reduced to Gilmore-Gomory traveling salesman problem, and solved it in polynomial time. In Glass's model, each job, on the completion of processing on the first machine must be transferred immediately (no wait) to the second machine (Glass et al. 2000).

The problem of scheduling two dedicated parallel machines with a single server was studied by Abdekhodaee and Wirth (2002). Here the jobs are treated as basic elements instead of operations, and can be carried out in parallel on the two machines. A single server is required to carry out job setups. Specifically, they assume alternate allocation to the two machines. This model is shown to be NP-complete and a special case is solved.

In Table 1, we summarize the known complexity results of related scheduling problems. These are in $\alpha|\beta|\gamma$ form, where we use "setup" in β field when the setup is separated from the processing. If there is only one server (S1), this is entered in the α field.

4. Preliminaries

We provide some preliminaries in this section, which include the design of experiments, lower bounds, representation, and validation of solutions.

4.1. Experiment design

All experiments in this work are carried out with test data with following properties:

- The constant processing time is fixed at 1000 units;
- For specific number of jobs N in the inputs, nine different ratios R of average setup time to processing time are used, which are 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, and 100 respectively;
- For each N and R, 10 test cases are randomly generated.

For example, if we choose $N = 10$ and $R = 0.1$, then each of the 10 test cases has constant processing time $p = 1000$ and setup times an uniformly distributed from 0 to 200 (average $= 100 = p \times R$). The purpose of generating test cases, in this way is to ensure diversity in

Table 1 Related models

Problem	Complexity	Reference
F2$\|\|C_{\max}$	O(nlogn)	Johnson, 1954
Fm$\|\|C_{\max}$ $(m > 2)$	NP-hard	Garey et al., 1976
F2$\|$setup$\|C_{\max}$	O(nlogn)	Yoshida et al., 1979
F2$\|$SDST$\|C_{\max}$	NP-hard	Hupta, 1986
F2$\|$FSMP$\|C_{\max}$	NP-hard	Gupta, 1988
F2,S1$\|$cyclic$\|C_{\max}$	NP-hard	Cheng et al., 1999
F2,S1$\|$no − wait$\|C_{\max}$	O(nlogn)	Glass et al., 2000
F2,S1$\|$P$_{ij} = p$, S$_{ij} = s\|C_{\max}$	O(n)	Brucker et al., 2001
P2,S1$\|$P$_{ij} = p\|C_{\max}$	NP-hard	Abdekhodaee, 2002
P2,S1$\|\|C_{\max}$	NP-hard	Abdekhodaee, 2002

testing by varying the setup times and processing times among jobs as much as possible. Since all test inputs are of these properties, in the later experiments, only N is mentioned for the inputs.

For each heuristic, an extensive experiment is carried out. The aim is to find out an approximate best combination of alternative components and parameters. The test data used for these experiments included those with $N = 5, 10, 15, 20, 50, 80$, and 100. (All experiments are done on Pentium III 1.4 GHz machines).

4.2. Lower bounds

A lower bound of $C_{\max}$ for the solutions of the TMFSP problem is the maximum of C_1, C_2 and C_3, where C_1, C_2 and C_3 are derived from the following propositions.

Proposition 1. $C_{\max} \geq C_1$, *where* $C_1 := \sum_{i=1}^{N} \sum_{j=1}^{2} S_{ij} + p$

Proof: No two setups can be carried out simultaneously on the single server. The best case is that there is no idle time on it. So $C_{\max} \geq$ sum of all setup time. In addition, after the last setup, there follows a processing, which will delay the finishing time by p (Fig. 1). □

Proposition 2. $C_{\max} \geq C_2$, *where* $C_2 := \sum_{i=1}^{N}(S_{i1} + P_{i1}) + \max\{\textit{minimum of } S_{i2}, p\}$

Proof: The best case on machine 1 is when is no idle time, so $C_{\max} \geq$ sum of setup and processing times of all stage-1 operations. For the last job on machine 1, its second operation can only start after its first is completed, and thus delays the finishing time by at least p (Fig. 2).

However, the setup of the last operation on machine 2 can only be carried out after completing the setups of all operations on machine 1. If the minimum setup time on machine 2 is larger than the constant processing time, then a delayed finishing time by the last setup on machine 2, in the best case, is the minimum setup time (Fig. 3). □

Proposition 3. $C_{\max} \geq C_3$, *where* $C_3 := \sum_{i=1}^{N}(S_{i2} + P_{i2}) + \textit{minimum } S_{i1}$

Proof: We have an optimal when there is no idle time on machine 2, so $C_{\max} \geq$ sum of setup and processing times of all stage-2 operations. However, by the flow-shop precedence rule, there

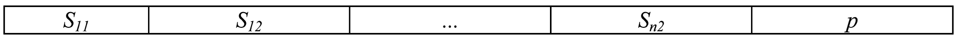


Fig. 1 Lower bound on server

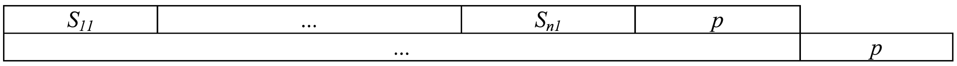


Fig. 2 Lower bound on machine 1

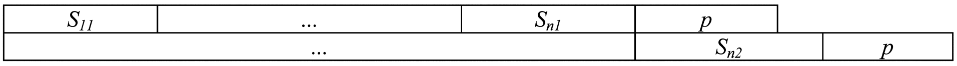
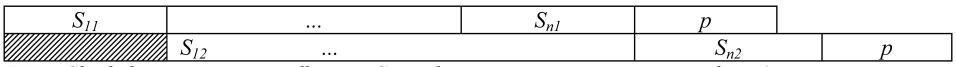


Fig. 3 Lower bound with large setups on machine 2



Note: Shaded area represents idle time. S_{11} is the minimum setup time on machine 1.

Fig. 4 Lower bound on machine 2

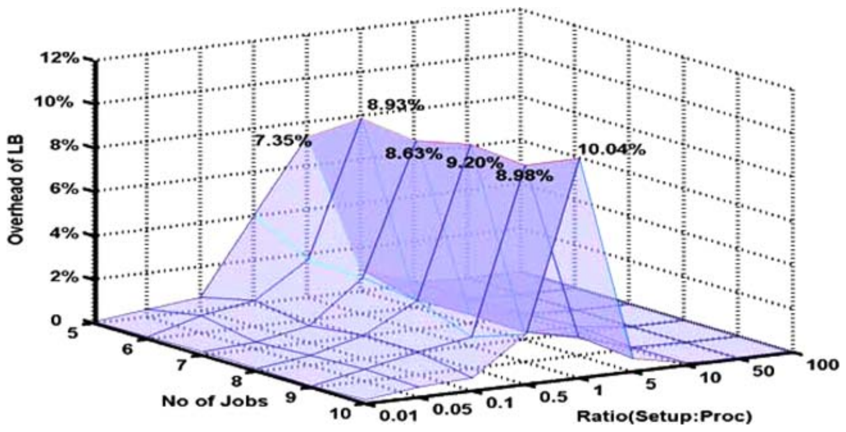


Fig. 5 Excess: LB to optimal solution

must be a setup on machine 1 before its first setup. The best case is when the first setup on machine 1 requires minimum time (Fig. 4). □

4.2.1. An experiment is carried out to check this lower bound (*LB*). The inputs include test cases of six different job sizes, $N = 5, 6, 7, 8, 9,$ and 10. The excess is calculated by $(LB - OPT)/OPT \times 100\%$, where *OPT* is the $C_{\max}$ of optimal solution found by a branch and bound method. The *LB*'s are tight when R is small or large, but loose when R is in the range (0.5,5) (Fig. 5). The reason for this is that when setup times and processing times are near each other, there is more avoidable idle time on both machines and the setup server.

In this work, as it is prohibitive to find optimal solutions for large *N*, when evaluating the performance of heuristics, all makespans are compared to our lower bounds (LB) and excess values are calculated as $excess := (C_{\max} - LB)/LB \times 100\%$.

4.3. Representation and Validation of Solutions

In our solution representation, the first operation of job i, O_{i1}, is represented as i while the second operation O_{i2} is represented as $i + N$, where N is the number of jobs. Each solution is then represented as an integer array with length $2 \times N$ and each entry represents an operation. The order of operations in the array is the order of operations being setup on the server. The starting time of an operation is then the earliest available time of the server and corresponding machine without violating the precedence rule of flow-shop. For example, given an input with $N = 3$, sequence $\{1, 2, 4, 5, 3, 6\}$ represents a solution in Fig. 6.

Different from classical flow-shop problems, optimal solutions may not have same sequences on the two machines, which means the size of solution space is $(2N)!$. We give the following example:

Table 2 Setup times

Job	Operation 1	Operation 2
1	2	24
2	9	18
3	3	1
4	19	1
5	2	4

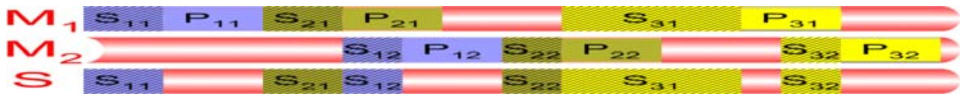


Fig. 6 Solution representation for TMFSP problems

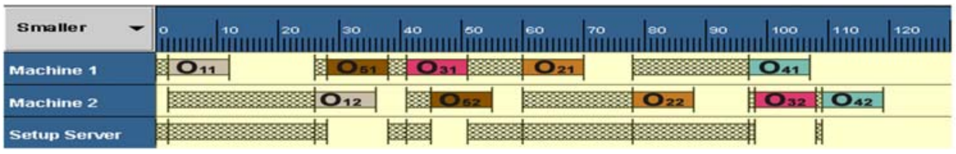


Fig. 7 Unique optimal solution

Input: $N = 5$, constant processing time $p = 10$ and setup times are listed in Table 2.
Output: The only optimal solution is as in Fig. 7.

However, some solutions may not be valid. A solution is *valid* only if all jobs obey the flow-shop precedence rule. In our solution representation, this means that operation $j (j > N)$ must come after $j - N$. In order to get feasible solutions, a *validation step* is needed as: all the operations in the solution are checked, if there is some job i with O_{i2} starting before finishing O_{i1}, then O_{i1} is moved to a random position in front of O_{i2}.

5. Some polynomial-time solvable cases

As TMFSP problem is NP-hard (Brucker et al., 2001), we do not expect to find polynomial time algorithms for it. However, special cases may be amenable to fast and exact solution.

Denote by $W_i (i = 1, 2, \ldots 2 \times N - 1)$ the server waiting time between the end of setup the ith and the start of the $(i + 1)$th scheduled operations. (Note: All scheduled order in this section refers to the order of scheduled operations on the setup server). Denote by $T_i (i = 1, 2, \ldots 2 \times N)$ the starting time of processing the ith scheduled operation. Then, $\forall i$, $i = 1, 2, \ldots,$ 2N-1, $W_i = T_{i+1} - T_i - S_i$, $C_{\max} = \sum_{i=1}^{N} \sum_{j=1}^{2} S_{ij} + \sum_{i=1}^{2N-1} W_i + p$. Hence, to minimize $C_{\max}$, we need only minimize $\sum_{i=1}^{2N-1} W_i$.

Proposition 4 (Short processing times). *If $P_{ij} \le S_{ij}$ for all $1 \le i \le N$, $1 \le j \le 2$, then an optimal permutation schedule (i.e., each machine process the jobs in the same order) exists in which the server serves the jobs one after the other and performs both setups for a job without interruption, i.e., any representation sequence $\{O_{L(1)1}, O_{L(1),2}, O_{L(2),1}, O_{L(2),2}, O_{L(3),1}, O_{L(3),2}, \ldots,$ where L is an arbitrary permutation of $\{1, 2, \ldots N\}$, is an optimal solution.*

Proof: We declare that in any such sequence s, $T_{i+1} = T_i + S_i (i = 1, 2, \ldots, 2N - 1)$. A proof by mathematical induction is given in Appendix A. Then, $W_i = T_{i+1} - T_i - S_i = 0$ for all i. As we have declared, minimizing $C_{\max}$ is to minimize $\sum_{i=1}^{2N-1} W_i$, which is equal to 0. Thus, s is an optimal solution. Note: without the condition that all processing times are constant (TMFS problems), the proposition is true if the job with minimum processing time of its second operation is scheduled at the last position.

In Brucker's paper (Brucker et al., 2001), $F2, S1|S_{ij} = s, P_{ij} = p|C_{\max}$ was proven to be polynomial time solvable. In Proposition 5, we generalize this. $\square$

Proposition 5. *If $S_{i1} = s1$ and $S_{i2} = s2$ for all $1 \le i \le N$, then an optimal permutation schedule exists in which the server serves the jobs one after the other and performs both setups for a job without interruption.*

Proof: Since now the length of each operation for all jobs becomes the same, it is sufficient to prove that any sequence with alternative setup on the two machines is optimal. We consider the following three cases.

Case 1. $s_1 \ge p$ and $s_2 \ge p$

This is a special case of short processing times, which has been proven in Proposition 4.

Case 2. $s_1 \ge s_2$ and $s_2 \le p$

We assume sequence s is an optimal solution and the setup of s is alternatively done on the two machines until the $(k - 1)$th scheduled job. It means that after the first operation of the kth scheduled job is setup, the next setup is still on machine 1, that is the $(k + 1)$th schedule job. Without loss of generality, we take the number of such continuous operations to be two. Denoting the first such operations on machine 1 as O_i and O_{i+1} respectively and after O_{i+1} the next operation to be setup on machine 2, to be O_h, say. Then, the end time of the operation before O_i, $E(O_{i-1}) \le T(O_i) + p$ as its setup must be completed before $T(O_i)$ and the idle time on the setup server between O_i and O_{i+1}, $W_i = T(O_{i+1}) - T(O_i) - s1 = p$, and $T(O_h) \ge T(O_{i+1}) + s1$ (Fig. 8).

Now we shift O_h to the left with the new starting time $T'(O_h) = T(O_{i+1}) - s2 = T(O_i) + s1 + p - s2$. Since $s2 \le p$, so $T'(O_h) \ge T(O_i) + s1$ and since $s_2 \le s_1$, so $T'(O_h) \ge T(O_i) + p \ge E(O_{i-1})$. Also, $T'(O_h) + s2 = T(O_{i+1}) - s2 + s2 = T(O_{i+1})$. Thus the shift is valid (Fig. 9).

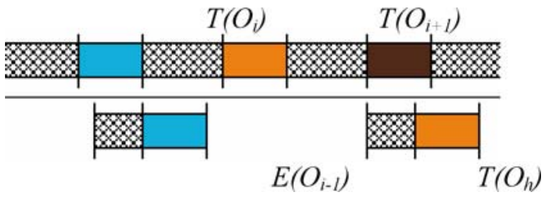


Fig. 8 Machine status before shift

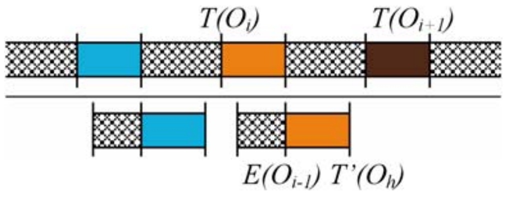


Fig. 9 Machine status after shift

If the second operation of the kth scheduled job is not O_h, then it must be somewhere after O_h and is easily swapped with O_h. Continuing shifting and swapping until all setups are done alternatively on the two machines. As each movement does not shift the last scheduled operation to the right, $C_{\max}$ of the new sequence is no larger than $C_{\max}$ of s. Since s is optimal, the new sequence is also optimal.

Case 3. $s2 \geq s_1$ and $s_1 \leq p$

This case is symmetric to Case 2. Please refer to Appendix B for the proof.

This completes the proof.

□

6. Heuristics and experiments

First, we develop two greedy initial solutions for the problem. We then use simulated annealing, tabu search, genetic algorithm and GRASP meta-heuristic approaches to develop solutions. We also propose hybrid heuristics.

6.1. Two greedy initial solutions

Two greedy methods are proposed to find a solution for the TMFSP problems.

Modified Johnson's algorithm (MJA)

Johnson's Algorithm (Johnson, 1954) was used to solve two-machine flow-shop problem. Here we modify the criteria to sorting the lists by setup times instead of the processing times and apply in TMFSP problems. The complexity of it is $O(N\log N)$.

A greedy construction method (GCM)

This is an incremental construction method, which inserts jobs into the scheduled partial solution in a greedy manner. The idea is to schedule all jobs one by one and each insertion will increase the objective function by a minimum value. The algorithm is in Table 3. Its complexity is $O(N^3)$. We do not use the method that attempts all unscheduled jobs in each iteration, as it will degenerate the method to $O(N^4)$ complexity, while the result is no better in experiments.

An experiment is carried out on 630 test cases to compare the two algorithms. GCM generates better solutions than MJA does for 46.51% of all cases and for the rest, their performances are equally good. As better initial solutions may not produce better final solutions in any heuristic,

Table 3 Greedy construction method

```
Greedy Construction Method (GCM)

Step 1. Order the $N$ jobs by increasing sums of their setup times of both
operations.
Step 2. Take the first job and schedule its operations in sequential
order, thus form a partial solution.
Step 3. For $k = 2$ to $N$ do
Insert the $k$th job at the place, its first operation has $2 \times k - 1$ possi-
ble places and the second operation can be put in any possible place after
the first one. The objective is to minimize the partial makespan.
```

Table 4 Standard simulated annealing algorithm

```
Standard Simulated Annealing Algorithm (SSA)

Step 1. Let $i = 0$. Randomly choose an initial solution $x$ and set the
initial temperature $\beta_0, C_{\max} = f(x)$
Step 2. Randomly generate a solution $y$ from the neighborhood $S(x)$ of the
current solution $x$. Set $i = i + 1$.
Step 3. If $f(y) < f(x)$ set $C_{\max} = f(y)$. Replace $x$ by $y$ with the
probability $\min\{1, \exp((f(x) - f(y))/\beta_1)\}$
Step 4. Update $\beta_i$, if $\beta_i < stop\_temp$ or $C_{\max} = LB$, then stop, else go to Step
2.
```

solutions from both methods as well as random solutions will be the candidate initial solutions in the heuristics proposed in this section.

6.2. Simulated annealing approach

Simulated annealing approach has been applied to various combinational problems, including the scheduling problems (Osman and Potts, 1989). It has been shown that this approach allows for escape from local optima by accepting sequences that momentarily deteriorate the objective function under some specific condition (Zegordi et al., 1995).

Standard SA approach (SSA)

The standard simulated annealing algorithm is as given in Table 4.

In order to apply algorithm to our problem, the main components, including neighborhood structure, temperature, and termination criterion need to be determined.

We use three kinds of neighborhood structures, which are:

- *Swapping*: randomly swap two operations
- *Insertion*: randomly choose one operation and insert into another position
- *Block insertion*: randomly choose a block of operations and insert into an arbitrarily position.

Note that each newly generated solution should be validated by the validation step as described above. In our SSA, a random choice of these three neighborhood structures (*Mixture*) is used, as it generates a broader neighborhood space.

The temperature for SA includes initial temperature and annealing. We set the initial temperature β_0 at which there is 95% chance to accept a random neighbor of the initial solution. For annealing at iteration k, we set $\beta_{k+1} = \beta_k \times \alpha$, where the cooling factor α is less than and close to 1.

In this SSA, we set $\alpha = 0.9999$; $Stop\_temp = 0.0001$; $f(x) = C_{\max}(x)$. An experiment is carried out to check the performance of SSA. Besides using a random initial solution, we also use solutions from MJA and GCM as the candidates (Fig. 10). As we see, SSA with initial solution from GCM works the best. However, these do not work well as the input size N grows.

Limitation of SSA

Firstly, the experiments show that using random initial solution is not a good strategy. The random initial solutions may be quite far away from good solutions, and thus difficult to reach the good search areas.

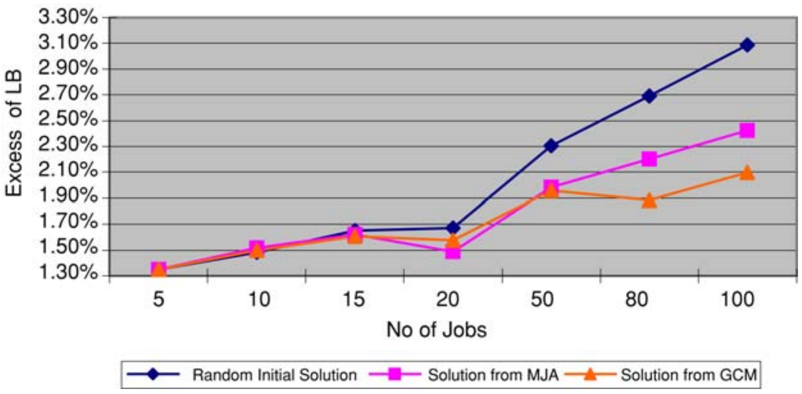


Fig. 10 Performance of SSA with different initial solutions

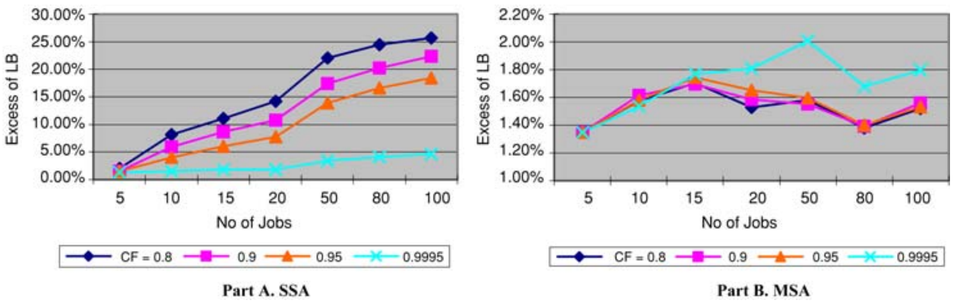


Fig. 11 Comparison of SA with different cooling factors

Secondly, slight changes of the cooling parameter may produce quite different results. In Fig. 11 Part A, we compare the performanc of SSA's with $\alpha = 0.8, 0.9, 0.95, 0.995$.

Next, SA usually stops when the temperature drops to a very small value, which is somewhat inefficient. Sometimes, good solutions or even optimal solutions have already been found when the temperature is still high, so that the rest of the search becomes useless. On the other hand, it may happen that even when the temperature has dropped to a very low value, the current best solution can be improved.

A modified SA approach (MSA)

In order to overcome the discussed limitations, we propose a modified version (MSA) of SA. Instead of choosing one neighbor in each iteration, K neighbors are randomly chosen. Then, one of these K neighbors is selected as a candidate to replace the current solution. Two possible transaction mechanisms are possible to choose the candidate. One is to select the best one whose objective value, makespan, is the minimum among the K neighbors (*Best*), the other is to select the first one that is better than the current solution in $C_{\max}$ (*First*). In the *First* approach, if no one is better, then the best one will be chosen, which becomes the same as *Best*. Extensive experimentation (to be described) shows that *Best* is slightly better in both the terms results and speed in TSFSP problems. With this modification, the probability of being replaced by bad neighbors decreases and thus the selection of cooling factor becomes less important. In Fig. 11

Table 5 Comparison of neighborhood structures

Metrics	Swapping	Insertion	Block Insertion
Average Excess of LB (%)	1.59	1.87	1.60
Average Time (ms)	3388	2843	8864

Table 6 Modified simulated annealing algorithm

Modified Simulated Annealing Algorithm

```
Step 1. Get the initial solution x and initial temperature $\beta_0$. Set cnt $= 0$
and oldcmx = $C_{\max}= f(x)$.
Step 2. Get K neighbors of the current solution x, and select one from
them using some transaction mechanism, say y is selected.
Step 3. If f(x) < $C_{\max}$, then update $C_{\max}= f(x)$. Replace x by y with the
probability min{1, exp((f(x) - f(y))/$\beta_c$)}.
Step 4. Update the current temperature $\beta_c$, i.e., do annealing step.
Step 5. If (oldcmx>$C_{\max}$) set cnt=0 and oldcmx=$C_{\max}$, otherwise increase cnt by
1.
Step 4. If cnt $\geq$ NOIPV or $C_{\max=LB}$, stop the algorithm, else, go to Step 2.
```

part B, the performance of MSA with $\alpha = 0.8, 0.9, 0.95, 0.995$ are compared. We can see, the change of performances of MSA is much less sensitive to α than it of SSA.

Another modification of SSA is on the termination criterion. Instead of waiting until the temperature drops to a very low value, we stop the algorithm if there is no improvement of the best solutions for continuous *NOIPV* iterations.

The three neighborhood structures given continue to be used. From the extensive experimentation, we realized that swapping and block insertion produce better solutions than insertion, while insertion gives the best speed (Table 5).

The basic algorithm of MSA is given in Table 6.

We now describe the extensive experimentation mentioned above. The aim is to find an approximately best combination of:

- Initial solutions from: Random, MJA, GCM;
- Neighborhood structure: Swapping, Insertion, Block Insertion and Mixture;
- Transaction Mechanism: Best, First;
- Cooling factor: $\alpha = 0.8, 0.9, 0.95, 0.9995$;
- Neighborhood Size $K = N, 2 \times N$;
- *NOIPV* $= 500, 750, 1000$.

The result from the 576 ($= 3 \times 4 \times 2 \times 4 \times 2 \times 3$) combinations was the following combination: Initial solution from GCM, Mixture neighborhood structure, *Best* transaction mechanism, $\alpha = 0.8$, $K = 2N$ and *NOIPV* $= 1000$.

In MSA, though the new termination criterion makes the algorithm more flexible with the stop temperature, it has a chance of being stuck at some local optima. Thus, we extend MSA by adding in the intensification and diversification move, i.e., MSA can be restarted several times. We choose to restart from an arbitrarily chosen best solution found so far. With this extension, the chance of being trapped at local optima is reduced. We stop restarting when there are more than L continuous iterations of restarting that do not improve the makespan of the best solutions.

Here, we compare the several kinds of SA discussed (Fig. 12). For the SSA, we use the best whose initial solution is from GCM. Modified SA is without restart (MSA) and the extensions are

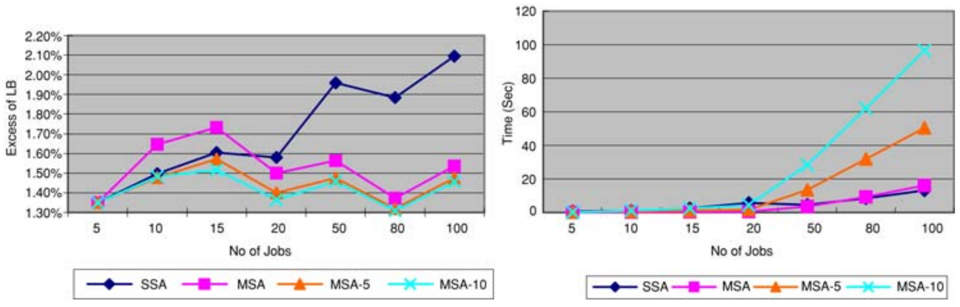


Fig. 12 Comparison of standard SA and modified SA

with $L = 5$ (MSA-5) and 10 (MSA-10). We can see that the MSA, particularly the ones with the restart extension, perform much better than SSA, especially when the number of jobs grows large.

6.3. Tabu search approach

Tabu search, introduced by Glover (1989), is another successful strategy for combinatorial optimization problems. It constrains search by defining a forbidden list of moves, i.e. tabu list, to prevent infinite loops and avoids getting stuck in local optima by accepting a move that yields the most favorable changes in the objective function value (even worse than the current value) subject to the restriction that only non-tabu moves are permitted.

We look at the main components implemented in our TS first. We use the same initial solutions and neighborhood structures as in SA. An interesting finding is that when the input, *N*, is small, the insertion works best among all the neighborhood structures and when *N* grows larger, swapping performs best (Fig. 13). A possible explanation for this is the design of the tabu list (to be introduced soon). Two tabu moves are recorded for swapping while only one is recorded for insertion. When *N* is small and two operations are restricted, the size of permitted neighbors is reduced significantly compared to the size of the total neighborhood space, And this compromises the advantage offered by the tabu list. Insertion outperforms swapping, since it forbids

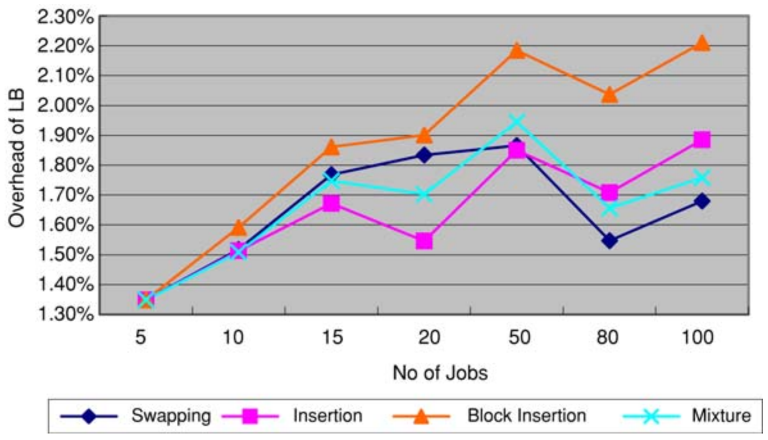


Fig. 13 Comparison of different neighborhood structures in TS

fewer neighbors. As N grows, the reduced neighbors become negligible compared with the huge neighborhood space; on the contrary, they exert the effect of tabu lists better. Thus swapping outperforms insertion for large N. Block insertion does not work well for the Tabu structure, and degenerates the performance of the fourth choice, mixture.

For the transaction mechanism, only *First* is considered, as *Best* appears very slow in experiments because of our tabu list structure. The use of tabu list is to prevent loops at local optima. To put the full sequence in the tabu list is not efficient in space or speed, especially when job size grows large. So we choose to keep tabu moving in the list in the following way: whenever an operation o is moved out of its current position p, (o, p) is inserted into the tabu list. Following this, o is not allowed to be put back to position p while (o, p) is still in the tabu list. Another choice is to keep the objective value $C_{\max}$ in the tabu list: The $C_{\max}$'s of sequences are inserted into the tabu list, and if the makespan of a new sequence is equal to some value in the tabu list, this new sequence is forbidden. Both of these two kinds of tabu list will reject some sequences that have not been selected so far, but are much faster than if the full sequence is checked. Experiment shows that using the first gives results better than using the second method for 36.2% of all 630 test cases while results are worst for only 0.3%. Thus, we recorded that tabu moves in our tabu list.

There are situations, when we want to override some tabu moves in order to get better result, so an aspiration criterion concept is used. We use the simplest aspiration criterion: a tabu move is accepted if it produces a solution better than the best obtained so far.

Another feature of TS is to use the "long-term memory" to jump out of local optima, which is intensification and diversification. An intensification scheme often takes the form of reinforcing attributes of good solutions while a diversification scheme typically consists of driving the search into regions, not yet explored. The combined use of intensification and diversification is aimed at finding high-quality, solutions. A frequency-matrix strategy has been used successfully for the classical flow-shop problem (Ben-Daya and Al-Fawzan, 1998). To apply this in our problem, a $2N \times 2N$ matrix is needed instead of an $N \times N$ one, as an entry is thus needed for each operation. The idea is to restart the algorithm not only in a different region of the solution space but also in a region that potentially contains good solutions. It therefore combines the goals of intensification and diversification, but where the validation step is needed to make the solution feasible. Besides this strategy, we can also restart from a random initial solution or a randomly selected current best solution. By using random initial solution, only the diversification goal is reached while intensification is not possible. By restarting from a best solution, the goal of intensification is easily reached. Moreover, since the tabu list is empty when restarting, the search area is now larger than it was when the solution was obtained in previous iterations. In addition, because a best solution is randomly selected for each restart, the search areas are different even when the best solutions do not improve in continuous iterations. The performance of these three strategies from experiments are compared in Fig. 14. From this, we see that the *Best* restart strategy works slightly better than the others.

The TS algorithm is stopped when there is no improvement for the best solutions among L consecutive calls of the intensification and diversification scheme. Experiment shows that the algorithm gets stable after $L \geq 20$. Inside each call, it stops when there are continuous *NOIPV* iterations of transactions that do not improve the best solutions.

The pseudo-code of TS algorithm for TSFSP problem is in Table 7.

Now we describe the extensive experiment. Candidate components and parameters are:

- Initial Solution: Random, MJA, GCM;
- Neighborhood Structure: Swapping, Insertion, Block Insertion, Mixture;
- Intensification and Diversification: Frequency-matrix, Random, Best restart;

Table 7 Tabu search algorithm

Tabu Search Algorithm
Step 1. Choose an initial solution s_0 and set $s^* = s_0, s_i = s_0, 1 = 0$ and *bestcmx* = *f*(s_0).
Step 2. Set *cnt* = *0* and $C_{\max}$=*oldcmx*=*f*(s^*).
Step 3. Generate a subset V^* of solution in *N*(s_i) such that either it does not contain a tabu move or it satisfies the aspiration condition.
Step 4. Choose a sequence s_j from V^* using the First strategy, and set $s_i = s_j$.
Step 5. If *f*(s_i)<*f*(s^*) then set $s^* = s_i$. If *f*(s^*) < $C_{\max}$, update $C_{\max}$=*f*(s^*).
Step 6. Update tabu list.
Step 7. If (*oldcmx* > $C_{\max}$) set *cnt* = *0* and *oldcmx* = $C_{\max}$, else *cnt* = *cnt+1*.
Step 9. If (*cnt* < *NOIPV*) go to Step 3, else if (*bestcmx* > $C_{\max}$) then set *bestcmx* = $C_{\max}$ and *1=0*, else *1=1+1*, if (*1* < *L*) set s^* to be initial solution of restarting then go to Step 2, otherwise, terminate.

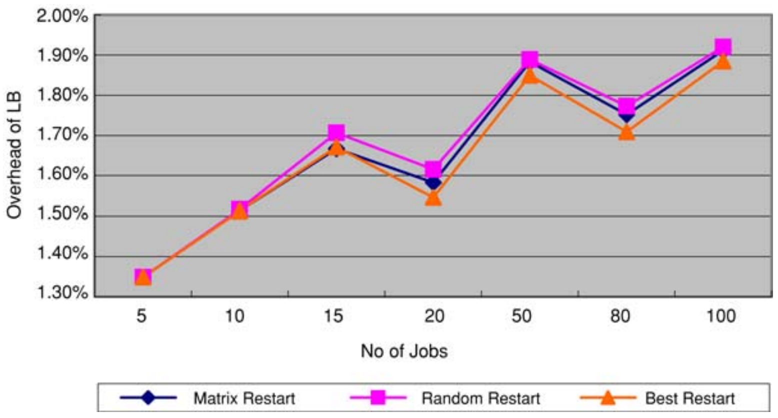


Fig. 14 Comparison of three restart strategies

- *NOIPV* : 500, 800, 1000;
- Neighborhood Size *NS* : N, $2 \times N$;
- Tabu List Size *tno*: 5, 7, 9.

We set $L = 1$ in the experiment to save time. The best one among all 648 ($= 3 \times 4 \times 3 \times 3 \times 2 \times 3$) combinations is: initial solution from GCM, neighborhood structure as insertion ($N \leq 50$) and swapping ($N > 50$), restart from a best solution, $NOIPV = 1000$, $NS = 2 \times N$ and $tno = 5$.

6.4. Genetic algorithm

Genetic algorithms (GA), developed by Holland (1975), are based on the idea of genetic evolution and fitness. The approach has been effective for many computationally complex problems. We examine some of its main components.

The first component is initial population. Both fixed and variable population size is possible. For our TSFSP problems, experiments show that GA with various population size ($5 \times N$ and $10 \times N$, where N is the number of jobs) works no better than with fix size; furthermore, it is much slower. Hence, only fix population size, say *POPSIZ*, is used. Then the initial population is constructed as: the first solution is from GCM, and the second solution is from MJA. The third is the mutation of this first solution, and mutating the second solution we get the fourth. This is repeated until the population size, *POPSIZ*, is reached.

Next is the fitness value and selection probability. Each member $s(i)$ in the population has a fitness value $f(s(i))$, and its selection probability is calculated based on $f(s(i))$, which is equal to $f(s(i))$ divided by $\sum_j f(s(j))$. Let's denote $C_{\max}$ of $s(i)$ as $C(s(i))$ and the maximum of $C(s(i))$ as *CMAX*. Then, $f(s(i))$ can be calculated as the deviation of $C(s(i))$ from *CMAX*(*linear fitness*), namely $CMAX - C(s(i))$. To realize the stronger selection pressure, we can also use the square of the deviation as the fitness (*quadratic fitness*).

An important component of genetic algorithm is the genetic operator, which consists of crossovers and mutations.

Crossover: Various crossover operators can be used in this TSFSP problem: Goldberg's PMX operator (Goldberg, 1989); Cyclic Crossover (CX) (Oliver et al., 1987); One-point crossover (1PC); Two-point crossover (Version I, II & III) (2PC-1, 2PC-2, 2PC-3); Position-based crossover (Version I, II) (PBC-1, PBC-2) (Murata et al., 1996). Details of these operators are in Appendix C. The newly generated sequence must be validated before use. Comparisons of these eight kinds of crossover from the extensive experiment are in Table 8; both the average excess of LB and CPU time are compared. Here, PMX is the best crossover.

Mutation: Various mutation operators can be used. Same as crossover, the mutated sequence must be processed by the validation step, in order to be feasible. Four kinds of mutation operators are tested in the extensive experiments, which are: *Adjacent two-operation change* (*J2C*)—the adjacent two operations to be swapped are randomly selected; *Arbitrary two-operation change* (*A2C*)—the two operations to be swapped are arbitrarily chosen; *Arbitrary three-operation change* (*A3C*)—the three operations to be changed are arbitrarily selected, and the order of them after the mutation is randomly specified (Murata et al., 1996); *Shift change* (*SC*)—an operation at one arbitrarily selected position is removed and put at another position. A comparison is in Table 9. From this, we see that *A2C* is the best both for the objective value and speed.

Table 8 Comparisons of crossover operators

Crossover	Avg Ovhd (%)	Time (ms)
PMX	***1.69***	***2486***
CX	1.72	4717
1PC	1.74	2657
2PC-1	1.73	2681
2PC-2	1.73	2997
2PC-3	1.73	4638
PBC-1	1.78	6143
PBC-2	1.80	3390

Table 9 Comparisons of crossover operators

Mutation	Avg Ovhd (%)	Time (ms)
J2C	1.84	2500
A2C	***1.69***	***2486***
A3C	1.71	2588
SC	1.74	2608

Table 10 Combinations of elitist strategy and fitness value

Ovhd(%)/Time(sec)	Elitist-1	Elitist-2
Linear fitness	1.69/47	**1.52/28**
Quadratic fitness	1.55/79	1.54/29

Next, we come to the elitist strategy that will keep the good solutions in the old population. Two possible strategies are used in our experiment:

- The new generation consists of all the newly generated children with the worst one is replaced by the best solution from the old population. (Elitist-1)
- Check each of the newly generated children, if its objective value is better than the worst of the old population and there is no old member with the same objective value, it will be inserted into the population and a worst old member is removed. This makes sure that only good member will be kept in the population. In addition, insert only good solutions whose objective values are not same as some old ones, which prevent the population from converging too fast. It may reject some good new solutions, however. (Elitist-2)

Elitist-2 is faster than the first; the reason is that it converges faster as it will select only good members. An interesting finding from the experiment is that the performance of the two kinds of fitness values introduped earlier are related to the choice of elitist strategy (Table 10). When Elitist-1 is used, the quadratic fitness is obviously better than the linear fitness; while using the second, linear fitness is slightly better. A reasonable explanation for this is that when using Elitist-1, quadratic fitness realizes the stronger selection pressure, so better members are more likely to be selected as parents, thus producing a better result. However, when using Elitist-2, each generation only keeps good solutions, and the search areas remain around the good members, and so it is better to give relatively similar chance to each member in the population, which is what linear fitness does. As the combination of linear fitness and Elitist-2 produce best results among all, we will employ this in the GA.

Next, we examine the termination criterion. We can use number of successive generations of not improving best objective value, *GENER*, as the termination criteria. We set $GENER = 500N$.

We now provide the outline of the genetic algorithm in Table 11.

We need to find out the best combinations of the following from experimentation:

- Elitist Strategy: Elitist-1, Elitist-2;
- Crossover: PMX, CX, $1PC$, $2PC$-1, $2PC$-2, $2PC$-3, PBC-1, PBC-2;
- Mutation: $J2C$, $A2C$, $A3C$, SC;
- $POPSIZ = 60, 80, 100$;
- Crossover rate $Pc = 0.8, 0.9, 1.0$;
- Mutation rate $Pm = 0.1, 0.2, 0.5, 0.8, 1.0$.

In total, there are $2 \times 8 \times 4 \times 3 \times 3 \times 5 = 2880$ combinations. (To save time, we set $GENER = 50000/POPSIZ$ in this experiment). The best combination from the experiment turned out to be the following combination: Elitist-2; *PMX*; A2C; $POPSIZ = 60$; $Pc = 0.9$ and $Pm = 0.5$.

Now we compare the performance of the first several heuristics, SA, TS, and GA (Fig. 15). From the comparison, we see that the extended version of MSA works best among all. MSA performs not well for small *N*, the reason for this is as we have explained earlier, that the termination criterion used possibly terminates the process too early. This is the motivation for an extended version. The TS approach does not work as well as SA. The reason may be because of the difference in the way they treat bad neighbors. SA accepts bad neighbors with a probability, which is reduced as the temperature cools down. However, TS accepts them if all the

Table 11 Genetic algorithm

```
Genetic Algoirithm

Step 1. (Initialization) Determine the initial population, S(t), where
t=0, the population size, POPSIZ and GENER. Set gen = 0, oldcmx=$C_{max}$.
Step 2. (Calculation) Calculate the fitness value and the selection
probability of each member, f(s(t)) and P(s(t)) separately, for
population, S(t).
Step 3. (Selection) Select a pair of members (parents) that will be used
for reproduction via the selection probability.
Step 4. (Crossover+Mutation) Apply Crossover operator to the two parents
with the crossover rate Pc, otherwise, parents remain unchanged. Then,
mutation operator is applied to the newly generated children with the
mutation rate Pm. Update $C_{max}$ accordingly.
Step 5. (Generation) Repeat from Step 3 until the size of the newly
generated children reaches POPSIZ.
Step 6. (Elitist) Apply the elitist strategy to get the new generation.
Step 7. If (oldcmx > $C_{max}$) set gen = 0 otherwise gen = gen+1. If ($gen \geq GENER$)
then stop else go to Step 2.
```

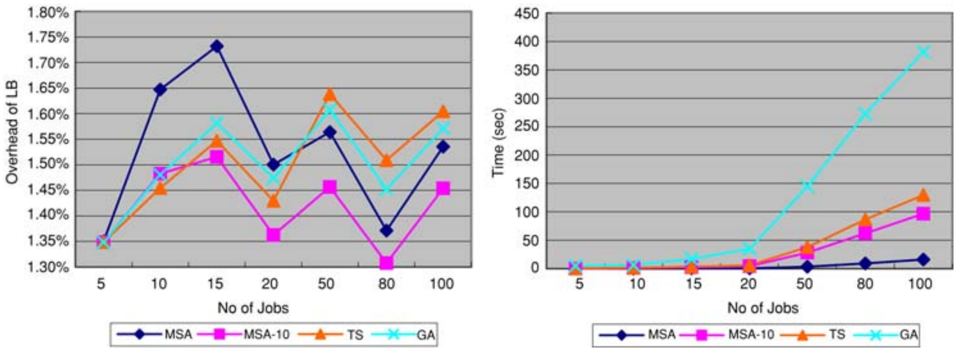


Fig. 15 Comparison of performances of SA, TS and GA

2N candidate neighbors are worse. This shortcoming may drag the search into a relative bad area. GA performs well only for small inputs, as it enforces only the high diversity of searching space, but not enough for intensification. Also, it is slow when compared with the other methods.

In the following heuristics, a local search step is involved. The Mixture neighborhood structure is used in local search. Because of the large neighborhood size, it is usually impossible to search everywhere. Thus we decided to stop the search for local optima after continuously getting *LS* neighbors that do not improve the current solution. We categorize those heuristics with local search step as slow-heuristics as the local search step is usually slow, and others as fast-heuristics.

6.5. Greedy randomized adaptive search procedures

A meta-heuristic called Greedy Randomized Adaptive Search procedure (GRASP) is introduced in this section. It is a multi-start or iterative process, in which each iteration consists of two phases: construction and local search. The construction phase builds a feasible solution using some greedy method, and then its neighborhood is investigated until local optima are found during the local search phase (Resende and Ribeiro, 2002).

Two ways are possible for the construction step. The first (*Operation-construction*) is to treat each operation as the basic element for incorporation. Its complexity is $O(N^3)$. The other way (*Job-construction*) is to consider each job as an incorporation element. In *Job-construction*, the first operation of a selected job e is firstly inserted into a position i of the constructed partial solution, then the second operation is inserted into position j, where $j > i$, due to the precedence rule of flow-shop. Experiments shows this way of construction is very slow, as its complexity is $O(N^4)$. We can modify it by fixing the order of jobs to be incorporated (*Block-construction*) and the pseudo-code is in Table 12. *Block-construction* method reduces the complexity to $O(N^3)$.

In all of the proposed construction methods, the restricted candidate list *RCL* is associated with a threshold parameter $\alpha \in [0, 1]$ and one element at a time from *RCL* is chosen to be incorporated into the partial solution. We can assign equal probabilities for being chosen to the elements in the list; however, any prpbability distribution can also be used to bias the selection toward some particular candidates. Usually, they are based on the rank $r(e)$ assigned to each candidate element e, according to $c'(e)$. The bias functions used in our extensive experiment include: random bias: $\text{bias}(r) = 1$; linear bias: $\text{bias}(r) = 1/r$; log bias: $\text{bias}(r) = \log^{-1}(r + 1)$; exponential bias: $\text{bias}(r) = e^{-r}$ and polynomial bias: $\text{bias}(r) = r^{-k}$. Then, the selection probability of e is taken as $sp(e) = \frac{\text{bias}(r(e))}{\sum_{e' \in RCL} \text{bias}(r(e'))}$.

The local search step is as described. The more neighbors we want to search in this step, the slower the algorithm will be. We consider the trade off between the quality of local search and the CPU time it requires. Thus, we choose to stop the local search after finding continuous *LS* neighbors that are no better than the current one, where $LS = 100 \times N$.

We tested experimentally to find the best combination of:

Table 12 Block_construction method

```
Block_construction Method

Order the $n$ jobs from input by increasing sums of the setup times of their
two operations and then put them into a vector $V$.
Solution ← Ø
while V is not empty do
    Select the first job $J$ from $V$, for each possible position $p1$ in
    Solution, put $O_{J1}$ at $p1$. And then find the position $p2$ so that the in-
    cremental cost is the smallest when $O_{J2}$ is at $p2$. Denote ($p1$, $p2$) by e
    and put e into $CL$. So, $c'(e)$ = the incremental cost of the new solu-
    tion where job $J$ is incorporated into the partial solution and its two
    operations are put at position $p1$ and $p2$ separately.
    $C_{\min} = \min\{c'(e) \mid e \in CL\}$
    $C_{\max} = \max\{c(e) \mid e \in CL\}$
    $RCL = \{e \in CL \mid c'(e) \leq C_{\min} + \alpha(C_{\max} - C_{\min})\}$
    Calculate the selection probability of each element in $RCL$ using some
    bias function
    Select element $s$ from the $RCL$ via the selection probability
    Construct Solution with the two operations of $J$ is put at the posi-
    tions of $S$
    Remove $J$ from $V$
End
Return Solution
```

- Construction Method: Operation_construction, Block_construction
- α : 0, 0.1, 0.2, 0.4, 0.5, 0.6, 0.8, 0.9, 1
- Bias Function: random, linear, log, exponential, polynomial

The best among all 90 $(= 2 \times 9 \times 5)$ combinations is obtained using: *block_construction*, $\alpha = 0.9$ and polynomial bias function with $k = 2$. With this α value, the initial solutions from the construction step have quite high diversity and most likely are better than a random initial solution; the *LS* step enforces the intensification factor. Thus, GRASP is expected to be better than the pure *LS* procedures as experiments will verify.

Reactive GRASP

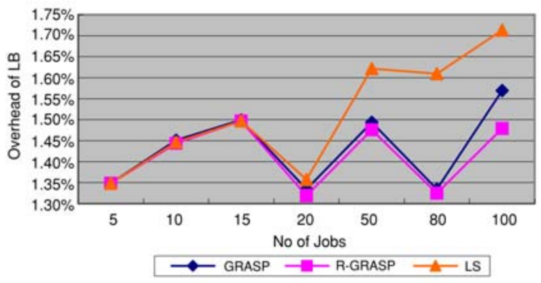
In GRASP, usually, a fix *RCL* parameter α cannot work well for all test cases. In Reactive GRASP, α is selected at each iteration from a discrete set Ψ of possible values and the selection is guided by the solutions found along the previous iterations. Let $\Psi = \{\alpha_1, \ldots, \alpha_m\}$.The probabilities associated with the choice of each value are initially equal to $p_i = m^{-1}, i = 1, \ldots, m$. Let A_i be the average value of all solutions found using $\alpha = \alpha_i$. Then the selection probabilities are periodically re-evaluated by taking $p_i = q_i / \sum_j q_j$, with $q_i = A_i^{-r}$. The value of q_i will be larger for α_i leading to the better solutions on average. Alternatively, we can rank A_i and then use the bias functions to assign the probabilities. *Block_construction* method is used in the construction step, since it works the best in our GRASP. The possible values for α are the same as what we use in GRASP.

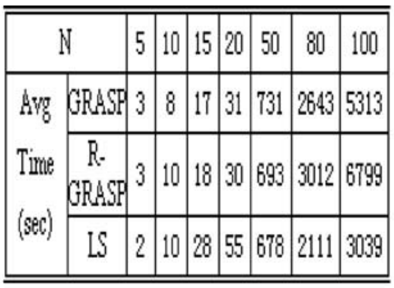
In the extensive experiment, we set the period for re-evaluating the selection probability of each α_i as 50 iterations. (The first time to re-evaluate is after 100 iterations, as we expect each α has been used). We try to find the best combination of:

- Bias function for *RCL*: random, linear, log, exponential, polynomial;
- Selection Probability: First way with $r = 1, 10, 50, 100, 250, 500, 1000$
  Second way with the same five bias functions as for *RCL*.

The best among all 60 $(= 5 \times 7 + 5 \times 5)$ combinations is: polynomial bias function for *RCL* with $k = 2$ and first way to calculate selection probability with $r = 500$.

Now we compare the performance of the two kinds of GRASP with LS procedure. All of them have the same termination criteria, that is, stopping after continuously 500 iterations that do not improve the best solutions. In LS procedure, the stop criterion for each local search is successively finding 500*N* non-improving neighbors. Figure 16 shows that GRASP works much better than





N		5	10	15	20	50	80	100
Avg Time (sec)	GRASP	3	8	17	31	731	2643	5313
	R-GRASP	3	10	18	30	693	3012	6799
	LS	2	10	28	55	678	2111	3039

Fig. 16 Comparison of GRASP, R-GRASP and LS

Table 13 Genetic local search approach

```
Procedure Genetic Local Search

Step 1. (Initialization)
Step 2. (local search for initial population) For each member in the
initial population, apply LS on it until the local optimum is found.
Step 3. (Calculation)
Step 4. (Selection)
Step 5. (Crossover + Mutation)
Step 6. Apply LS for the newly generated solutions.
Step 7. (Generation)
Step 8. (Elitist)
Step 9. If the termination criterion is met, then stop else go to Step 3.
```

LS. With the re-evaluation of α in Reactive-GRASP, the performance is even improved, yet not very obvious.

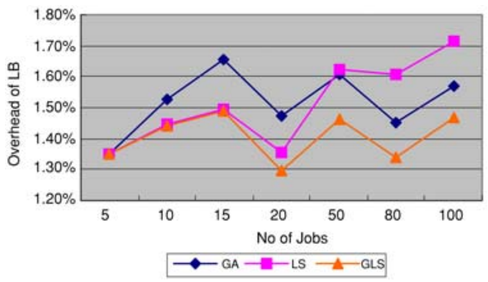
6.6. Genetic local search approach

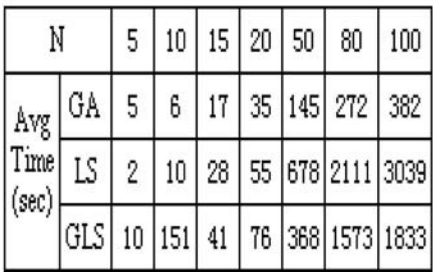
In this section, we propose a hybrid method, Genetic Local Search (GLS), that merges LS with GA. For each member in the initial population, as well as the newly generated child, a local search step is applied. The algorithm is in Table 13. (Refer to Section 6.4 for details)

One difficulty of GLS is the enormous computation time of local search step for the members. We consider the trade off between the quality of result and CPU time required, and modify the previous GA.

Firstly, we use variable population size in GLS as: *POPSIZ* $= 1000/N$. To avoid too large or too small a *POPSIZ*, we set the maximum value for it as 80 and minimum 20. Random initial population is used with solutions from MJA and GCM are included. We do not use the same initial solution as what we do in previous GA. The reason is that when N is large and *POPSIZ* is small, the sequences got after LS in the previous population will converge very fast: We also modify the termination criteria for LS and GA. LS step is stopped if the current solution is not improved after successively finding $500N$ neighbors. And, if the best solution is not improved for 10 generations, then GA is stopped. Again, we tune the parameters and then set crossover rate $Pc = 0.8$ and Mutation rate $Pm = 0.1$.

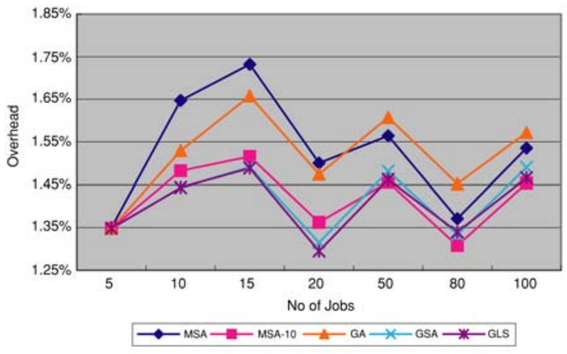
Next, we compare the performance of the previous Genetic Algorithm, Local Search with this hybrid algorithm (Fig. 17). Here, GLS gives much better solutions than GA and LS do, as it exploits both the diversification advantage of GA and intensification advantage of LS.

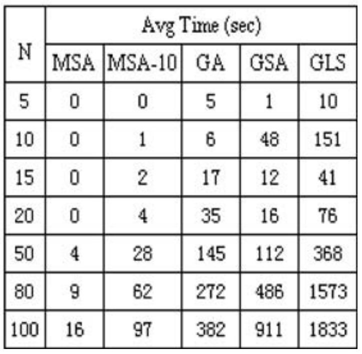




N		5	10	15	20	50	80	100
Avg Time (sec)	GA	5	6	17	35	145	272	382
	LS	2	10	28	55	678	2111	3039
	GLS	10	151	41	76	368	1573	1833

Fig. 17 Comparison of GA, LS and GLS





N	Avg Time (sec)				
	MSA	MSA-10	GA	GSA	GLS
5	0	0	5	1	10
10	0	1	6	48	151
15	0	2	17	12	41
20	0	4	35	16	76
50	4	28	145	112	368
80	9	62	272	486	1573
100	16	97	382	911	1833

Fig. 18 Comparison of SA, GA, GSA and GLS

6.7. Genetic simulated annealing approach

Alternatively, we can use simulated annealing to replace the local search step in the GLS approach. That is, using SA in steps 2 and 6 of the algorithm in Table 13. Here SA refers to our Modified SA algorithm given earlier. Further to this, a constant temperature is used to avoid extreme deterioration of the current solution during the initial state of annealing with high temperatures. (It is difficult to find a suitable constant temperature for all cases). We can only approximate a good value by getting the temperature at which the algorithm accepts a rate R of bad solutions among 100 randomly generated neighbors. We tested the performance with $R = 0, 0.05, 0.1, 0.2, 0.5$, and 0.8, where experiments showed that the best performance is attained when $R = 0$. However, this is not equal to local search, as the temperature is calculated based on the first randomly generated 100 neighbors of the initial sequence, which may also accept worse solutions each time replacing the current sequence. Also they are different in the way candidate neighbor are chosen. In addition, we modify the stop criterion for SA by setting $NOIPV = 500$.

We then compare the performance of SA, GA, GSA, and GLS (Fig. 18). Here SA includes both modified SA without and with restarting ($L = 10$). The performance of GSA is much better than GA and SA without restarting. It is also better than the extended version of SA for small number of jobs. As the number grows large, its performance degenerates may be because of the intensification is not as good as in simulated annealing. From the aspect of running time, we can see that, GSA is much faster than GLS. The advantage of GSA may come from the transaction strategy. It evaluates 2N neighbors at a time and then chooses the best to be for the candidate replacement. Then, it is more likely to transform to a good solution and thus converge more quickly. Because of our specific termination criteria, it ends much earlier than GLS does.

6.8. A new hybrid method of GRASP and GSA

We propose a hybrid heuristic that combines our previous GRASP and GSA. The algorithm is same as GSA, except that we use solutions from the construction step of GRASP as the initial population. In the construction step, we use the fixed α, value, and $\alpha = 0.25, 0.5$, and 0.75 are tested. Experiments show that the method works best when $\alpha = 0.5$. With this α-value, the initial population still keeps a high diversity. Moreover, the *Block_construction* method in GRASP ensures the quality of the solutions. We compare the performance of this hybrid method with previous GSA algorithm in Fig. 19, from which we see that the hybrid method is better than

Table 14 Comparisons to optimal solutions for TMFSP problems

HEURISTIC	$N = 5$				$N = 10$			
	Opt no	Optimal (100%)	Avg Excess (100%)	Max Excess (100%)	Opt no	Optimal 100%	Avg Excess (100%)	Max Excess (100%)
TS	90	100	0	0	82	91.1	0.010	0.291
GA	90	100	0	0	83	92.9	0.012	0.465
MSA	90	100	0	0	59	65.6	0.106	1.164
MSA-10	90	100	0	0	76	84.4	0.018	0.465
GRASP	90	100	0	0	82	91.1	0.009	0.211
Reactive-GRASP	90	100	0	0	87	96.7	0.002	0.169
GSA	90	100	0	0	83	92.9	0.012	0.465
GLS	90	100	0	0	87	96.7	0.002	0.151
GLS + GRASP	90	100	0	0	86	95.6	0.001	0.076

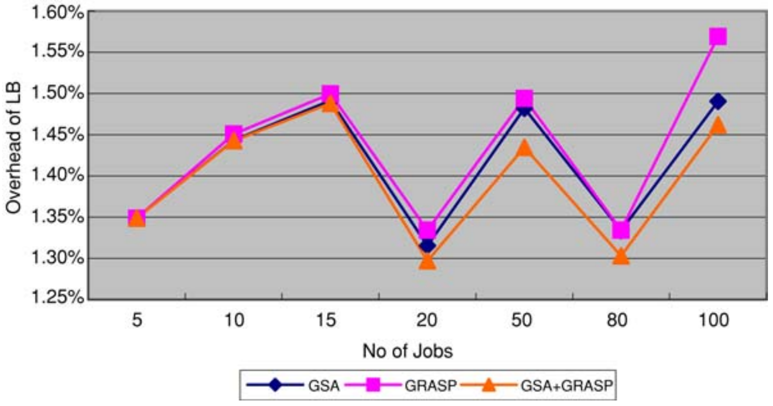


Fig. 19 Comparison of GSA, GRASP and Hybrid

both GLS and GRASP. However, as GSA has already ensured a high level of diversification and intensification, the improvement is limited.

6.9. Comparisons to optimal solutions

Finally, we compare the solutions from each proposed heuristic for small inputs with optimal solutions. The comparisons are listed in Table 14. We see that all heuristics can find optimal or nearly optimal solutions for small inputs.

7. Problem extension

In order to generalize the problem, we remove the constant processing time constraint for TMFS problems. We use the same representation and validation of solutions for TMFS problems as we did for TMFSP problems. The test cases are generated with the same property but the processing times are uniformly distributed frdm 0 to 1000. New *LBs* are calculated.

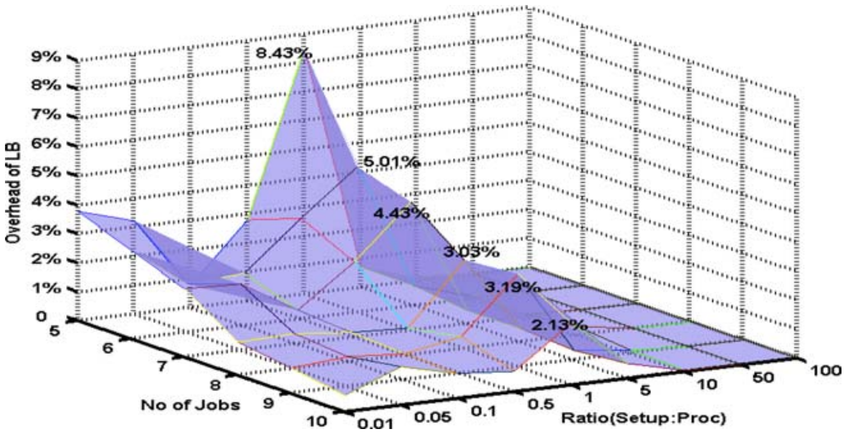


Fig. 20 Excess: LB to optimal solution for TMFS problems

Proposition 6. *A Lower Bound of* $C_{\max}$ *for the solutions of TMFS is max* $\{C'_1, C'_2, C'_3\}$ *where*
$C'_1 := \sum_{i=1}^{N} \sum_{j=1}^{2} S_{ij} + \mathit{minimum}\ P_{i2}$
$C'_2 := \sum_{i=1}^{N} (S_{i1} + P_{i1}) + \mathit{minimum}\,\{\mathit{max}\{S_{j2} - P_{j1}, 0\} + P_{j2}, j = 1, 2, \ldots, N\}$
$C'_3 := \sum_{i=1}^{N} (S_{i2} + P_{i2}) + \mathit{minimum}\ S_{i1}$

The proof is similar as for TMFSP problems, and is omitted. Optimal solutions for small N are compared to this lower bound, *LB*. Again, $\mathit{excess} := (LB - OPT)/OPT$ and *OPT* is the makespan of optimal solutions (Fig. 20). The LB's become tighter as N grows larger. However, it is a little loose for all N when the ratio of average setup time to processing time, R, is near 1.

All heuristics discussed for TMFSP problems can be applied into the generalized problems. A modification is made to the two greedy initial solutions: the criteria for sorting became the sum of both setup and processing times of the two operations, as processing times are no longer constant. The same modification is made to GRASP. In order to make the heuristics work well for the new problem, we ran again some of the experiments to tune the parameters, thereby, obtaining the best combinations as follows:

- SA: *GCM*; Mixture neighborhood; Best transaction; $\alpha = 0.95$; $K = 2N$ and $NOIPV = 1000$.
- TS: *GCM*; Mixture neighborhood; Best Restart; $NOIPV = 1000$; $NS = 2N$ and $tno = 5$;
- GA: Elitist-2; linear fitness; *CX*; *A2C*; $POPSIZ = 100; Pc = 0.9; Pm = 1$;

Here, the main reason that the combinations for TMFS problems are different from those for TMFSP problems is the processing times. In TMFSP, since all the processing times are constant, the differences of $C_{\max}$ among different sequences are determined by the setup times only. But as this constant constraint is removed, processing times also take effect. When a sequence is changed, the $C_{\max}$ may change much more than before. This makes the neighborhoods and generic operators work differently from TMFSP problems.

The comparisons of the performance of different heuristics are given in Table 14 and comparisons to optimal solutions for small inputs are in Table 15.

Every heuristic is able to find close-to-optimal solutions for small inputs. GA and TS are able to find good solutions whose excessses to the lower bounds are small. MSA does not work well for small input data; here, the reason is that it may converge too quickly becoming stuck at some local optima. To address this, we added the diversification method into it, restarting several times

Table 15 Comparisons of heuristics to TMFS problems

	N	5	10	15	20	50	80	100	AVG
Ovhd	MSA	2.20	0.91	0.75	0.66	0.41	0.33	0.42	0.81
of LB	Extended MSA	2.20	0.69	0.50	0.48	0.29	0.32	0.25	0.68
(%)	TS	2.20	0.71	0.67	0.66	0.47	0.47	0.36	0.79
	GA	2.20	0.73	0.67	0.65	0.54	0.56	0.45	0.83
	GRASP	2.20	0.68	0.50	0.56	0.44	0.51	0.43	0.76
	GSA	2.20	0.68	0.45	0.43	0.36	0.31	0.21	0.66
	GRASP + GSA	2.20	0.68	0.45	0.43	0.31	0.30	0.21	0.65
Avg	MSA	0	0	1	2	18	31	8	9
Time (sec)	Extended MSA	1	7	16	23	245	375	121	112
	TS	2	7	19	29	484	908	151	229
	GA	3	7	12	23	285	497	98	132
	GRASP		6	13	30	536	1852	2706	735
	GSA	5	12	56	120	521	902	1528	449
	GRASP + GSA	6	15	51	140	550	1021	1434	459

Table 16 Comparison to optimal solutions for TMFS problems

	$N = 5$				$N = 10$			
	Opt no	Optimal (100%)	Avg excess (100%)	Max excess (100%)	Opt no	Optimal 100%	Avg excess (100%)	Max excess (100%)
TS	90	100	0	0	83	92.9	0.033	1.240
GA	90	100	0	0	83	92.9	0.054	1.236
MSA	89	98.9	0.023	2.092	75	83.3	0.223	3.962
Extended MSA	90	100	0	0	89	98.9	0.006	0.550
GRASP	90	100	0	0	89	98.9	0.001	0.080
GSA	90	100	0	0	90	100	0	0
GSA + GRASP	90	100	0	0	90	100	0	0

from a good area. With this extension, its performance improved significantly, while its speed remained good. The heuristics with local search step, i.e., GRASP, GSA, and GRASP + GSA, work well for small inputs data. Yet, they are quite slow compared to the others. GSA produces better solutions than the extended MSA does. However, the improvement from merging with the construction step of GRASP is negligible in TMFS problems.

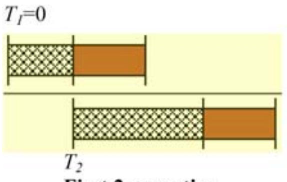
8. Conclusion

In this paper, two-machine flow-shop problems with a single setup server is surveyed. We first considered problems, where all processing times are constant. A lower bound of $C_{\max}$ is proposed for this kind of problems. Some special cases are proven to be solved in polynomial time. For the general cases, which are known to be NP-complete, heuristics are proposed. Two categories of heuristics are developed. The first is fast, which include the SA, TS, and GA approaches. We proposed a modified version of SA, which works much better than the standard one, as well as TA and GA. The second category includes heuristics with a local search step. GRASP is first introduced and a reactive version is found to be much more flexible for the restricted candidate list. A hybrid method which merges both GA and LS is introduced. It produced good results but was slower. Replacing the LS with SA, the speed was improved significantly. The construction step of GRASP is then merged with GSA, which improved the combination of GA and SA. We then extended our study to TMFS problems removing the constant processing time constraint. With some modification on the lower bounds and heuristics, TMFS problems are also solved. Experiments show that the heuristics developed, obtain nearly optimal solutions.

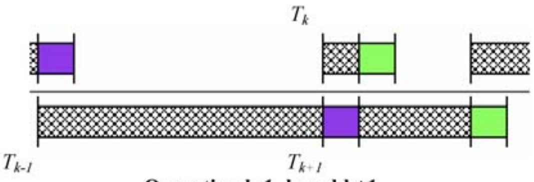
Appendix

A. Proof of $T_{i+1} = T_i + S_i$ for Proposition 4.

Proof: $T_1 = 0$. If we start setup of the second scheduled operation on machine 2, the setup will finish no earlier than the end of processing of the first scheduled operation as $P_1 \leq S2$. Thus, $T_2 = T_1 + S_1$.



First 2 operation

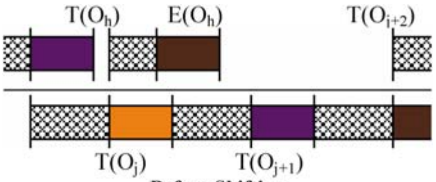


Operation k-1, k and k+1

We assume $T_k = T_{k-1} + S_{k-1}$, then at time $T_k + S_k$ the processing of the $(k-1)$th scheduled operation has finished processing because of $P_{k-1} \leq S_k$ and end time for the $(k-1)$th scheduled operation $= T_{k-1} + S_{k-1} + P_{k-1} \leq T_k + S_k$. (Figure) So, the setup for the $(k+1)$th scheduled operation can be started at $T_k + S_k$ i.e., $T_{k+1} = T_k + S_k$. Thus, $T_{i+1} = T_i + S_i$ for $\forall i = 1, 2, \ldots, 2 \times N$ when the scheduled sequence is alternating the setup of two operations of a job on the two machines. □

B. Proof of Case 3 for Proposition 5.

Proof: Because the number of operations on machine 1 is equal to number of operations on machine 2, and before the kth schedule job, all the two operations of jobs are already done, then there must be a successive at least two setup on machine 2. Again we assume the number of continuous setup operations on a machine as 2 and the first such operations are denoted as O_j and O_{j+1}. Then the operations before O_j and after O_{j+1} are on machine 1, which is denoted as O_h and O_{j+2} separately. $T(O_{j+1}) = T(O_j) + s2 + p.E(O_h) \leq T(O_j) + p$ because $T(O_h) + s1 \leq T(O_j)$. $T(O_{j+2}) = T(O_{j+1}) + s2$. And the time between $E(O_h)$ and $T(O_{j+2})$ is idle on machine 1.

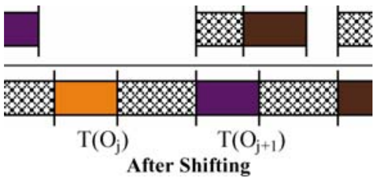


Before Shifting

Now we shift the Oh to the right till $T'(O_h) = T(O_j) + s2$. Then, $T'(O_h) + s1 = T(O_j) + s2 + s1 = T(O_{j+1}) - p - s2 + s2 + s1 = T(O_{j+1}) - p + s1$. Because s1 $\leq p$, so $T'(O_h) + s1 \leq T(O_{j+1})$. Also, $E'(O_h) = T'(O_h) + s1 + p = T(O_{j+1}) - p + s1 + p =$

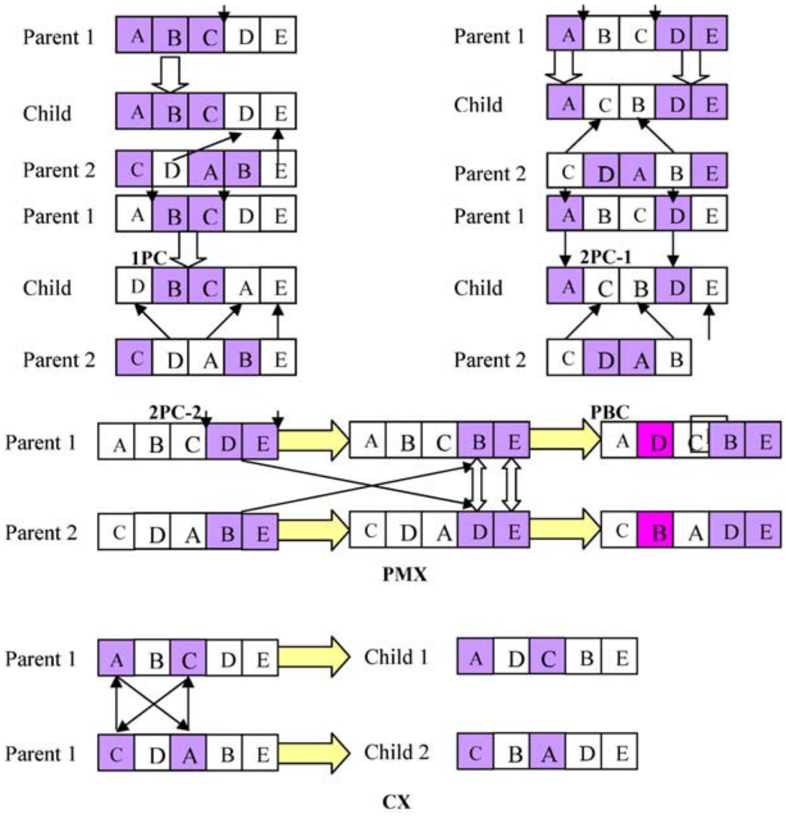
$T(O_{j+1}) + s1 \leq T(O_{j+1}) + s2 = T(O_{j+2})$. Thus, this shifting is valid.

$T'(O_h)\, E'(O_h) T(O_{j+2})$



If O_h is the first operation of O_j, then swap O_j and O_{j+1}. This swapping is easy because they have the same length of setup times and processing times. Continue this shifting and swapping until all setups are done alternatively on the two machines. As each movement does not the last operation on machine two to the right, so the $C_{\max}$ of the new sequence is less than or equal to $C_{\max}$ of s and since s is optimal, the new sequence is also optimal. □

C. Crossover Operators



PMX: Choose an interval of consecutive positions from each of the two parents, determine mappings in the two intervals and then swap the intervals; exchange the mapping elements outside the intervals; if infeasible, do forwarding mappings.

CX: Start from the first element from a parent, find its corresponding position in the other, and then find the corresponding position of the element at this new position. Repeat until a cycle is formed. Inherit those positions in the cycle and copy the rest from the other parent.

1PC: One point is randomly selected for dividing one parent. The set of elements on one side (randomly selected) is inherited from one parent, and the others are placed in the order of their appearance in the other parent.

2PC-1: Two points are randomly selected for dividing one parent. The elements outside the two points are inherited, and the others are placed in the same manner as 1PC.

2PC-2: The set of elements between the two randomly selected points is inherited.

2PC-3: Mixture of 2PC-1 and 2PC-2, each with probability 0.5.

PBC-1: The elements at randomly selected positions are inherited from one parent, and the others are placed in the order of their appearance in the other parent. Number of inherited positions is first chosen and then the positions are selected.

PBC-2: Same as PBC-1 but each position has probability 0.5 to be chosen as a inherit position.

References

Abdekhodaee, A. H. and A. Wirth, "Scheduling parallel machines with a single server: Some solvable cases and heuristics," *Computers and Operations Research*, **29**(3), 295–315 (2002).

Ben-Daya, M. and M. Al-Fawzan, "A tabu search approach for the flow-shop scheduling problem," *European Journal of Operational Research*, **109**(l), 88–95 (1998).

Brucker, P., S. Knust, and G. Wang, "Complexity results for flow-shop problems with a single server," *OSM Reihe P*, Heft 237, (2001).

Chen, C. L., V. S. Vempati, and N. Aljaber, "An application of genetic algorithms for flow-shop problems," *European Journal of Operational Research*, **80**(2), 389–396 (1995).

Cheng, T. C. E., G. Wang, and C. Sriskandarajah, "One-operator two-machine flow-shop scheduling with setup and dismounting times," *Computers and Operations Research*, **26**(7), 715–730 (1999).

Garey, M. R., D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of Operations Research*, **1**(2), 117–128 (1974).

Glass, C. A., Y. M. Shafransky, and V. A. Strusevich, "Scheduling for parallel dedicated machines with a single server," *Naval Research Logistics*, **47**, 304–328 (2000).

Glover, F., "Tabu Search-Part I," *ORSA Journal on Computing*, **1**(3), 190–206 (1989).

Glover, F., "Tabu Search-Part II," *ORSA Journal on Computing*, **2**(1), 4–32 (1990).

Goldberg, D. E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1989.

Gupta, J. N. D., "Two-stage, hybrid flowshop scheduling problem," *Journal of Operational Research Society*, **39**, 359–364 (1988).

Gupta, J. N. D. and W. P. Darrow, "The two-machine sequence dependent flowshop scheduling problem," *European Journal of Operational Research*, 439–446 (1986).

Holland, J., *Adaptation in Nature and Artificial Systems*, University of Michigan Press, Ann Arbor, MI, USA, 1975.

Ishibuchi, H., S. Misaki, and H. Tanaka, "Modified simulated annealing algorithms for the flow-shop sequencing problem," *European Journal of Operational Research*, **81**(2), 388–398, (1995).

Johnson, S. M., "Optimal two- and three-stage production schedules with set-up times included," *Naval Research Logistics Quarterly*, **1**(l), 61–68 (1954).

Krajeswski, L. J. and L. P. Ritzman, *Operations Management: Strategy and Analysis*, Reading, MA: Addison-Wesley, 1987.

Laarhoven, P. J. M. and E. H. L. Aarts, *Aarts, Simulated Annealing: Theory and Applications*, Kluwer Academic Publishers, Norwell, MA, 1987.

Narasimhan, S. L. and S. S. Panwalkar, "Scheduling in a two-stage manufacturing process," *International Journal of Production Research*, **22**, 555–564 (1984).

Murata, T., H. Ishibuchi, and H. Tanaka, "Genetic algorithms for flowshop scheduling problems," *Computer and Industrial Engineering*, **30**(4), 1061–1071 (1996).

Ogbu, F. A. and D. K. Smith, "The application of the simulated annealing algorithm to the solution of the n/m/$C_{\max}$ flowshop problem," *Computers and Operations Research*, **17**(3), 243–253 (1990a).

Ogbu, F. A. and D. K. Smith, "Simulated annealing for the permutation flowshop problem," *OMEGA*, **19**(1), 64–67 (1990b).

Oliver, I., D. Smith, and J. Holland, "Simulated annealing for the permutation flowshop scheduling," *OMEGA*, **19**, 64–67 (1987).

Osman, I. H. and C. N. Potts., "Simulated annealing for permutation flow-shop scheduling," *OMEGA*, **17**(6), 551-557 (1989).

Resende, M. G. C. and C. C. Ribeiro, "Greedy randomized adaptive search procedures," *AT&T Labs Research Technical Report* TD-53RSJY, version, **2** (2002).

Taillard, E. D., "Some efficient heuristic methods for the flow shop sequencing problem," *European Journal of Operational Research*, **47**(1), 65–74 (1990).

Taillard, E. D., "Parallel taboo search techniques for the job shop scheduling problem," *ORSA Journal on Computing* **6**(2), 108–117 (1994).

Widmer, M. and A. Hertz, "A new heuristic method for the flow-shop sequencing problem," *European Journal of Operational Research*, **41**(2), 186–193 (1989).

Yamada, T. and C. R. Reeves, "Permutation flowshop scheduling by genetic local search," *Proceedings of Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA' 97)*," *Institution of Electrical Engineers* London, 232–238 (1997).

Yamada, T. and C. R. Reeves, "Genetic algorithms, path relinking and the flowshop sequencing problem," *Evolutionary Computation journal* (MIT press), **6**(1), 230–234 (1998b).

Yoshida, T. and K. Hitomi, "Optimal two-stage production scheduling with setup times separated," *AIIE Transactions*, **11**, 261–263 (1979).

Zegordi, S. H., K. Itoh, and T. Enkawa, "Minimizing makespan for flow-shop scheduling by combining simulated annealing with sequencing knowledge," *European Journal of Operational Research*, **85**(3), 515–531 (1995).