# CHAPTER 3

# GENETIC ALGORITHMS FOR FLOWSHOP SCHEDULING PROBLEMS

## 3.1 INTRODUCTION

Flowshop scheduling is one of the most well-known problems in the area of scheduling. Various approaches have been proposed since Johnson's work [56]. Because Johnson described an optimization algorithm for minimizing the makespan on *n*-job and two-machine flowshop scheduling problems [56], this scheduling criterion has been often used in various approaches. A lot of researchers have tried to constructing an optimization method for scheduling problems. It is difficult, however, to find the optimal solution of a flowshop scheduling problem involving many jobs and machines (*e.g.*, 100 jobs and 10 machines). Many approaches can be classified into two categories: optimization algorithms for the exact solution (for example, Ignall & Schrage [33] and Lomnicki [69]) and heuristic algorithms for near optimal solutions (for example, Dannenbring [8] and Nawaz *et al.*[84]). Because the CPU power of available computers was rapidly improved, several heuristic approaches based on iterative improvement procedures have been applied to the flowshop scheduling. Osman & Potts [90] proposed simulated annealing heuristics, Taillard [108] and Widmer & Hertz [119] proposed tabu search heuristics.

Recently many authors applied GAs (see, Davis [9], Goldberg [23], and Holland [27]) to combinatorial optimization problems such as traveling salesman problems (for example, Jog *et al.*[55], Starkweather *et al.*[101], and Ulder *et al.*[115]) and scheduling problems (for example, Fox & McMahon [17], Glass *et al.*[22], Ishibuchi *et al.*[49], Manderick & Spiessens [70] and Syswerda [106]). Some empirical studies [9,101,115] showed that the ability of GAs to find near optimal solutions was a bit inferior to other search algorithms.

In this chapter, we apply GAs to single-objective flowshop scheduling problems. We first explain flowshop scheduling. Next we examine several crossover operators and mutation

operators to construct genetic algorithms for flowshop scheduling. By computer simulations, we point out that the combination of high performance crossover and mutation operators does not always lead to a high performance genetic algorithm [79,80]. We compare the genetic algorithm constructed for flowshop scheduling with other search algorithms such as local search, simulated annealing [90], and tabu search [108,119]. It is shown that the genetic algorithm is a bit inferior to the other search algorithms [82] as shown by several researchers [9, 101,115]. Then, we examine two hybrid genetic algorithms to improve the performance of the genetic algorithm [82]. One is a genetic local search algorithm, and the other is a genetic simulated annealing algorithm. We also introduce some modifications of search mechanisms in these hybrid genetic algorithms [82]. While careful parameter specifications are required for constructing GAs with high performance, it is shown that we can construct the genetic local search algorithm without careful parameter specifications.

## 3.2 FLOWSHOP SCHDULING PROBLEMS

We first briefly describe *n*-job and *m*-machine flowshop scheduling problems. General assumptions of flowshop scheduling problems can be written as follows (for details, see Dudek *et al.*[10]): Jobs are to be processed on multiple stages sequentially. There is one machine at each stage. Machines are available continuously. A job is processed on one machine at a time without preemption, and a machine processes no more than one job at a time. In this thesis, we assume that *n* jobs are processed in the same order on *m* machines. This means that our flowshop scheduling is the *n*-job sequencing problem. Let the processing time and the completion time of job *j* on machine *i* be $t_P(i, j)$ and $t_C(i, j)$, respectively. The sequence of *n* jobs is denoted by an *n*-dimensional vector $\boldsymbol{x} = (x_1, x_2, ..., x_n)$, where $x_k$ represents the *k*-th processing job. The completion time of each job corresponding to the sequence $\boldsymbol{x}$ can be calculated as

$$t_C(1,x_1) = t_P(1, x_1), \tag{3.1}$$

$$t_C(i, x_1) = t_C(i-1, x_1) + t_P(i, x_1), \quad \text{for } i = 2, 3, ..., m, \tag{3.2}$$

$$t_C(1, x_k) = t_C(1, x_{k-1}) + t_P(1, x_k), \quad \text{for } k = 2, 3, ..., n, \tag{3.3}$$

$$t_C(i, x_k) = \max\{t_C(i-1, x_k), t_C(i, x_{k-1})\} + t_P(i, x_k), \quad \text{for } i = 2, 3, ..., m;\ k = 2, 3, ..., n. \tag{3.4}$$

Flowshop scheduling problems are to determine the sequence $\boldsymbol{x}$ of *n* jobs based on a specific criterion. One of the makespan, the total flowtime, the maximum tardiness, and the total tardiness is often used as a scheduling criterion. Each of these criteria can be defined as follows in flowshop scheduling:

Makespan: $f_1(\boldsymbol{x}) = t_C(m, x_n)$, (3.5)

Total flowtime: $f_2(\boldsymbol{x}) = \sum_{k=1}^{n} t_C(m, x_k)$, (3.6)

Maximum tardiness: $f_3(\boldsymbol{x}) = \max\{\max\{t_C(m, x_k) - d(x_k), 0\} \mid k = 1, 2, \ldots, n\}$, (3.7)

Total tardiness: $f_4(\boldsymbol{x}) = \sum_{k=1}^{n} \max\{t_C(m, x_k) - d(x_k), 0\}$, (3.8)

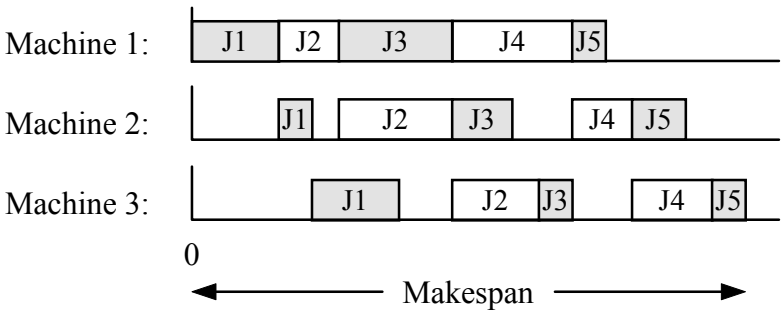


**Fig. 3.1** An example of a feasible solution of a five-job and three-machine flowshop scheduling problem and its makespan.

where $d(x_k)$ is the duedate of the *k*-th processing job $x_k$. We use the makespan as a criterion in this chapter because a lot of heuristic algorithms (*i.e.*, simulated annealing, tabu search, and so on) were proposed for flowshop scheduling with this criterion. The other criteria are considered in the following chapter about multi-objective flowshop scheduling. Fig. 3.1 shows an example of a feasible solution of a five-job and three-machine flowshop scheduling problem. The makespan is the completion time of the last job. That is, the completion time of Job 5 on Machine 3 is the makespan of this schedule. We will examine various genetic operators to construct GAs for flowshop scheduling problems where the makespan is employed as an objective.

## 3.3 GENETIC OPERATORS FOR PERMUTATION STRINGS

As we explained the difference between the binary coding and the permutation coding in Subsections 2.2.1, 2.2.4, and 2.2.5, we should carefully design GAs according to the type of optimization problems to be considered. The aim of flowshop scheduling problems with the objective of minimizing the makespan is to find the sequence $\boldsymbol{x}$ of $n$ jobs with the minimum makespan. A job sequence is directly treated as an individual in GAs. That is, the permutation coding method is appropriate for flowshop scheduling problems. We consider various genetic operators which fulfill the requirement of permutation problems that each element of a string should appear once in a generated string. We explain seven crossover operators and five mutation operators in the following subsections [78,79,80,82].

### *3.3.1 Crossover*

In this subsection, we examine seven crossover operators. The following two operators are often used for GAs with the permutation coding.

***(1) One-point order crossover***

This crossover was briefly explained as an example of a crossover operator for permutation strings in Subsection 2.2.4. This crossover is illustrated in Fig. 3.2. One point is randomly selected for dividing one parent string. The set of jobs on one side (*i.e.*, either the head part or the tail part of the string) is chosen with the same probability and inherited from the parent string to the offspring, and all the remaining jobs on the other side are placed in the order of their appearance in the other parent string.

***(2) Two-point order crossover***

This crossover is illustrated in Fig. 3.3. Two points are randomly selected for dividing one parent string. The jobs outside the selected two points (*i.e.*, the head part and the tail part) are always inherited from one parent string to the offspring, and the other jobs (*i.e.*, the mid part of the string) are placed in the same manner as the one-point order crossover.

The following versions of the position based order crossover (see, Syswerda [106]) were also examined in computer simulations:

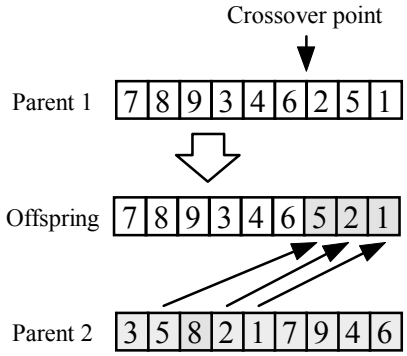


**Fig. 3.2** The one-point order crossover.

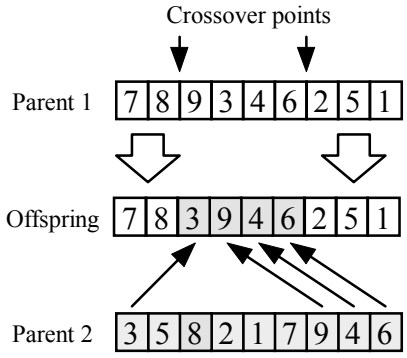


**Fig. 3.3** The two-point order crossover.

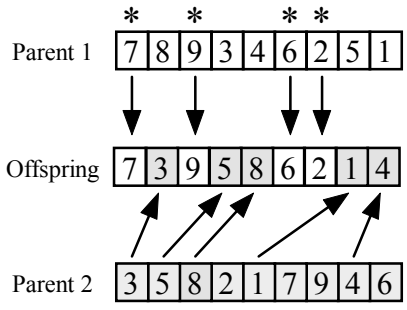


**Fig. 3.4** The position based order crossover.

***(3) Position based order crossover***

This crossover is illustrated in Fig. 3.4. The jobs at randomly selected positions marked by " * " are inherited from one parent string to the offspring, and the other remaining jobs are placed in the order of their appearance in the other parent string.

The following four crossover operators, which had been mainly proposed for traveling salesman problems, are also examined in this chapter for flowshop scheduling problems. We omit the detailed explanations of these crossovers because they turn out to be inappropriate for the flowshop scheduling by the following computer simulations.

***(4) Partially matched crossover in Goldberg [23]***

***(5) Cycle crossover in Oliver et al.[89]***

***(6) Edge recombination crossover in Whitley et al.[118]***

***(7) Enhanced Edge recombination crossover in Starkweather et al.[101]***

### *3.3.2 Mutation*

We examine the following five mutation operators for flowshop scheduling problems.

***(1) Adjacent two-element change***

Adjacent two elements are exchanged as shown in Fig. 3.5. The adjacent two elements to be exchanged are randomly selected.

***(2) Arbitrary two-element change***

This mutation was explained as an example of the mutation operator for permutation strings in Subsection 2.2.5. Arbitrary selected two jobs are changed as shown in Fig. 3.6. The two elements to be changed are arbitrary and randomly selected. This mutation includes the adjacent two-element change as a special case.

***(3) Arbitrary three-element change***

Arbitrary selected three elements are arbitrary changed as shown in Fig. 3.7. The three elements to be changed are arbitrary and randomly selected, and the order of the selected elements after the mutation is randomly specified from the five candidate transitions. This mutation includes the above two mutations as a special case.

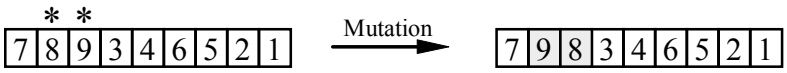


**Fig. 3.5** Adjacent two-element change.

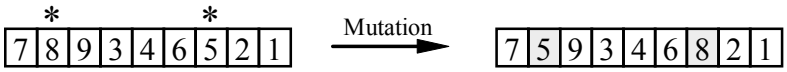


**Fig. 3.6** Arbitrary two-element change.

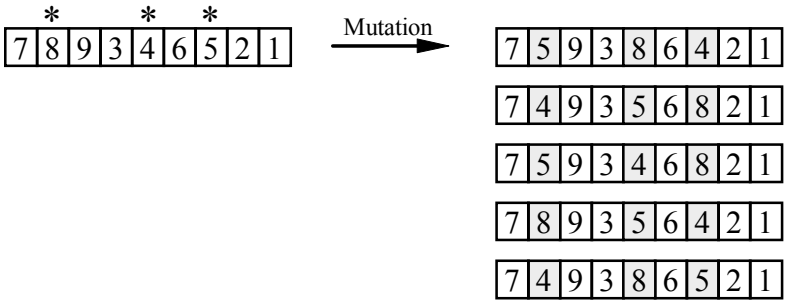


**Fig. 3.7** Arbitrary three-element change.

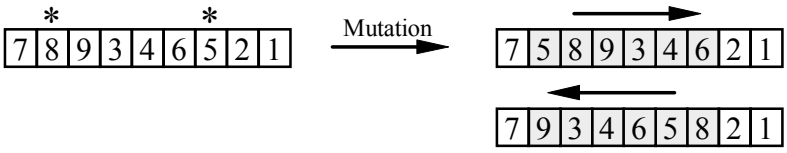


**Fig. 3.8** Shift change.

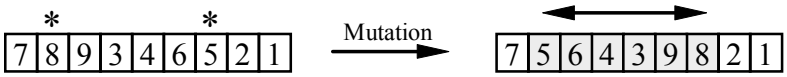


**Fig. 3.9** Inversion.

***(4) Shift change***

In this mutation, an element at one position is removed and put at another position as shown in Fig. 3.8. The two positions are randomly selected. This mutation includes the adjacent two-element change as a special case and has an intersection with the arbitrary three-element change.

***(5) Inversion***

Inversion mutation reverses the order of the elements between randomly selected two positions as shown in Fig. 3.9. This mutation includes the adjacent two-element change as a special case and has an intersection with the arbitrary two-element change and the arbitrary three-element chage.

As we can see from Fig. 3.5 ~ Fig. 3.9, these mutation operators are applied to an entire string while a mutation operator is usually applied to each position of a string in the case of binary coding. Thus it should be noted that the mutation probability $P_{\mathrm{m}}$ is defined for each string in the case of permutation coding. Since mutation operator can be viewed as a transition from the current solution to its neighborhood solution in local search algorithms, then mutation operators may improve individuals locally.

## 3.4 POSITIVE AND NEGATIVE COMBINATION EFFECTS OF GENETIC OPERATORS

Various crossover and mutation operators have been examined for sequencing problems. Because the performance of GAs strongly depends on the choice of such operators, the selection of appropriate operators is critical for constructing high performance GAs. The evaluation of each genetic operator is usually performed by computer simulations using a specially constructed genetic algorithm. That is, when the performance of a crossover operator is evaluated, a genetic algorithm without mutation is employed. The evaluation of a mutation operator is performed by a genetic algorithm without crossover. For example, various crossover and mutation operators were examined in this manner by Manderick & Spiessens [70].

In this section, we point out that the combination of high performance crossover and mutation operators does not always lead to a high performance genetic algorithm. We clearly illustrate this fact by computer simulations on flowshop scheduling problems. First we evaluate each of various genetic operators in order to select high performance crossover and mutation operators. Next we combine the selected crossover and mutation operators into a genetic algorithm, and evaluate its performance. While the performance of each operator is good when it is independently evaluated, the genetic algorithm constructed by these two operators does not always work well. This negative combination effect is due to the complementary nature of crossover and mutation operators in genetic algorithms. Finally we explain how crossover and mutation operators can be appropriately specified to construct a high performance genetic algorithm by utilizing the positive combination effect of these two operators.

### *3.4.1 Flowshop scheduling problems*

In this chapter, we simply transform the makespan $f_1(\boldsymbol{x})$ in (3.5) to the fitness function $f(\boldsymbol{x})$ as follows:

$$f(\boldsymbol{x}) = -f_1(\boldsymbol{x}) = -t_C(m, x_n). \tag{3.9}$$

We use the following simple genetic algorithm that has often been employed for evaluating genetic operators in the literature:

*Step 0 (Initialization)*: Let $t := 1$ where $t$ is the index of generation. Randomly generate an initial population $\Psi_t$ including $N_{\text{pop}}$ solutions where $N_{\text{pop}}$ is the population size.

*Step 1 (Evaluation)*: Calculate the makespan of each schedule in the current population $\Psi_t$.

*Step 2 (Selection)*: Select $N_{\text{pop}}$ pairs of solutions from the current population $\Psi_t$ according to the selection probability $P_{\text{s}}(\boldsymbol{x})$ based on the linear scaling [23]:

$$P_{\text{s}}(\boldsymbol{x}) = \frac{f(\boldsymbol{x}) - f_{\min}(\Psi_t)}{\sum_{\boldsymbol{x}' \in \Psi_t} \{f(\boldsymbol{x}') - f_{\min}(\Psi_t)\}}, \tag{3.10}$$

where $f_{\min}(\Psi_t)$ is the minimum fitness value (*i.e.*, the worst fitness value) in the current population $\Psi_t$.

*Step 3 (Crossover)*: Apply a crossover operator to each of the pairs selected in Step 2 to generate $N_{\text{pop}}$ solutions with the crossover probability $P_{\text{c}}$. When the crossover operator is not applied, one of the parents is handled as an offspring.

*Step 4 (Mutation)*: Apply a mutation operator to each of the generated $N_{\text{pop}}$ solutions in Step 3 with the mutation probability $P_{\text{m}}$.

*Step 5 (Termination test)*: If a prespecified stopping condition is satisfied, stop the algorithm. Otherwise, update $t$ as $t := t + 1$ and return to Step 1. In computer simulations, the total number of generations is used as the stopping condition.

It is noted that the elitist strategy described in Subsection 2.2.6 was not employed in this algorithm. We specified the population size $N_{\text{pop}}$ as $N_{\text{pop}} = 100$, and the stopping condition $t_{\max}$ as $t_{\max} = 500$ in computer simulations.

As a test problem, we randomly generated a flowshop scheduling problem with 20 jobs and 10 machines. The processing time of each job at each machine was randomly specified as an integer in the closed interval [1, 99]. We applied the genetic algorithm to this test problem 30 times.

### 3.4.2 Performance measure

In computer simulations, we used the following performance measure in order to evaluate each genetic operator for our scheduling problem:

$$Performance = f(\boldsymbol{x}^*) - f(\boldsymbol{x}^*_{\text{initial}}), \tag{3.11}$$

where $\boldsymbol{x}^*_{\text{initial}}$ is the best solution in the initial population and $\boldsymbol{x}^*$ is the best solution among all the populations. That is, $f(\boldsymbol{x}^*_{\text{initial}})$ is the fitness value corresponding to the minimum makespan in the initial population and $f(\boldsymbol{x}^*)$ corresponds to the minimum makespan during the execution of the genetic algorithm. Thus the performance measure in (3.11) can be viewed as the total improvement of the makespan during the execution of the genetic algorithm.

### 3.4.3 Effectiveness of genetic operators

Manderick & Spiessens [70] showed that statistical measures related to the fitness landscape can be used to tune and to optimize the performance of the genetic algorithm. They used correlation coefficients of genetic operators as a statistical measure to select the best operators. The correlation coefficient $\rho_{OP}$ of a genetic operator *OP* is obtained as follows: The operator *OP* in the case of crossover is applied to two individuals and it generates offspring. For each application of the operator *OP*, the values $f_p$ and $f_c$ represent the mean fitness of the two parents and the fitness of their offspring. In the case of mutation, the operator *OP* is applied to one individual and it generates offspring. The values $f_p$ and $f_c$ represent the fitness of the one parent and its offspring. To calculate the correlation coefficient $\rho_{OP}$ of the operator *OP*, we take a number of parent strings, apply the operator *OP* to get their offspring, and calculate the correlation coefficient between the fitness values $f_p$ and $f_c$:

$$\rho_{OP}(f_p, f_c) = \frac{Cov(f_p, f_c)}{\sigma(f_p)\sigma(f_c)}, \tag{3.12}$$

where $Cov(f_p, f_c)$ is the covariance between the values of $f_p$ and $f_c$, and $\sigma(f_p)$ and

$\sigma(f_c)$ are the standard deviation of the values of $f_p$ and $f_c$, respectively. Manderick & Spiessens [70] showed that correlation coefficients of genetic operators are related with their performance by computer simulations. That is, they said that the effectiveness of genetic operators can be expected by calculating correlation coefficients of genetic operators. It should be noted that either a crossover operator or a mutation operator is used in their genetic algorithms. That is, they used genetic algorithms with no mutation operator in order to measure the performance of crossover operators, and genetic algorithms with no crossover operator in order to measure the performance of mutation operators.

We will examine seven crossover operators and five mutation operators described in Section 3.3 in the following subsections. First we examine the performance of genetic operators in the way proposed by Manderick & Spiessens [70]. Then we consider the positive and negative combination effects of crossover and mutation operators.

### *3.4.4 Examination of crossover operators*

In order to examine the seven crossover operators in Subsection 3.3.1, we applied the genetic algorithm in Subsection 3.4.1 to our test problem 30 times by specifying the mutation probability $P_{\mathrm{m}}$ as $P_{\mathrm{m}} = 0$ in the same manner as in Manderick & Spiessens [70]. This means that no mutation operator was used in the genetic algorithm when the crossover operators were examined. Each crossover operator was examined using ten crossover probabilities: $P_{\mathrm{c}} = 0.1,\ 0.2,\ ...,\ 1.0$. These values were also used in Manderick & Spiessens [70].

We examined each crossover operator by 30 runs of the genetic algorithm. The average value of the performance measure in (3.11) over the 30 runs was calculated for each crossover operator with each crossover probability. The best crossover probability was determined for each crossover operator by these computer simulations (*e.g.*, $P_{\mathrm{c}} = 0.9$ for the one-point order crossover). In Table 3.1, we show the crossover probability determined in this manner for each crossover operator. We also show in Table 3.1 the average value of the performance measure obtained by each crossover operator with the best specification of the crossover probability. From Table 3.1, we can see that the best performance was obtained by the position based order crossover with the crossover probability $P_{\mathrm{c}} = 0.9$.

We also calculated the correlation coefficients in (3.12) for each crossover operator by applying a crossover operator to 30,000 pairs of parent strings which were randomly generated.

**Table 3.1** Average value of the performance measure over 30 runs for each crossover operator with its best crossover probability.

| Crossover | $P_c$ | Performance |
|---|---|---|
| One-point order crossover | 0.9 | 176.80 |
| Two-point order crossover | 0.9 | 144.33 |
| **Position based order crossover** | **0.9** | **206.53*** |
| Partially matched crossover | 1.0 | 180.93 |
| Cycle crossover | 1.0 | 152.73 |
| Edge recombination | 0.5 | 124.70 |
| Enhanced edge recombination | 0.5 | 125.56 |

* Best result

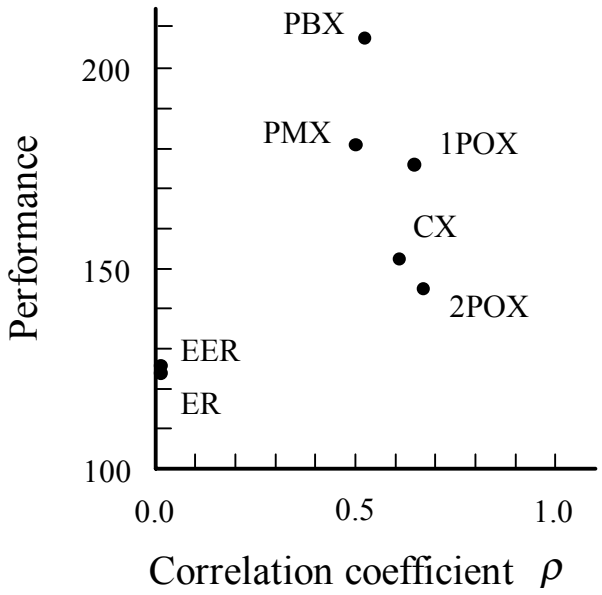


**Fig. 3.10** Relation between the performance and the correlation coefficients of crossover operators.

Fig. 3.10 shows the relation between the performance and the correlation coefficients of the seven crossover operators where 1POX is the one-point order crossover, 2POX is the two-point order crossover, PBX is the position based order crossover, PMX is the partially matched crossover, CX is the cycle crossover, ER is the edge recombination, and EER is the enhanced edge recombination. From Fig. 3.10, we can observe the correlation coefficient of the crossover operator may be related with its performance.

### *3.4.5 Examination of mutation operators*

In order to examine the five mutation operators in Subsection 3.3.2, we applied the genetic algorithm 30 times to our test problem by specifying the crossover probability $P_c$ as $P_c = 0$ in the same manner as in Manderick & Spiessens [70]. This means that no crossover operator was used in the genetic algorithm when the mutation operators were examined. Each mutation operator was examined using ten mutation probabilities: $P_m = 0.1, 0.2, ..., 1.0$. These values were also used in Manderick & Spiessens [70].

We examined each mutation operator in the same manner as in the last subsection. Simulation results are shown in Table 3.2. From Table 3.2, we can see that the best performance was obtained by the shift change mutation with the mutation probability $P_m = 0.4$.

We also calculated the correlation coefficient in (3.12) for each mutation operator by applying a mutation operator to 30,000 pairs of parent strings which were randomly generated. Fig. 3.11 shows the relation between the performance and the correlation coefficients of the five mutation operators. From Fig. 3.11, we can observe the correlation coefficient of the mutation operator is related with its performance.

From Table 3.1 and Table 3.2, the combination of the position based order crossover with $P_c = 0.9$ and the shift change mutation with $P_m = 0.4$ seems to lead to a high performance genetic algorithm. The performance of this combination is examined in the next subsection.

**Table 3.2** Average value of the performance measure over 30 runs for each mutation operator with its best mutation probability.

| Mutation | $P_m$ | Performance |
|---|---|---|
| Adjacent two-element change | 1.0 | 195.20 |
| Arbitrary two-element change | 0.2 | 197.60 |
| Arbitrary three-element change | 0.3 | 192.43 |
| **Shift change** | **0.4** | **211.13*** |
| Inversion | 0.3 | 181.60 |

***** Best result

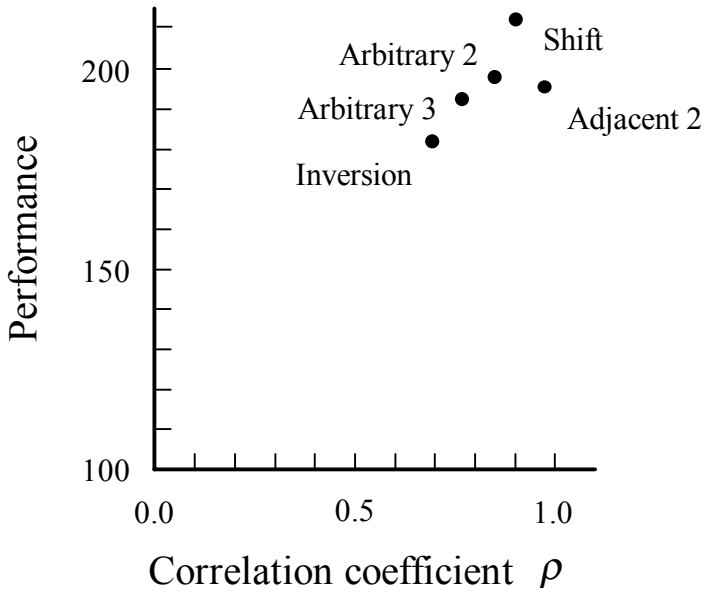


**Fig. 3.11** Relation between the performance and the correlation coefficient of mutation operators.

### *3.4.6 Combination of genetic operators*

Based on the simulation results in Table 3.1 and Table 3.2, we constructed a genetic algorithm using the position based order crossover with $P_c = 0.9$ and the shift change mutation with $P_m = 0.4$. This genetic algorithm was applied to our test problem 30 times with the same population size and the same stopping condition as in Subsections 3.4.4 and 3.4.5. By this computer simulation, we had the following result:

**Average performance: 198.2.**

From the comparison of this value with Table 3.1 and Table 3.2, we can see that the performance of the genetic algorithm was deteriorated by the combination of the high performance crossover and mutation operators.

This result suggests the existence of the negative combination effect of crossover and mutation operators. That is, crossover and mutation operators can not be appropriately specified by the independent examination of one operator from the other operator because of their complementary nature.

### *3.4.7 Selection of genetic operators and their probabilities*

In this subsection, we explain several methods for appropriately specifying crossover and mutation operators.

***A. Probability tuning***

One reason of the poor performance of the genetic algorithm in the last subsection may be the inappropriate specifications of the crossover and mutation probabilities. Those probabilities were specified by the computer simulations in Subsections 3.4.4 and 3.4.5 where either crossover or mutation operator (not both operators) was used in genetic algorithms. Thus those probabilities may be inappropriate when both operators are used in genetic algorithms. This means that the performance of the genetic algorithm in Subsection 3.4.6 may be improved by the tuning of those probabilities.

In order to appropriately specify those probabilities, we applied a genetic algorithm with the position based order crossover and the shift change mutation to our test problem 30 times by using the following crossover and mutation probabilities:

$$P_{\mathrm{c}} = 0.1, 0.2, 0.3, \ldots, 1.0\,, \quad P_{\mathrm{m}} = 0.1, 0.2, 0.3, \ldots, 1.0\,.$$

Therefore there were 100 combinations of the crossover and mutation probabilities.

The following best result was obtained by $P_{\mathrm{c}} = 0.3$ and $P_{\mathrm{m}} = 0.2$:

**Average performance: 216.8.**

From the comparison of this result with the value "198.2" in Subsection 3.4.6, we can see that the performance of the genetic algorithm was improved by tuning the crossover and mutation probabilities. The above result is also better than all the results in Table 3.1 and Table 3.2 where either crossover or mutation operator (not both operators) was employed in genetic algorithms. This means that the positive combination effect can be realized by appropriately specifying the crossover and mutation probabilities.

***B. Search for good genetic operators***

In the last subsection, we used the crossover and mutation operators that were independently selected by the computer simulations in Subsections 3.4.4 and 3.4.5. Here we examine a search

method of appropriate genetic operators by considering the combination effects of crossover and mutation. There are two versions of the search method: One starts from the selection of a crossover operator, and the other starts from the selection of a mutation operator.

The first version of the search method can be written as follows:

*Step 1 (Initial selection of a crossover operator)*: Evaluate applicable crossover operators with various crossover probabilities, and choose the best crossover operator and the best crossover probability as in Table 3.1. In this step, the evaluation is performed by a genetic algorithm with no mutation operator.

*Step 2 (Initial selection of a mutation operator)*: Evaluate applicable mutation operators with various mutation probabilities by a genetic algorithm with the crossover operator and the crossover probability specified in Step 1. That is, the evaluation is performed by a genetic algorithm with both mutation and crossover. Choose the best mutation operator and the best mutation probability.

*Step 3 (Search for a better crossover operator)*: Evaluate applicable crossover operators with various crossover probabilities by a genetic algorithm with the mutation operator and the mutation probability specified in the previous step (*i.e.*, Step 2 or Step 4). Choose the best crossover operator and the best crossover probability. If the performance of the genetic algorithm is not improved, stop the algorithm. Otherwise go to Step 4 with the selected crossover operator and crossover probability.

*Step 4 (Search for a better mutation operator)*: Evaluate applicable mutation operators with various mutation probabilities by a genetic algorithm with the crossover operator and the crossover probability specified in Step 3. Choose the best mutation operator and the best mutation probability. If the performance of the genetic algorithm is not improved, stop the algorithm. Otherwise return to Step 3 with the selected mutation operator and mutation probability.

We applied this algorithm to our test problem. In each step, we selected a genetic operator and its probability by average results over 30 runs. In Step 1, the seven crossover operators were examined by a genetic algorithm with no mutation operator. As shown in Table 3.1, the position based order crossover with $P_c = 0.9$ was selected, and the average performance measure "206.53" was obtained in this step. In Step 2, the five mutation operators were

examined by a genetic algorithm with the position based order crossover. This means that the combinations of the position based order crossover and the five mutation operators were examined. In this step, the adjacent two-element change mutation with $P_m = 0.6$ was selected, and the average performance measure "224.3" was obtained. In Step 3, the combinations of the adjacent two-element change mutation and the seven crossover operators were examined. Then the position based order crossover with $P_c = 0.6$ was selected, and the average performance measure "225.8" was obtained. Because the performance of the genetic algorithm was improved in Step 3 from 224.3 to 225.8, we went to Step 4 where the combinations of the position based order crossover with $P_c = 0.6$ and the five mutation operators were examined. In Step 4, the adjacent two-element change mutation with $P_m = 0.6$ was selected, and the average performance measure "225.8" was obtained. Because the performance of the genetic algorithm was not improved in this step, the algorithm was terminated. Therefore the following final result was obtained by the position based order crossover with $P_c = 0.6$ and the adjacent two-element change mutation with $P_m = 0.6$:

**Average performance: 225.8.**

This result is better than any results in Table 3.1 and Table 3.2 where either crossover or mutation operator (not both operators) was employed in genetic algorithms. This means that the positive combination effect of crossover and mutation was realized by selecting the genetic operators by the above algorithm.

The other version of the proposed method starts from the selection of a mutation operator. This version was also applied to our test problem and the following result was obtained by the two-point order crossover with $P_c = 0.9$ and the shift change mutation with $P_m = 0.3$:

**Average performance: 222.3.**

***C. Examination of all combinations***

One of the most straightforward methods for selecting crossover and mutation operators is to examine all the possible combinations of various operators. Using our test problem, we examined the 35 combinations of the seven crossover operators and the five mutation operators. The crossover and mutation probabilities in Table 3.1 and Table 3.2 were used in computer

simulations. Simulation results were summarized in Table 3.3 where the average value of the performance measure over 30 runs is shown for each combination. From this table, we can see the following best result was obtained by the combination of the cycle crossover and the adjacent two-element change mutation:

**Average performance: 227.2.**

It should be noted that the cycle crossover and the adjacent two-element change mutation were not highly evaluated in Table 3.1 and Table 3.2 when each of these operators was independently examined. Therefore the good result "227.2" was attained by the positive combination effect of crossover and mutation in genetic algorithms.

The crossover and mutation probabilities were also tuned for this combination in the same manner as in Subsection 3.4.7.A. Then the following improved result was obtained by $P_c = 0.8$ and $P_m = 1.0$:

**Average performance: 228.3.**

**Table 3.3** The performance of all the possible combinations of various genetic operators.

| | Adjacent 2 ($P_m = 1.0$) | Arbitrary 2 ($P_m = 0.2$) | Arbitrary 3 ($P_m = 0.3$) | Shift ($P_m = 0.4$) | Inversion ($P_m = 0.3$) |
|---|---|---|---|---|---|
| 1POX ($P_c = 0.9$) | 211.5 | 211.8 | 210.3 | 213.7 | 200.8 |
| 2POX ($P_c = 0.9$) | 206.9 | 214.9 | 209.9 | 217.1 | 196.1 |
| PBX ($P_c = 0.9$) | 217.6 | 205.6 | 188.3 | **198.2** | 182.2 |
| PMX ($P_c = 1.0$) | 172.0 | 164.2 | 150.0 | 156.1 | 148.2 |
| CX ($P_c = 1.0$) | **227.2*** | 205.8 | 201.2 | 211.7 | 191.6 |
| ER ($P_c = 0.5$) | 126.2 | 118.7 | 115.0 | 120.3 | 123.4 |
| EER ($P_c = 0.5$) | 121.8 | 121.0 | 112.1 | 125.3 | 126.0 |

**198.2** is the result by the combination of the best operators in Table 3.1 and Table 3.2.

**227.2*** is the best result in this table.

## 3.5 COMPARISON WITH OTHER SEARCH ALGORITHMS

In this section, we compare the GA with other search algorithms such as local search, tabu search and simulated annealing. In computer simulations, the neighborhood structure based on the shift change mutation was used in all the search algorithms.

### *3.5.1 Other search algorithms*

***A. Local search algorithm***

In a local search algorithm, first an initial solution $\boldsymbol{x}$ is randomly generated. Then the solutions in the neighbor of the current solution $\boldsymbol{x}$ are examined in random order. When a better solution is found by this neighborhood examination, the current solution is immediately replaced with that solution. That is, the current solution is replaced with the first solution that improves the current one. When there is no solution that improves the current one in the neighborhood, the current solution $\boldsymbol{x}$ can be regarded as a local optimal solution. Then the search procedure restarts from another initial solution that is generated randomly again. This search procedure is iterated until a prespecified stopping condition is satisfied.

In the above algorithm, the current solution is immediately replaced with the first solution that improves the current one. There is another strategy for a transition from the current solution. That is, the current solution is replaced the best solution of all the neighborhood solutions of the current solution. The former strategy is called the first move strategy, and the latter is called the best move strategy. In this section, we employed the first move strategy.

***B. Tabu search algorithm***

In computer simulations, we used the following tabu search algorithm, which is basically the same as an algorithm mentioned in Taillard [108]. First an initial solution $\boldsymbol{x}$ is randomly generated, and the tabu list is specified as $\phi$ where $\phi$ shows that the tabu list is empty. Next the neighborhood solutions that are not included in the tabu list are examined in random order. Let $\boldsymbol{y}^*$ be the first solution that improves the current one. If no solution improves the current one, then let $\boldsymbol{y}^*$ be the best solution in the neighborhood solutions that are not included in the tabu list. If a prespecified stopping condition is satisfied, then stop the search procedure. Otherwise, let $\boldsymbol{x} := \boldsymbol{y}^*$, renew the tabu list, and continue the search procedure from the updated current solution $\boldsymbol{x}$. In computer simulations, we used the tabu list defined by the pairs of

positions and jobs. When the job $x_k$ at the $k$-th position is removed and put at another position, the pair $(k, x_k)$ is added to the tabu list. The length of the tabu list was specified as seven.

As in the case of local search algorithm, the best move strategy can be utilized in this algorithm. The comparison of the first move and the best move was examined in Taillard [108]. In his paper, the first move strategy was better than the best move strategy. Therefore we employed the first move strategy in this algorithm.

***C. Simulated annealing algorithm***

In this section, we used the simulated annealing algorithm proposed by Osman & Potts [90]. First an initial solution $\boldsymbol{x}$ is randomly generated. The transition from the current solution $\boldsymbol{x}$ to a neighborhood solution $\boldsymbol{y}$ is accepted by the following probability:

$$P_{\text{trans}}(\boldsymbol{x} \to \boldsymbol{y}) = \min\{1,\ \exp(-\frac{f(\boldsymbol{x}) - f(\boldsymbol{y})}{c_l})\},\ \ l = 1, 2, ..., N_{\text{iteration}}, \quad (3.13)$$

where $f(\cdot)$ is a fitness function in (3.9) to be maximized, $c_l$ is a control parameter called temperature, and $N_{\text{iteration}}$ is the total iteration number of simulated annealing algorithms. In simulated annealing algorithms, the value of the control parameter is gradually decreased from a large initial value to a small final value. In computer simulations, we specified the sequence of $c_l$ as

$$c_{l+1} = c_l / (1 + \beta \cdot c_l),\ \ l = 1, 2, ..., N_{\text{iteration}} - 1, \quad (3.14)$$

where $\beta$ is a positive constant. We determined the positive constant $\beta$ and the initial value of $c_l$ according to Osman & Potts [90]. The current solution $\boldsymbol{x}$ is replaced with the candidate solution $\boldsymbol{y}$ with the acceptance probability in (3.13). This search procedure is iterated until a prespecified stopping condition is satisfied.

## *3.5.2 Simulation results*

We applied the genetic algorithm (GA), the local search algorithm (LS), the tabu search algorithm (TS) and the simulated annealing algorithm (SA) to randomly generated 100 test

problems with 20 jobs and 10 machines. We employed the two-point order crossover and the shift change mutation as genetic operators in the genetic algorithm. For comparison, we also applied a random sampling technique with the same computation load in the other algorithms. We also applied these algorithms to randomly generated 100 test problems with 50 jobs and 10 machines.

Simulation results are shown in Table 3.4 and Table 3.5 for 20-job problems and 50-job problems, respectively. We plotted the results for 20-job problems in Fig. 3.12. In these tables, the average makespan to be minimized obtained by each algorithm are normalized using the result by the simulated annealing algorithm with 200,000 evaluations (which were the second best result in Table 3.4, and the best result in Table 3.5). From these tables and figure, we can see that the genetic algorithm, which is much superior to the random sampling technique, is a bit inferior to the other search algorithms.

**Table 3.4** Comparison of the search algorithms for 20-job and 10-machine problems.

| Evaluations | 10,000 | 50,000 | 200,000 |
|---|---|---|---|
| GA | 101.5 | 101.0 | 100.7 |
| LS | 101.2 | 100.5 | 100.2 |
| TS | 101.1* | 100.5* | 100.0** |
| SA | 100.9** | 100.2** | 100.0* |
| Random | 109.6 | 108.4 | 107.5 |

** and * show the best result and the second best result in each column, respectively.

**Table 3.5** Comparison of the search algorithms for 50-job and 10-machine problems.

| Evaluations | 10,000 | 50,000 | 200,000 |
|---|---|---|---|
| GA | 102.3 | 101.4 | 101.1 |
| LS | 101.9 | 101.2 | 100.7 |
| TS | 101.4* | 101.0* | 100.5* |
| SA | 101.2** | 100.4** | 100.0** |
| Random | 111.4 | 110.5 | 109.8 |

** and * show the best result and the second best result in each column, respectively.

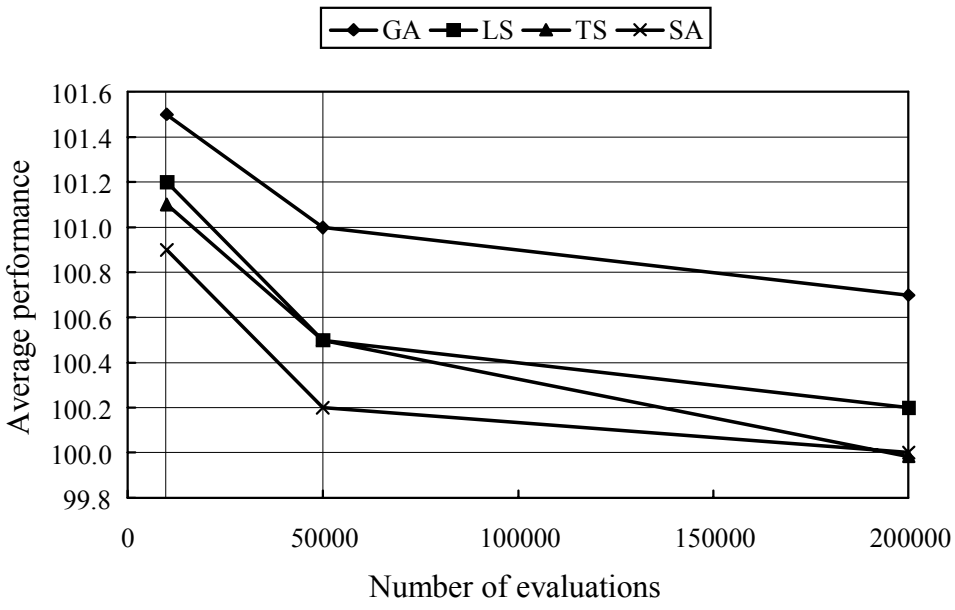


**Fig. 3.12** Simulation results by the four search algorithms for 20-job and 10-machine problems.

## 3.6 HYBRIDIZATION OF GENETIC ALGORITHMS WITH OTHER SEARCH ALGORITHMS

In this section, we examine two hybrid algorithms (*i.e.*, genetic local search and genetic simulated annealing) to improve the performance of the genetic algorithm which was a bit inferior to the other search algorithms in the previous section. While careful parameter specifications are required for constructing GAs with high performance, it is shown that we can construct the genetic local search algorithm without careful parameter specifications.

### *3.6.1 Genetic local search*

Genetic local search have been proposed by several authors for mainly traveling salesman problems (for example, see Glass *et al.*[22], Jog *et al.*[55], and Ulder *et al.*[115]). Generally a local search procedure can be written as follows:

**[Local Search Procedure]**

*Step 0*: Specify an initial solution $\boldsymbol{x}$.

*Step 1*: Examine a neighborhood solution $\boldsymbol{y}$ of the current solution $\boldsymbol{x}$.

*Step 2*: If $\boldsymbol{y}$ is a better solution than $\boldsymbol{x}$, replace the current solution $\boldsymbol{x}$ with $\boldsymbol{y}$ (*i.e.*, let $\boldsymbol{x} := \boldsymbol{y}$) and return to Step 1.

*Step 3*: If all the neighborhood solutions of the current solution $\boldsymbol{x}$ have been already examined (*i.e.*, if there is no neighborhood solution that improves $\boldsymbol{x}$), then end this procedure. Otherwise return to Step 1 (*i.e.*, another neighborhood solution is examined in Step 1).

As we can see from Step 3, this local search procedure is terminated when there is no better solution in the neighborhood of the current solution $\boldsymbol{x}$. This means that all the neighborhood solutions of the current solution $\boldsymbol{x}$ should be examined before the procedure is terminated. Therefore the total number of solutions examined by this local search procedure for a single initial solution is more than or equal to the number of the neighborhood solutions. For example, if we define the neighborhood solutions by exchanging arbitrary two elements for a flowshop scheduling problem with 20 jobs, the number of the neighborhood solutions is ${}_{20}C_2 = 190$. This means that at least 190 solutions are examined before the local search procedure is

terminated for a single initial solution. When we applied a genetic local search algorithm where the above mentioned local search algorithm was incorporated into a genetic algorithm to 100 test problems with 20 jobs, the average number of generations was 7.4 (in the case of $N_{\text{pop}} = 10$ and 50,000 evaluations). If we want to efficiently utilize the global search ability of GAs in the genetic local search algorithm, we have to reduce the computation time spent by the local search. This can be realized by restricting the number of neighborhood solutions examined by the local search procedure. We modify the local search procedure as follows:

**[Modified Local Search Procedure]**

*Step 0*: Specify an initial solution $\boldsymbol{x}$.

*Step 1*: Examine a neighborhood solution $\boldsymbol{y}$ of the current solution $\boldsymbol{x}$.

*Step 2*: If $\boldsymbol{y}$ is a better solution than $\boldsymbol{x}$, replace the current solution $\boldsymbol{x}$ with $\boldsymbol{y}$ (*i.e.*, let $\boldsymbol{x} := \boldsymbol{y}$) and return to Step 1.

*Step 3*: If a certain number of neighborhood solutions of the current solution $\boldsymbol{x}$ have been already examined (*i.e.*, if there is no better solution among the a certain number of neighborhood solutions of the current solution $\boldsymbol{x}$), then end this procedure. Otherwise return to Step 1 (*i.e.*, another neighborhood solution is examined in Step 1).

This algorithm is terminated if no better solution is found among a certain number of neighborhood solutions that are randomly selected from the neighborhood of the current solution. Therefore if we examine a small number of neighborhood solutions, the local search procedure may be terminated soon. On the contrary, if we examine a large number of neighborhood solutions, the local search procedure examines many solutions. In this manner, we can adjust the computation time spent by the local search procedure in the genetic local search algorithm. In computer simulations, we used the following genetic local search algorithm.

*Step 0 (Initialization)*

*Step 1 (Evaluation)*

*Step 2 (Selection)*

*Step 3 (Crossover)*

*Step 4 (Mutation)*

*Step 5 (Elitist strategy)*

*Step 6 (Modified local search and termination test)*: Apply the modified local search algorithm to the $N_{\text{pop}}$ solutions in the current population. If a prespecified stopping condition is satisfied during the local search, stop the algorithm. If the local search is completed for all the $N_{\text{pop}}$ solutions, let the set of the obtained $N_{\text{pop}}$ solutions be the current population.

*Step 7 (Termination test)*: Return to Step 1.

### *3.6.2 Genetic simulated annealing*

In the genetic local search algorithm, we can use the simulated annealing instead of the local search to construct a genetic simulated annealing algorithm. That is, we have the genetic simulated annealing algorithm by modifying Step 6 of the genetic local search algorithm in the last subsection. In computer simulations, we applied the simulated annealing algorithm with the constant temperature to each of the $N_{\text{pop}}$ solutions in the current population. We used the constant temperature to avoid extreme deterioration of the current solution during the initial state of annealing with high temperature. The simulated annealing algorithm was iterated 500 times for each solution of the current population in computer simulations. In order to improve the performance of the genetic simulated annealing algorithm, we modify the simulated annealing algorithm by randomly selecting $k$ neighborhood solutions of the current one and letting the best one be the candidate solution for the next transition in the simulated annealing (for such modification of simulated annealing, see Ishibuchi *et al.*[34]).

### *3.6.3 Simulation results*

As in a similar manner to the computer simulations in the last section, we applied the genetic local search algorithm (GLS) and the genetic simulated annealing algorithm (GSA) to the 100 test problems with 20 jobs and 10 machines. The simulation results are shown in Table 3.6. In the GLS, $\alpha$% ($\alpha$ = 100, 75, 50, 25, 10, 5) of the neighborhood solutions of the current solution were examined in each local search procedure. When $\alpha = 100$, the local search algorithm in the GLS is the same as the standard local search algorithm. In the GSA, $k$ ($k = 1$,

**Table 3.6** Performance of the genetic local search and the genetic simulated annealing for 20-job and 10-machine problems.

| Search Algorithms | | Number of evaluations | | |
|---|---|---|---|---|
| | | 10,000 | 50,000 | 200,000 |
| GA | | 101.48 | 101.03 | 100.71 |
| GLS | 100% | 101.14 | 100.14 | 99.85* |
| GLS | 75% | 101.05 | 100.12** | 99.82** |
| GLS | 50% | 101.04* | 100.18 | 99.92 |
| GLS | 25% | 101.02** | 100.19 | 99.97 |
| GLS | 10% | 101.16 | 100.28 | 100.01 |
| GLS | 5% | 101.25 | 100.52 | 100.24 |
| GSA | $k = 1$ | 101.25 | 100.25 | 99.92 |
| GSA | $k = 2$ | 101.18 | 100.20 | 99.93 |
| GSA | $k = 4$ | 101.21 | 100.22 | 99.96 |
| GSA | $k = 6$ | 101.10 | 100.12* | 99.87 |
| GSA | $k = 8$ | 101.26 | 100.23 | 99.92 |
| GSA | $k = 10$ | 101.27 | 100.17 | 99.92 |
| GSA | $k = 20$ | 101.51 | 100.20 | 99.94 |

** and * show the best result and the second best result in each column, respectively.

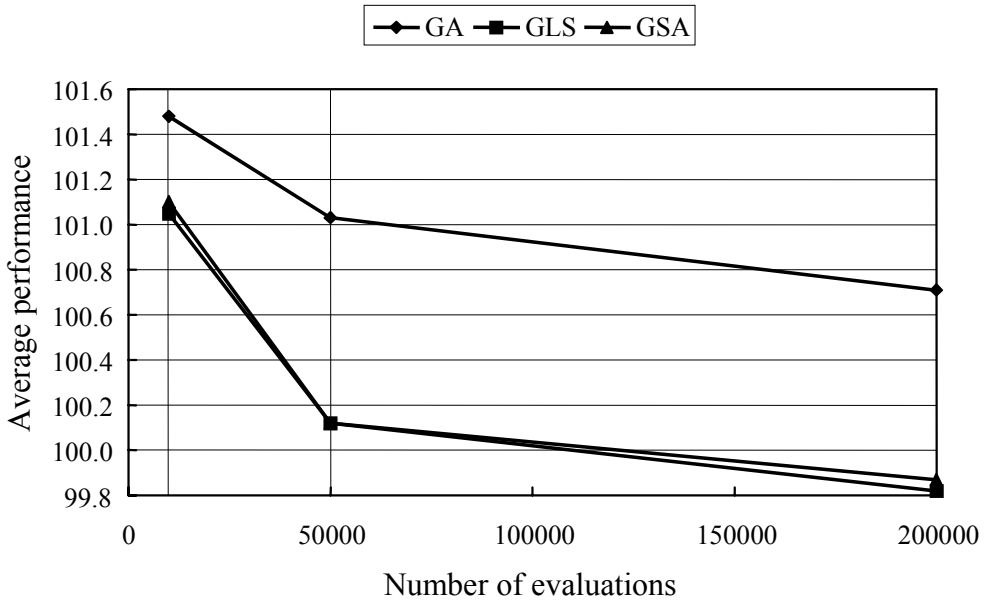


**Fig. 3.13** Simulation results by the GA, the GLS ($\alpha = 75$), and the GSA ($k = 6$).

2, 4, 6, 8, 10, 20) candidate solutions were selected from the neighbor of the current solution, and the transition to the best solution was examined with the acceptance probability in (3.13) of the simulated annealing algorithm. When $k = 1$, the simulated annealing in the GSA is the same as the standard algorithm. In this section, the constant temperature in the GSA was specified as $c = 2$. In Fig. 3.13 we plotted the results obtained by the GA, the GLS ($\alpha = 75$), and the GSA ($k = 6$). From the comparison between the results in Table 3.4 and Table 3.6, we can see that the performance of the genetic algorithm was improved by combining it with local search or simulated annealing. Table 3.6 shows that the modification of the local search procedure in the GLS is effective to improve the performance of the GLS. Table 3.6 also shows that the introduction of $k$ into the simulated annealing improved the performance of the genetic simulated annealing algorithm. From Table 3.4 and Table 3.6, we can see that the best result was obtained by the GLS with $\alpha = 75\%$. It should be noted that not only genetic algorithm but also the local search and the simulated annealing were improved by the hybridization (compare Table 3.4 with Table 3.6).

### *3.6.4 Sensitivity of the performance of the genetic local search algorithm to parameter specifications*

As we have already shown in Section 3.4, careful parameter specifications are required for constructing GAs with high performance. In this section, it is shown that we can construct the GLS without careful parameter specifications.

***A. Specifications of a test problem and a genetic algorithm***

As a test problem, we randomly generated a flowshop scheduling problem with 20 jobs and 10 machines in the same manner as in Subsection 3.4.1. The processing time of each job at each machine was randomly specified as an integer in the closed interval [1, 99]. We applied the genetic algorithm to this test problem 10 times. We employed the two-point order crossover and the shift change mutation as genetic operators in the GA and the GLS. In the GLS, the neighborhood structure based on the shift change was used. We examined the following crossover and mutation probabilities.

$$P_{\mathrm{c}} = 0.1, 0.2, 0.3, \ldots, 1.0, \quad P_{\mathrm{m}} = 0.1, 0.2, 0.3, \ldots, 1.0.$$

Then the number of combination of crossover and mutation probabilities is $10 \times 10 = 100$. In the GLS, we examined $\alpha\%$ neighborhood solutions of the current solution where

$$\alpha = 3, 5, 10, 25, 50, 75, 100.$$

***B. Simulation results***

In this subsection, we employed the modified local search procedure with multiple start solutions (MLS) in order to compare the performance of the GLS. In the case of $\alpha = 100$ in the MLS, the algorithm can be regarded as the standard local search algorithm. Both the algorithms were stopped when 100,000 solutions were evaluated.

Simulation results obtained by the GA and the GLS with 100 probability combinations are shown in Fig. 3.14 and Fig. 3.15. In Fig. 3.14, $\alpha$ was specified as $\alpha = 0$. In this case, the GLS can be regarded as the GA with no local search algorithm. In Fig. 3.15, we specified $\alpha$ as $\alpha = 50$. We used the average makespan over 10 times as a performance measure. From these figures, we can observe that the performance of the GLS does not depend on probability specifications while the performance of the GA highly depends on those. From this observation, we can see that the performance of the GLS is not sensitive to the parameter specifications.

In order to clarify the robustness of the GLS, we plotted the average performance of each algorithm over $\alpha$ in Fig. 3.16. In Fig. 3.16, the best results were obtained by the GLS where the best probability combination was employed. The worst results were obtained by the GLS where the worst combination was employed. The average results were calculated from the results over all results. From this figure, we can observe that the results obtained by the GLS are always better than that of the standard local search algorithm (*i.e.*, $\alpha = 100$ in the MLS). We can also observe that the difference between the best results and the worst results decreases by incorporating the local search in the GAs.

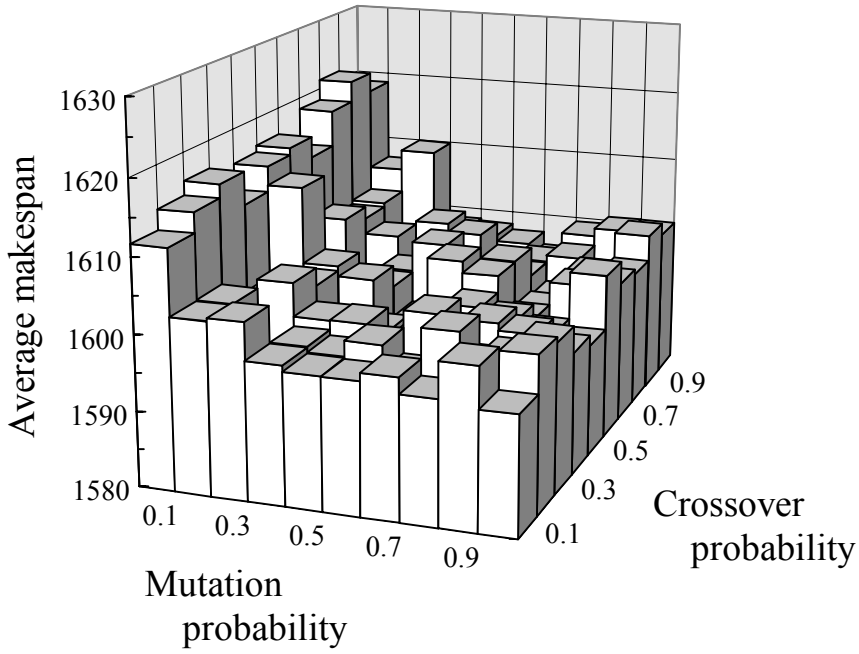


**Fig. 3.14** The average makespan obtained by the GA over 100 probability combinations.

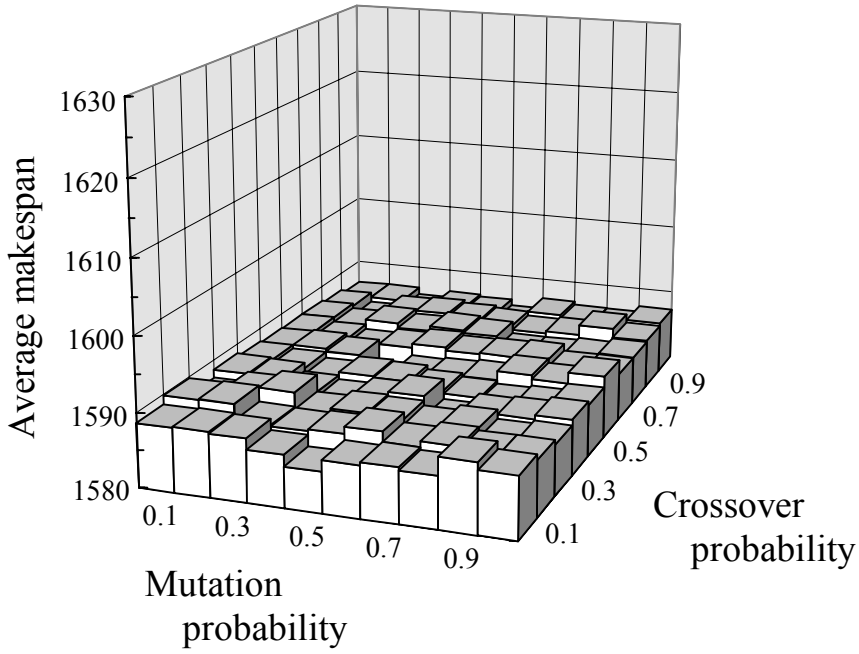


**Fig. 3.15** The average makespan obtained by the GLS over 100 probability combinations.

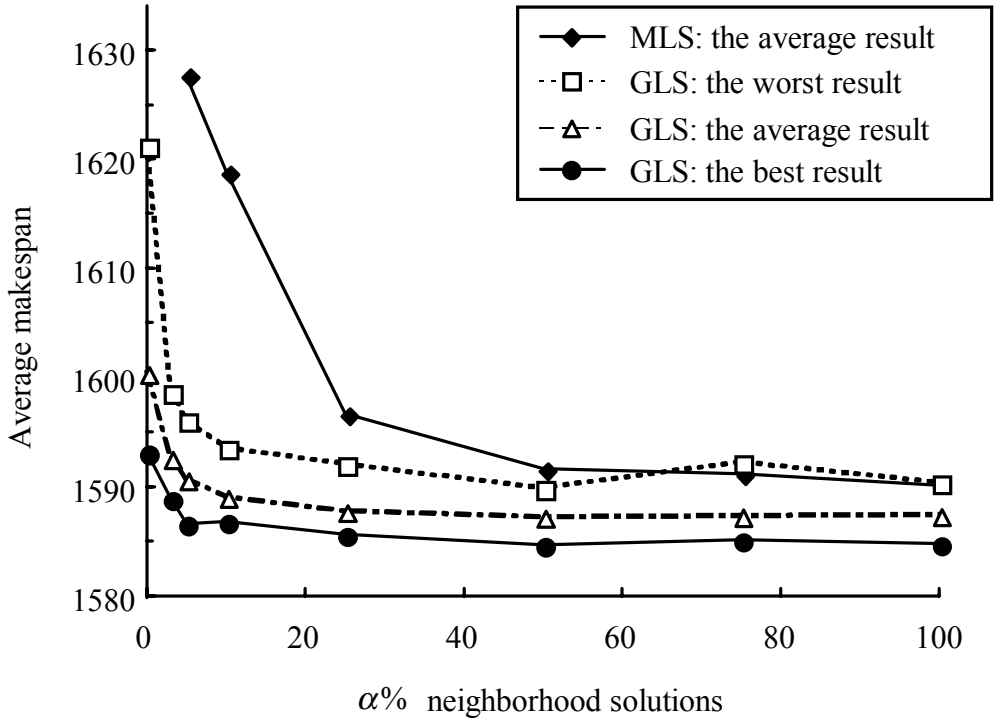


**Fig. 3.16** The relation between $\alpha$ and average performance of the MLS and the GLS.

## 3.7 SUMMARY

In this chapter, we considered genetic algorithms with a single objective for flowshop scheduling problems. First we examined seven crossover operators and five mutation operators for permutation strings. By computer simulations, we pointed out that the combination of high performance crossover and mutation operators did not always lead to a high performance genetic algorithm. We illustrated how high performance genetic algorithms could be constructed by utilizing the positive combination effect of crossover and mutation operators.

From simulation results in this chapter, we can conclude the following:

(i) The combination of high performance crossover and mutation operators did not always mean a high performance genetic algorithm (see, Table 3.1, Table 3.2 and the combination of PBX and Shift in Table 3.3). This suggested the existence of the negative combination effect of crossover and mutation operators.

(ii) A high performance genetic algorithm could be constructed by crossover and mutation operators each of which is not highly evaluated (see, Table 3.1, Table 3.2 and the combination of CX and Adjacent 2 in Table 3.3). This suggested the existence of the positive combination effect of crossover and mutation operators.

(iii) When each of the crossover and mutation probabilities was independently specified by computer simulations using genetic algorithms with either crossover or mutation operator (not both operators), the selected values were not appropriate (*i.e.*, too large) for constructing a high performance genetic algorithm (see the results in Subsection 3.4.6 ( $P_c = 0.9$ , $P_m = 0.4$ ) and Subsection 3.4.7.A ( $P_c = 0.3$ , $P_m = 0.2$ ).

Our intention in this chapter was not to find the best specifications of crossover and mutation of the simple genetic algorithm for a specific class of permutation problems (*i.e.*, flowshop scheduling). We performed many runs of the genetic algorithm with various specifications of crossover and mutation operators in order to clearly demonstrate that the selection of genetic operators and the specifications of their probabilities could not be appropriately done by independent examinations of one genetic operator (*i.e.*, crossover or mutation) from the other. We believe this is true for various classes of combinatorial optimization problems.

Next, we compared the genetic algorithm, which was constructed for flowshop scheduling, with other search algorithms such as local search, simulated annealing [90], and tabu search [108,119]. It was shown that the genetic algorithm was a bit inferior to the other search

algorithms [82]. We examined two hybrid genetic algorithms to improve the performance of the genetic algorithm [82]. One is a genetic local search algorithm and the other is a genetic simulated annealing algorithm. We also introduced some modifications of search mechanisms in these hybrid genetic algorithms [82]. From the computer simulations, we observed that the performance of the GLS did not depend on probability specifications while the performance of the GA highly depended on those.